Data General

Customer Documentation

Commands, Statements, and Functions in Business BASIC

Commands, Statements, and Functions in Business BASIC

093-000351-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000351

Rev. 02, May, 1991

Notice

Commands, Statements, and Functions in Business BASIC
093-000351-02

Revision History:		Effective with:
Original Release	- December, 1984	
First Revision	- December, 1989	
Second Revision	- May, 1991	Business BASIC for AViiON™ Systems, Rev. 1.10 Business BASIC for INTERACTIVE UNIX Systems, Rev. 1.10 AOS Business BASIC, Rev. 4.20 AOS/VS Business BASIC, Rev. 5.20 RDOS Business BASIC, Rev. 8.20 DG/RDOS Business BASIC, Rev. 8.30

A vertical bar in the margin of a page indicates substantive technical change from the previous revision.

Preface

This manual describes how to use the commands, statements, and functions available with Business BASIC. This manual explains these features and how they work on each of the operating systems supporting Business BASIC—AOS, AOS/VS, AOS/VS II, DG/RDOS, RDOS, DG/UX™, and INTERACTIVE UNIX® systems.

Since DG/UX and INTERACTIVE UNIX software are related to UNIX software, this document sometimes refers to those two software products as UNIX products. These references are solely for the purpose of improved readability and occur only where there are no significant differences between DG/UX, INTERACTIVE UNIX, and UNIX software. UNIX® is a registered trademark of AT&T.

This manual is for the experienced Business BASIC programmer who is familiar with the particular operating system being used. The programmer who is not familiar with the operating system should consult the documentation related to the system before using this manual.

NOTE: INTERACTIVE Systems Corporation has replaced the name 386/ix™ with INTERACTIVE UNIX. References to INTERACTIVE UNIX and 386/ix refer to the same product line.

Document Set

Business BASIC is documented by a set of manuals that describe the language, the operating system features that affect its use, and its utilities. This manual is a companion manual to *Subroutines, Utilities, and Business BASIC CLI.* Both of these manuals apply to Business BASIC on all operating system platforms.

Other manuals in the Business BASIC manual set apply only to certain operating systems. For information on the other manuals in the set and their ordering numbers, see the "Related Documents" section at the end of this manual.

Scope

Commands, Statements, and Functions in Business BASIC is a reference manual for experienced Business BASIC programmers. This manual provides the following information for each command, statement, and function in Business BASIC:

- Which operating systems it works with
- How to code it
- What it does
- How to use it

You can use these commands, statements, and functions to make your Business BASIC work easier.

Organization

The Business BASIC statements, commands, and functions are listed in alphabetical order in Chapter 1. The commands used to interface with an INFOS® II data base are listed in Chapter 2.

Terms Used in This Book

The term "AOS/VS" is used to refer to AOS/VS, and AOS/VS II systems. In cases where there are differences between these products, the differences will be noted.

Business BASIC supports two database structures: the PARAM file database structure and the logical file database structure. Certain Business BASIC features work with one database structure and not the other. To distinguish between the two database formats, this manual uses the terms *master file* and *subfile* to refer to files in the PARAM structure and *database file* and *logical file* to refer to files in the logical file structure. These terms are defined below:

Master file	A physical file in the PARAM database structure
Subfile	A file within a master file in the PARAM database structure
Database file	A physical file in the logical file database structure
Logical file	A file within the a database file in the logical file database structure

When the term "search path" is used in this book, it has the following meanings:

AOS/VS systems:	The directories you have selected using the AOS/VS **SEARCHLIST** command.
DG/RDOS systems:	In DG/RDOS systems, you do not have a search path, so Business BASIC searches your directory first and then the library directory ($LIB or $LIB3 for triple precision).
UNIX systems:	The directories you have listed in the UNIX **BBPATH** environment variable.

When the term "switch" is used, it has the following meanings:

AOS/VS and DG/RDOS systems:	A switch that is preceded by a slash.
UNIX systems:	An option that is preceded by a hyphen.

The phrase "Business BASIC user's guide" refers to *Business BASIC System Manager's Guide* if you are using an AOS/VS or DG/RDOS system and *Using Business BASIC on DG/UX™ and INTERACTIVE UNIX® Systems* if you are using a UNIX system.

Coding Conventions

The coding conventions used in this manual are described below.

UPPERCASE BOLD	Indicates a Business BASIC command, statement, or function.
lowercase italics	Indicates a place holder to be replaced by your variable name or literal.
Hyphen (-)	Between italicized words, indicates one complete entry to be supplied by the programmer. Do not enter the hyphen.
{ }	Enclose a part of the format from which you must make a single selection. Do not enter the braces.
[ ]	Enclose an optional part of the format; do not enter the brackets.
...	Indicates that the preceding item can be repeated.

The following boxes indicate whether a command, statement, or function is available on a particular operating system:

AOS/VS	DG/RDOS	UNIX

If only a UNIX box appears above the "Format" section for a command, that command is available only on UNIX systems. If all three boxes appear, the command can be used on all operating systems that run Business BASIC.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining the Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1-800-877-4787 or 1-512-345-5316.

End of Preface

Contents

Chapter 2 Statements Related to INFOS II Files

Tables

Figures

Chapter 1
Commands, Statements, and Functions

The Business BASIC commands, statements, and functions (referred to collectively here as "commands") can often be used on more than one operating system. The features of some commands, however, are available only on certain operating systems. In addition, some examples are coded for a specific operating system but can be modified to work on other operating systems, while other examples work only on the operating systems specified in the examples' descriptions. Also, some terms used in this chapter have different meanings for different operating systems.

Features Available Only on Some Operating Systems

In the "Format" and "Argument" sections of a command description, the features available only on UNIX systems are called out as follows:

string-expression	A string variable, string literal, substring, or string array element (UNIX only) used to receive the error message; it must be dimensioned large enough to contain the error message.

In this example, you can use a string array element for *string-expression* only if you are using a UNIX system.

In other sections of a command description, information that is applicable only for specific operating systems is indicated with a phrase like "On AOS/VS and UNIX systems ...."

How to Use the Examples in This Chapter

Some examples in this chapter use AOS/VS and DG/RDOS conventions, and others use UNIX conventions. Unless a particular example states otherwise, if a command is available on all operating systems, its examples work on all operating systems, provided you make any changes necessary to conform to an operating system's conventions. For example, pathnames in AOS/VS and DG/RDOS use a colon (:) as the pathname delimiter. UNIX systems use a slash (/) as the pathname delimiter unless you specified pathname conversion by including the –P option on the command line you used to execute Business BASIC. To run an AOS/VS example on a UNIX system, you must change the colons to slashes if you did not specify pathname conversion when you executed Business BASIC.

For some commands, the format line shows part of the format enclosed in parentheses ( ), but the example for that command shows the same portion of the format enclosed in brackets [ ]. In these cases, the Business BASIC interpreter accepts either parentheses or brackets as input. If parentheses are used, however, the interpreter changes them to brackets when you save the program.

. (period)

Command

Sends the line in the edit buffer to working storage.

AOS/VS	DG/RDOS	UNIX

Format

. (period)

What It Does

The . **(period)** command is a keyboard edit command that works on the edit buffer. It is part of the "dot editor" group of commands.

When you type a line that causes an error, Business BASIC puts that line into the edit buffer. When you list the contents of working storage, Business BASIC puts the last line listed into the edit buffer. The . **(period)** command sends whatever is currently in the edit buffer to working storage for interpretation.

How to Use It

Use the . **(period)** command if you have used **.A**, **.I**, or **.E** to edit the line in the edit buffer. The . **(period)** command sends only the current contents of the edit buffer to working storage.

Example

Correct an incorrect word.

```
* 10 PRNT "HELLO"
ERROR 2-STATEMENT OR COMMAND SYNTAX IS INVALID
* .E/PRNT/PRINT
10 PRINT "HELLO
* .
10 PRINT "HELLO"
* LIST
00010 PRINT "HELLO"
* RUN
HELLO
```

.A

Command

Appends a string literal to the line in the edit buffer.

AOS/VS	DG/RDOS	UNIX

Format

.A *string*

Arguments

string	A string literal (without quotation marks) that you want to append to the line. You must separate *string* from .A with a delimiter. Any character (including a blank space) that appears after the .A is the delimiter.

What It Does

The .A command is a keyboard edit command that works on the line in the edit buffer. It is part of the "dot editor" group of commands.

When you type a line that causes an error, Business BASIC puts that line into the edit buffer. When you list the contents of working storage, Business BASIC puts the last line listed into the edit buffer. The .A command appends to that line whatever you type for *string*, but it does not send the line back into working storage for interpretation. You can send it back with the . **(period)** command.

How to Use It

The .A command only affects the line currently in the edit buffer. Use the **.P** command to see what is in the edit buffer. If you want to append text to a line that is not in the edit buffer, use the **LIST** *line-number* command to put the line in the edit buffer. (Business BASIC displays the line at your terminal.)

Example

Append a word to an incomplete line.

```
* 10 PRINT
* LIST
00010 PRINT
* .P
00010 PRINT
* .A X
00010 PRINT X
* .
00010 PRINT X
* LIST
00010 PRINT X
```

.C

Command

Changes a string in a line in the edit buffer and passes the line to working storage.

AOS/VS	DG/RDOS	UNIX

Format

AOS/VS and DG/RDOS Systems:

.C/*string1*/*string2* [/G]

UNIX Systems:

$.C/\textit{string1}/\textit{string2} \left[\left\{ \begin{array}{l} /G \\ /n \end{array} \right\} \right]$

Arguments

string1	A string literal (without quotation marks) that is in the line in the edit buffer.
string2	A string literal (without quotation marks) to which you want to change *string1*.
/G	Optional switch to change all occurrences of *string1* within the edit buffer to *string2*. Without this switch, .C changes only the first occurrence.
/*n*	Optional switch that allows you to specify which occurrence of *string1* you want to change to *string2*. You supply a value of 1 to 9 for *n*. The default value is 1. (UNIX only)

What It Does

The .C command is a keyboard edit command that works on the edit buffer. It is part of the "dot editor" group of commands.

The edit buffer contains either the line you just typed (if it caused an error) or the last line listed. The .C command without /G changes the first occurrence of *string1* to *string2*. The .C command with /G changes all occurrences of *string1* to *string2*. On UNIX systems, you can enter the .C command with /*n* in order to change only the *n*th occurrence of *string1* to *string2*. Each of these forms automatically passes the line back to working storage for interpretation. The .C command is the same as the **.E** command except that .C passes the line to working storage.

How to Use It

The .C command changes only the line currently in the edit buffer. Use the **.P** command to see what is in the edit buffer. Use the **LIST** command to put a line in the edit buffer.

Because any character can be used as a delimiter, the .C command can also be used to change a slash (/). For example, the command line below changes a slash to an equal sign. A period is the delimiter.

```
* .C./.=
```

Examples

1. Notice in this example that the line number is an integral part of the statement that is being edited.

   ```
   * 10 PRINT 10
   * LIST
   00010 PRINT 10
   * .C/10/20
   00020 PRINT 10
   ```

   The first occurrence of a 10 (the line number) is changed to 20. You can use this technique to duplicate lines of code.

   ```
   * LIST
   00010 PRINT 10
   00020 PRINT 10
   ```

2. The edit buffer below is empty because .C sent the buffer contents to working storage.

   ```
   * 10 LET X=20
   * LIST
   00010 LET X=20
   * .C/20/30
   00010 LET X=30
   * .P
   Error 73 - Edit buffer is empty
   ```

   Display working storage to see the change.

   ```
   * LIST
   00010 LET X=30
   ```

3. This UNIX example uses the */n* switch to change the second occurrence of string1 to string2.

   ```
   * 10 REM STRING1 STRING1
   * LIST
   00010 STRING1 STRING1
   * .C/STRING1/STRING2/2
   00010 REM STRING1 STRING2
   ```

.E

Command

Changes a string in a line in the edit buffer.

AOS/VS	DG/RDOS	UNIX

Format

AOS/VS and DG/RDOS Systems:

.E/*string1*/*string2* [/G]

UNIX Systems:

$.E/\textit{string1}/\textit{string2} \left[\begin{Bmatrix} /G \\ /n \end{Bmatrix} \right]$

Arguments

string1	A string literal (without quotation marks) that is in the line in the edit buffer.
string2	A string literal (without quotation marks) to which you want to change *string1*.
/G	Optional switch to change all occurrences of *string1* to *string2* within the edit buffer. Without this switch, .E changes only the first occurrence.
/*n*	Optional switch that allows you to specify which occurrence of *string1* you want to change to *string2*. You supply a value of 1 to 9 for *n*. The default value is 1. (UNIX only)

What It Does

The .E command is a keyboard edit command that works on the edit buffer. It is part of the "dot editor" group of commands.

The edit buffer contains either the line you just typed (if it caused an error) or the last line listed. The .E command without /G changes the first occurrence of *string1* to *string2*. The .E command with /G changes all occurrences of *string1* to *string2*. On UNIX systems, you can enter the .E command with /*n* in order to change only the *n*th occurrence of *string1* to *string2*. The .E command is the same as the .C command except that .E does not pass the line to working storage.

How to Use It

The .E command changes only the line currently in the edit buffer. The edit buffer contains either the line you just typed (if it caused an error) or the last line listed. After using .E, you can use the . **(period)** command to send the line back to working storage for interpretation.

continued **.E**

Because any character can be used as a delimiter, the **.E** command can also be used to change a slash (/). For example, the command line below changes a slash to an equal sign. A period is the delimiter.

```
* .E./.=
```

Examples

1. Change the character string "20" to the character string "30".

   ```
   * 10 LET X=20
   * LIST
   00010 LET X=20
   * .E/20/30
   00010 LET X=30
   * .
   ```

   The . **(period)** command sends the change to working storage.

   ```
   * LIST
   00010 LET X=30
   ```

2. Notice that if the . **(period)** command is not used, the original value of 20 remains in working storage.

   ```
   * 10 LET X=20
   * LIST
   00010 LET X=20
   * .P
   00010 LET X=20
   * .E/20/30
   ```

   This changes 20 to 30 in the edit buffer.

   ```
   00010 LET X=30
   ```

   Display the contents of working storage.

   ```
   * LIST
   00010 LET X=20
   ```

3. On UNIX systems, use the */n* switch to change the second occurrence of string1 to string2.

   ```
   * 10 REM STRING1 STRING1
   * LIST
   00010 STRING1 STRING1
   * .E/STRING1/STRING2/2
   00010 REM STRING1 STRING2
   ```

.I

Command

Changes a line in the edit buffer.

AOS/VS	DG/RDOS	UNIX

Format

.I *string*

Arguments

string	A string literal (without quotation marks) that you want to replace the current line in the edit buffer.

What It Does

The **.I** command is a keyboard edit command that changes the line in the edit buffer. It is part of the "dot editor" group of commands.

The edit buffer contains either the line you just typed (if it caused an error) or the last line listed. The **.I** command changes the entire line to *string*, but it does not pass the line to working storage for interpretation.

How to Use It

The **.I** command changes only the line currently in the edit buffer. After using **.I**, you must use the **.** **(period)** command to send the line back to working storage.

Example

Substitute correct syntax for incorrect syntax.

```
* 10 PRINTHELLO
Error 2 - Statement of command syntax is invalid
* .I 10 PRINT "HELLO"
* .
```

The **.** **(period)** command sends the change in the edit buffer to working storage. The corrected command is displayed, and Business BASIC can now execute the command.

```
00010 PRINT "HELLO"
* RUN
HELLO
```

AERM$

Function

Puts an error message into a string.

AOS/VS	DG/RDOS	UNIX

Format

LET *string-variable*=AERM$(*number*)

Arguments

string-variable	A string variable, substring, or string array element (UNIX only) used to receive the error message; it must be dimensioned large enough to contain the error message.
number	The record number in the **BASIC.ER** file for the error message you want, or the code associated with an error message in an AOS/VS parameter file.

What It Does

AERM$ retrieves the error message specified by *number*. The number typically comes from SYS(31) or SYS(42) (AOS/VS and UNIX only).

How to Use It

When you trap errors in your program (by using **ON ERR**), you can have the program print the appropriate error message. Also, you can generate your own unique error messages by adding messages to the end of the **BASIC.ER** file and then using *number* to specify which message you want to retrieve.

You can use **AERM$** only in **LET** statements or commands because you have to assign the error message to a string variable. The largest error message is 64 bytes long.

See **SYS**, **ERM$**, **UERM$** for more information on how to use the error-retrieval functions.

AERM$

continued

Examples

1. When Business BASIC encounters an error in this program, control passes to line 500, where the appropriate error code is selected and the error message associated with that error code is printed.

```
00010  ON ERR THEN GOTO 00500
00020  DIM ER$(64)
. . .
00500  REM ERROR ROUTINE
00510  IF SYS(7)=-60 THEN
00520    LET ER = SYS(31)
00530    LET ER$ = AERM$(ER)
00540  ELSE
00550    LET ER=SYS(7)
00560    LET ER$=ERM$(ER)
00570  END IF
00580  PRINT "ERROR # ";ER;"= ";ER$
00590  END
. . .
```

2. This example uses **ERM$**, **AERM$**, and **UERM$** to retrieve error messages from **SYS(41)**, **SYS(42)**, and **SYS(43)**.

```
01000 REM * error handler
01010 IF SYS(41)=-60 THEN               :Same as SYS(7) and SYS(40)
01012   IF SYS(42)=-276 THEN
01014     LET ER$=UERM$(SYS(43))
01016   ELSE
01020     LET ER$=AERM$(SYS(42))        :Same as SYS(31)
01025   END IF
01030 ELSE
01040   LET ER$=ERM$(SYS(41))           :Same as SYS(7) and SYS(40)
01050 END IF
```

.P

Command

Displays the contents of the edit buffer.

AOS/VS	DG/RDOS	UNIX

Format

.P

What It Does

The **.P** command is a keyboard edit command that displays the contents of the edit buffer. It is part of the "dot editor" group of commands.

When you type a line that causes an error, Business BASIC puts that line in the edit buffer. When you list the contents of working storage, Business BASIC puts the last line listed in the edit buffer. Once a line is in the edit buffer, you can edit it using the **.A**, **.C**, **.E**, **.I**, and **.** **(period)** commands.

How to Use It

The **.P** command displays only the contents of the edit buffer. If the edit buffer is empty, you get `Error 73 - Edit buffer is empty`. To place a line of code in the edit buffer, use the **LIST** command.

Examples

1. In this example, the **PRINT** command is placed into the edit buffer.

```
* 10 PRINT X
* LIST
00010 PRINT X
* .P
00010 PRINT X
```

2. In this example, the buffer is empty because **LIST** has not been used to put the statement into the edit buffer.

```
* 10 PRINT X
* .P
Error 73 - Edit buffer is empty
```

ABS

Function

Returns the absolute value of a number.

AOS/VS	DG/RDOS	UNIX

Format

ABS(*expression*)

Arguments

expression	Any numeric expression.

What It Does

ABS returns the absolute (positive) value of a negative or positive number. If *expression* is a positive number, **ABS** returns the number. If *expression* is negative, **ABS** strips the negative sign from the number and returns the resulting positive number.

How to Use It

You can use the **ABS** function wherever numeric expressions are allowed. *expression* can be:

- A number
- A single, double, triple, or quad precision variable assigned a numeric value
- A numeric array element
- A function
- Any combination of these in a numeric expression with arithmetic or relational operators

Examples

1. Compute and display the absolute value.

   ```
   * PRINT ABS(-30) 30
   ```

2. Use the absolute value in a computation.

   ```
   00010  INPUT X,Y
   00020  FOR I = ABS(X) TO ABS(Y)
   00030    PRINT I
   00040  NEXT I
   00050  STOP
   *RUN
   ? -3,-6
   3
   4
   5
   6
   STOP AT 00050
   *
   ```

AND

Function

Logically compares two expressions.

AOS/VS	DG/RDOS	UNIX

Format

AND(*expression1*,*expression2*)

Arguments

expression1,*expression2*	Numeric expressions or variables to be compared.

What It Does

The **AND** function is used to compare the bit patterns of two binary expressions. The binary representations of the two expressions are compared bit by bit. If a bit is set to 1 in both expressions, that bit is set to 1 in the result.

How to Use It

If, for example, the 16-bit value returned by **SYS(30)** was examined to determine whether the system was triple precision, the **AND** function could be used with **SYS(30)** and a mask which had the bit representing the value 2 set. The result would be 0 if the system was not triple precision, or 2 if the system was triple precision.

To check several bits at once, add the masks for the bits together and check the result of the **AND** comparison of the summed mask with the value to be examined. You can use the **AND** function in any numeric expression.

Figure 1–1 shows the result when the bit values of two expressions are compared using AND. Figure 1–2 shows the decimal values for each power of 2 from 0 to 63.

AND (192, 64)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Power of 2
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	$expr_1$ = 192
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	$expr_2$ = 64
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	RESULT = 64

Figure 1–1 Logical AND of Two Numbers

AND

continued

2^n	Decimal Value	2^n	Decimal Value
2^{31}	2147483648	2^{63}	9223372036854775808
2^{30}	1073741824	2^{62}	4611686018427387904
2^{29}	536870912	2^{61}	2305843009213693952
2^{28}	268435456	2^{60}	1152921504606846976
2^{27}	134217728	2^{59}	576460752303423488
2^{26}	67108864	2^{58}	288230376151711744
2^{25}	33554432	2^{57}	144115188075855872
2^{24}	16777216	2^{56}	72057594037927936
2^{23}	8388608	2^{55}	36028797018963968
2^{22}	4194304	2^{54}	18014398509481984
2^{21}	2097152	2^{53}	9007199254740992
2^{20}	1048576	2^{52}	4503599627370496
2^{19}	524288	2^{51}	2251799813685248
2^{18}	262144	2^{50}	1125899906842624
2^{17}	131072	2^{49}	562949953421312
2^{16}	65536	2^{48}	281474976710656
2^{15}	32768	2^{47}	14073748835528
2^{14}	16384	2^{46}	70368744177664
2^{13}	8192	2^{45}	35184372088832
2^{12}	4096	2^{44}	17592186044416
2^{11}	2048	2^{43}	8796093022208
2^{10}	1024	2^{42}	4398046511104
2^{9}	512	2^{41}	2199023255552
2^{8}	256	2^{40}	1099511627776
2^{7}	128	2^{39}	549755813888
2^{6}	64	2^{38}	274877906944
2^{5}	32	2^{37}	137438953472
2^{4}	16	2^{36}	68719476736
2^{3}	8	2^{35}	34359738368
2^{2}	4	2^{34}	17179869184
2^{1}	2	2^{33}	8589934592
2^{0}	1	2^{32}	4292967296

Figure 1-2 Powers of 2 (0 to 63)

Examples

1. Check whether this is a triple precision system.

```
00010 IF AND(SYS(30),2) THEN
00020   PRINT "This is a triple precision system"
00030 ELSE
00040   PRINT "This is not a triple precision system"
00050 END IF
```

2. On AOS/VS or DG/RDOS system, check for several of the possible switches used when Business BASIC was brought up. The /A switch is in the leftmost bit of the 32-bit value returned by **SYS(11)**; this bit is represented by the value 2^{31}.

```
00010 DIM A$(26)
00020 LET SWTCHS=SYS(11)
00030 IF AND(SWTCHS,2^28) THEN LET A$[0]="/D"
00040 IF AND(SWTCHS,2^11) THEN LET A$[0]="/U"
00050 PRINT "Some of the switches used were: ";A$
```

On a UNIX system, you would enter the following code for this example:

```
00010 DIM A$[72]
00020 LET SWTCH=SYS(11)
00030 IF AND(SWTCH,2^28) THEN LET A$[0]="D"
00040 IF AND(SWTCH,2^60) THEN LET A$[0]="d"
00050 PRINT "Some of the switches used were: ";A$
```

3. If the bits represented by the values 64 and 32 are both set, go to line 100; if one or the other bit is set, go to line 200; otherwise, fall through to line 50.

```
00030 IF AND(X,64+32)=64+32 THEN GOTO 00100
00040 IF AND(X,64+32) THEN GOTO 00200
00050 REM Neither of the two bits was set
```

AND

Operator

Boolean logical AND — joins two expressions.

AOS/VS	DG/RDOS	UNIX

Format

expression AND *expression*

Arguments

expression	A numeric or relational expression or a variable. If either expression evaluates to 0, the result of joining the expressions is false; otherwise, the result is true. Relational expressions are two numeric or string expressions separated by a relational operator. The relational operators are greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), equal to (=), and not equal to (<>).

What It Does

AND joins two expressions into a single expression. When you execute a **LET** statement, the expressions joined by the **AND** function are evaluated and reduced to 0 or 1; the result of the **AND** is 1 if both expressions are non-zero and is 0 otherwise. When you execute an **IF** statement, the expressions united by the **AND** are evaluated. Any false expression makes the entire **AND** expression false.

NOTE: An **AND** function that logically compares two bit patterns is also available and is described separately.

How to Use It

AND may be used any place a numeric expression is valid.

Before the Boolean logic operator is executed, the expressions are evaluated as true or false, and the operands are reduced to 1 or 0, respectively.

The precedence of all operators is given below.

```
highest      ^ ( exponential)
.            unary +, unary -, NOT
.            *, /
.            +, -
.            <>, >, >=, =, <=, <
.            AND
lowest       OR
```

Example

When two conditions are met (evaluate to a true value), execute the **GOTO**.

```
00010 IF A>1 AND B=2 THEN GOTO 00100
```

ASC

Function

Gives the ASCII value of a string.

AOS/VS	DG/RDOS	UNIX

Format

ASC(*string-expression*)

Arguments

string-expression	A string variable, string literal, substring, or string array element (UNIX only).

What It Does

The ASC function transfers bytes directly from a string to a numeric variable. Each byte of a string contains the ASCII value of the character it represents; the several bytes of a numeric variable contain a binary number which translates directly into the decimal value displayed by **PRINT**. The number of characters in a string that **ASC** can transfer depends on the precision you are using. Numeric variables hold data in four bytes in a double precision system, six bytes in triple precision, and 8 bytes in quad precision. If a string field is shorter than the maximum for the variable's precision, the result is justified toward the low-order bytes of the numeric variable; thus, **ASC("A")** is 65 while **ASC("A<0><0><0>")** is $65*2^{24}$.

If you include the **-X** option when you execute a UNIX system, the **ASC** function performs identically to the **ASX** function.

How to Use It

Use the ASC function wherever numeric expressions are allowed.

ASC may not change the sign of the value in *string-expression*. If you use **CHR$** to put a negative value in a string, and then use **ASC** to extract the value and place it in a variable that is less than four bytes in double precision, six bytes in triple precision, or eight bytes in quad precision, you must correct the value's sign. However, on UNIX systems, when you execute Business BASIC with the **-X** option, the sign is included.

It is sometimes more efficient to pass data to or from files or the common area as strings. If you use the **CHR$** function to put a binary value into a string, you should use the ASC function to extract the binary value from that string.

You can use **PACK** and **UNPACK** instead of ASC and **CHR$**. With **PACK** and **UNPACK**, you have more formatting flexibility (you can use **RFORM**) and you do not need to convert numeric signs. No matter what precision you use, Business BASIC retains the sign.

continued **ASC**

Examples

1. Line 30 makes a four-byte string of integer B, and line 100 extracts B from A$.

```
00010  DIM A$(12)
00020  INPUT B
00030  LET A$(1,4)=CHR$(B,4)
. . .
00100  LET B=ASC(A$(1,4))
. . .
```

2. Display the decimal ASCII value for the character "A".

```
* PRINT ASC("A") 65

*
```

3. This is an example for a triple precision system.

```
00010  DIM A$(6)
00020  LET X#=123456789012
00030  LET A$=CHR$(X#,6)
00040  LET ANS = ASC(A$)
00050  PRINT ANS
* RUN
123456789012
```

4. This example uses a negative number in a double precision system.

```
00010 DIM X$[9]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00040 LET A%=ASC(X$[1,2])
00050 PRINT "ASC value of a negative value stored in a 2-byte
          string = ";A%
00060 LET A%=OR(A%,-AND(A%,2^15))
00070 PRINT "           value AFTER the sign correction = ";A%
00080 LET A=ASC(X$[3,5])
00090 PRINT "ASC value of a negative value stored in a 3-byte
          string = ";A
00100 LET A=OR(A,-AND(A,2^15))
00110 PRINT "           value AFTER the sign correction = ";A
00120 LET A=ASC(X$[6,9])
00130 PRINT "ASC value of a negative value stored in a 4-byte
          string = ";A
```

ASC

continued

```
* RUN

ASC value of a negative value stored in a 2-byte
         string =  65492
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 3-byte
         string = 16777172
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 4-byte
         string =  -44
```

5. This example uses a negative number in a triple precision system.

```
00010 DIM X$[15]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00038 LET X$[10,15]=CHR$(-44,6)
00040 LET A%=ASC(X$[1,2])
00050 PRINT "ASC value of a negative value stored in a 2-byte
         string = ";A%
00060 LET A%=OR(A%,-AND(A%,2^15))
00070 PRINT "          value AFTER the sign correction = ";A%
00080 LET A=ASC(X$[3,5])
00090 PRINT "ASC value of a negative value stored in a 3-byte
         string = ";A
00100 LET A=OR(A,-AND(A,2^15))
00110 PRINT "          value AFTER the sign correction = ";A
00120 LET A=ASC(X$[6,9])
00130 PRINT "ASC value of a negative value stored in a 4-byte
         string = ";A
00140 LET A=OR(A,-AND(A,2^31))
00150 PRINT "          value AFTER the sign correction = ";A
00160 LET A=ASC(X$[10,15])
00170 PRINT "ASC value of a negative value stored in a 6-byte
         string = ";A
```

```
* RUN

ASC value of a negative value stored in a 2-byte
         string =  65492
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 3-byte
         string =16777172
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 4–byte
         string = 4294967252
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 6-byte
         string = -44
```

6. This example uses a negative number in a quad precision system.

```
00010 DIM X$[23]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00038 LET X$[10,15]=CHR$(-44,6)
00039 LET X$[16,23]=CHR$(-44,8)
00040 LET A%=ASC(X$[1,2])
00050 PRINT "ASC VALUE OF A NEGATIVE VALUE STORED IN A 2-BYTE
           STRING = ";A%
00060 LET A%=OR(A%,-AND(A%,2^15))
00070 PRINT "          VALUE AFTER THE SIGN CORRECTION = ";A%
00080 LET A=ASC(X$[3,5])
00090 PRINT "ASC VALUE OF A NEGATIVE VALUE STORED IN A 3-BYTE
           STRING = ";A
00100 LET A=OR(A,-AND(A,2^15))
00110 PRINT "         VALUE AFTER THE SIGN CORRECTION = ";A
00120 LET A=ASC(X$[6,9])
00130 PRINT "ASC VALUE OF A NEGATIVE VALUE STORED IN A 4-BYTE
           STRING = ";A
00140 LET A=OR(A,-AND(A,2^31))
00150 PRINT "         VALUE AFTER THE SIGN CORRECTION = ";A
00160 LET A=ASC(X$[10,15])
00170 PRINT "ASC VALUE OF A NEGATIVE VALUE STORED IN A 6-BYTE
           STRING = ";A
00180 LET A=OR(A,-AND(A,2^47))
00190 PRINT "         VALUE AFTER THE SIGN CORRECTION = ";A
00200 LET A=ASC(X$[16,23])
00210 PRINT "ASC VALUE OF A NEGATIVE VALUE STORED IN AN 8-BYTE
           STRING = ";A
```

```
* RUN

ASC value of a negative value stored in a 2-byte
         string =  65492
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 3-byte
         string =16777172
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 4-byte
         string = 4294967252
         value AFTER the sign correction = -44
ASC value of a negative value stored in a 6-byte
         string = 281474976710612
         value AFTER the sign correction = -44
ASC value of a negative value stored in an 8-byte
         string = -44
```

ASX

Function

Gives the ASCII value of a string with automatic sign correction.

AOS/VS	UNIX

Format

ASX(*string-expression*)

Arguments

string-expression	A string variable, string literal, substring, or string array element (UNIX only).

What It Does

The ASX function transfers bytes directly from a string to a numeric variable. Each byte of a string contains the ASCII value of the character it represents; the several bytes of a numeric variable contain a binary number that translates directly into the decimal value displayed by **PRINT**. The number of characters in a string that ASX can transfer depends on the precision you are using. Numeric variables hold data in four bytes in a double precision system, six bytes in triple precision, and eight bytes in quad precision. If a string field is shorter than the maximum for the variable's precision, the result is justified toward the low-order bytes of the numeric variable; thus, ASX("A") is 65 while ASX("A<0><0><0>") is $65*2^{24}$.

NOTE: On UNIX systems, if you execute Business BASIC with the –X option, the ASC function and the ASX function perform identically.

How to Use It

Use the **ASX** function wherever numeric expressions are allowed.

ASX will correct the sign of the value in *string-expression* if necessary. If you use **CHR$** to put a negative value in a string, and then use **ASX** to extract the value and place it in a variable that is less than four bytes in double precision, six bytes in triple precision, or eight bytes in quad precision, you do not need to correct the sign as you would with the ASC function. It is sometimes more efficient to pass data to or from files or the common area as strings. If you use the **CHR$** function to put a binary value into a string, you would use either the **ASX** or the **ASC** function to extract the binary value from that string.

You can use **PACK** and **UNPACK** instead of **ASX** and **CHR$**. With **PACK** and **UNPACK**, you have more formatting flexibility (you can use **RFORM**).

Examples

1. Line 30 makes a four-byte string of integer B, and line 100 extracts B from A$.

```
00010  DIM A$(12)
00020  INPUT B
00030  LET A$(1,4)=CHR$(B,4)
. . .
00100  LET B=ASX(A$(1,4))
. . .
```

2. Display the decimal ASCII value for the character "A".

```
* PRINT ASX("A") 65

*
```

3. This is an example for a triple precision system.

```
00010  DIM A$(6)
00020  LET X#=123456789012
00030  LET A$=CHR$(X#,6)
00040  LET ANS = ASX(A$)
00050  PRINT ANS

* RUN
123456789012
```

4. This example uses a negative number in a double precision system.

```
00010 DIM X$[9]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00040 LET A%=ASX(X$[1,2])
00050 PRINT "ASX value of a negative value stored in a 2-byte
       string = ";A%
00080 LET A=ASX(X$[3,5])
00090 PRINT "ASX value of a negative value stored in a 3-byte
       string = ";A
00120 LET A=ASX(X$[6,9])
00130 PRINT "ASX value of a negative value stored in a 4-byte
       string = ";A

* RUN
ASX value of a negative value stored in a 2-byte string =  -44
ASX value of a negative value stored in a 3-byte string =  -44
ASX value of a negative value stored in a 4-byte string =  -44
```

ASX

continued

5. This example uses a negative number in a triple precision system.

```
00010 DIM X$[15]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00038 LET X$[10,15]=CHR$(-44,6)
00040 LET A%=ASX(X$[1,2])
00050 PRINT "ASX value of a negative value stored in a 2-byte
       string = ";A%
00080 LET A=ASX(X$[3,5])
00090 PRINT "ASX value of a negative value stored in a 3-byte
       string = ";A
00120 LET A=ASX(X$[6,9])
00130 PRINT "ASX value of a negative value stored in a 4-byte
       string = ";A
00160 LET A=ASX(X$[10,15])
00170 PRINT "ASX value of a negative value stored in a 6-byte
       string = ";A
```

```
* RUN
ASX value of a negative value stored in a 2-byte string = -44
ASX value of a negative value stored in a 3-byte string = -44
ASX value of a negative value stored in a 4-byte string = -44
ASX value of a negative value stored in a 6-byte string = -44
```

6. This example uses a negative number in a quad precision system (UNIX systems only).

```
00010 DIM X$[23]
00020 LET X$[1,2]=CHR$(-44,2)
00030 LET X$[3,5]=CHR$(-44,3)
00035 LET X$[6,9]=CHR$(-44,4)
00038 LET X$[10,15]=CHR$(-44,6)
00039 LET X$[16,23]=CHR$(-44,8)
00040 LET A%=ASX(X$[1,2])
00050 PRINT "ASX VALUE OF A NEGATIVE VALUE STORED IN A 2-BYTE
       STRING = ";A%
00080 LET A=ASX(X$[3,5])
00090 PRINT "ASX VALUE OF A NEGATIVE VALUE STORED IN A 3-BYTE
       STRING = ";A
00120 LET A=ASX(X$[6,9])
00130 PRINT "ASX VALUE OF A NEGATIVE VALUE STORED IN A 4-BYTE
       STRING = ";A
00160 LET A=ASX(X$[10,15])
00170 PRINT "ASX VALUE OF A NEGATIVE VALUE STORED IN A 6-BYTE
       STRING = ";A
00200 LET A=ASX(X$[16,23])
00210 PRINT "ASX VALUE OF A NEGATIVE VALUE STORED IN AN 8-BYTE
       STRING = ";A
```

continued **ASX**

```
* RUN
ASX VALUE OF A NEGATIVE VALUE STORED IN A 2-byte STRING = -44
ASX VALUE OF A NEGATIVE VALUE STORED IN A 3-byte STRING = -44
ASX VALUE OF A NEGATIVE VALUE STORED IN A 4-byte STRING = -44
ASX VALUE OF A NEGATIVE VALUE STORED IN A 6-byte STRING = -44
ASX VALUE OF A NEGATIVE VALUE STORED IN A 8-byte STRING = -44
```

BBSTAT

Statement and Command

Displays the status of all Business BASIC jobs.

AOS/VS	UNIX

Format

BBSTAT

What It Does

BBSTAT displays the following information about all current Business BASIC processes:

- PID
- Username
- Console name
- Current program name

How to Use It

BBSTAT may be used in keyboard mode or as a statement.

If there is more data than can be displayed on one screen, Business BASIC prompts you with the word MORE at the bottom of the screen. At this point, press any key to scroll down one screen.

Examples

1. On AOS/VS systems, **BBSTAT** produces the following display, which always includes information on the Business BASIC process issuing the **BBSTAT** command. Processes are sorted by PID, in ascending order. Each section is described after the display.

```
* BBSTAT
   49  FRANK6  VCON4             SCRATCH
   58  BARRY8  CON7              AR002
   61  JOHNS8  CON24             AP005
   70  TEMP 8  CON32             MAINMENU

   a.  b.      c.                d.
```

Column	Description
a.	The PID number
b.	Business BASIC user name
c.	Console name
d.	Program name

continued

BBSTAT

2. This example shows the BBSTAT display that appears on a UNIX system. The columns and their descriptions are the same as they were in the previous example.

```
* BBSTAT
  7895  BARRY8  /dev/ttyp1      SCRATCH
  8213  FRANK6  /dev/tty01      ARMENU
  8226  JOHNS8  /dev/ttyp4      AP001
  8237  MARY 8  /dev/tty10      GLMENU
```

BLOCK READ

Statement and Command

Retrieves blocks from a file or common area.

AOS/VS	DG/RDOS	UNIX

Format

BLOCK READ [FILE(*channel*[,*start*]),]*variable*

Arguments

channel	Channel number of a file, expressed as a number or numeric expression.
start	Block number from which to start reading, if reading from a file. *start* can be a number or numeric expression. This argument is optional. When it is not given, the value 0 is used to indicate the beginning of the file.
variable	A string variable or string array element (UNIX only) dimensioned to at least 512 bytes, or an array variable for an array that holds at least 512 bytes. `Error 55 - Block I/O error` is generated if the variable is less than 512 bytes.

What It Does

Use **BLOCK READ** to read data from the common area, and **BLOCK READ FILE** to read data from a file. Blocks are 512 bytes. Whenever you use **BLOCK READ** to read from a file or common area, you read no less than 512 bytes into your variable. See Appendix B for the size of the common area on your operating system.

BLOCK READ FILE reads as many blocks from a file as the string variable or array can hold. If string variable A$ is dimensioned to 1,536 bytes, **BLOCK READ FILE** reads three blocks into it; however, if A$ is 1,535 bytes long, only two blocks are read. **BLOCK READ FILE** starts reading blocks sequentially from the block number you specify in *start*. The byte position in the file (**GPOS**) after a **BLOCK READ FILE** is undefined.

Under DG/RDOS, if the string variable is larger than the number of bytes read, the variable is padded with nulls; under AOS/VS and UNIX it is not. For example, if you dimension A$ to 512 bytes, but your program reads only 256 bytes, under DG/RDOS the last 256 bytes of A$ contain nulls; under AOS/VS and UNIX, they contain what they contained prior to the **BLOCK READ**. If you write programs under AOS/VS or UNIX, you may want to fill the string with nulls prior to the **BLOCK READ** to avoid having garbage in the last part of the string after reading the last block in a file.

BLOCK READ (without **FILE**) reads from the common area. (You cannot specify *start* and *channel* without **FILE**.) Each job has its own common area that can be used to pass information between programs during **SWAP** and **CHAIN** statements. See Appendix B for the size of the common area on your operating system.

BLOCK READ FILE is equivalent to DG/RDOS block I/O—not buffered. There could be a problem if the same file is opened for another type of I/O that is buffered. This is not a Business BASIC problem, but rather a problem inherent with combining buffered and unbuffered I/O to the same file. The statements in Business BASIC that use buffered I/O are **READ FILE, WRITE FILE, LREAD FILE, LWRITE FILE, PRINT FILE,** and **INPUT FILE**. Do not mix the **BLOCK READ/WRITE FILE** statements with the buffered I/O statements.

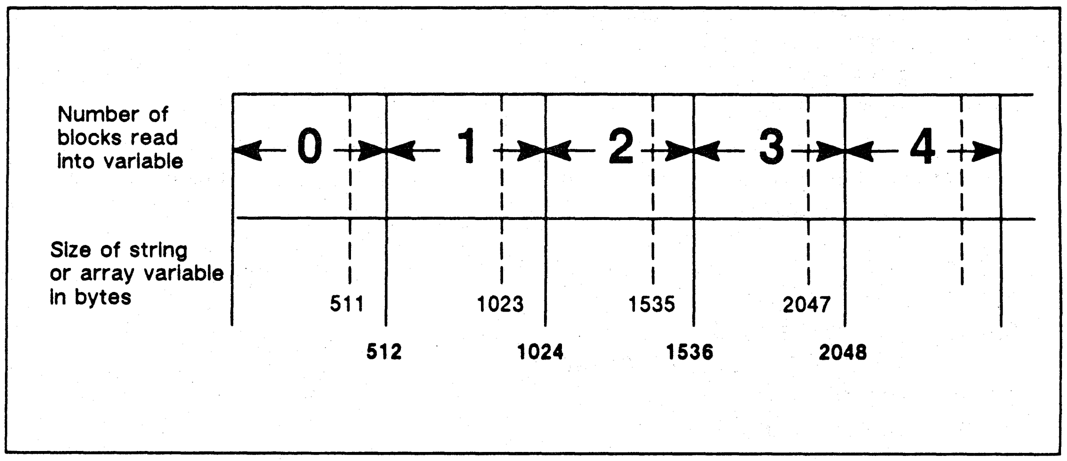

Number of blocks read into variable
0
1
2
3
4
Size of string or array variable in bytes
511
1023
1535
2047
512
1024
1536
2048

Figure 1-3 BLOCK READ

How to Use It

To read blocks from a file, first open the file and assign a channel number to it. You can specify the block number at which to begin reading in the *start* field. **BLOCK READ FILE** reads sequentially from that point.

The size of the variable determines the number of blocks that are transferred. String variables must be dimensioned to at least 512 bytes. If you make them larger, **BLOCK READ FILE** reads in as many entire blocks as the string variable can hold. The same is true for arrays. For numeric arrays, each element holds 4 bytes in a double precision system, 6 bytes in a triple precision system, and 8 bytes in a quadruple precision system.

The data you put in the common area stays there until you log off or until a program overwrites the common area with **BLOCK WRITE**. Many utilities and CLI commands use **BLOCK WRITE** and thus overwrite the common area.

BLOCK READ

continued

Examples

1. Line 30 sends X$ into the common area, where **OPEN** finds it. **OPEN** uses a **BLOCK READ** to read X$, then puts a new X$ back into the common area with a **BLOCK WRITE**. Line 50 reads the new X$.

```
00005  REM--WRITE X$ TO COMMON
00007  REM--WHERE OPEN MAY FIND IT.
00010  DIM X$(512)
00020  LET X$="SUB1,5,SUB2,5,PHYS,6",FILL$(0)
00030  BLOCK WRITE X$
00040  SWAP "OPEN"
00050  BLOCK READ X$
```

2. Line 40 starts at block 0 in file 0 and reads 2 blocks into AR. Array AR is 4 times 64, or 256 elements.

```
00010  REM -- READ BLOCKS FROM "INFO" INTO ARRAY AR
00020  DIM AR(3,63)
00030  OPEN FILE(0,0),"INFO"
00040  BLOCK READ FILE(0,0),AR
00050  PRINT AR
00060  CLOSE FILE(0)
```

BLOCK WRITE

Statement and Command

Outputs blocks of data to a file or common area.

AOS/VS	DG/RDOS	UNIX

Format

BLOCK WRITE [**FILE**(*channel*[,*start*]),]*variable*

Arguments

channel	Channel number of a file, expressed as a number or numeric expression.
start	The number of the first block in the file to receive output, expressed as a number or numeric expression. This argument is optional. When it is not given, the value 0 is used to indicate the beginning of the file.
variable	A string variable or string array element (UNIX only) dimensioned to at least 512 bytes, or an array variable that holds at least 512 bytes. Error number 55 is generated if the variable is less than 512 bytes.

What It Does

BLOCK WRITE writes data to the common area. **BLOCK WRITE FILE** writes blocks of data to a file. A block is 512 bytes. Whenever you use **BLOCK WRITE** to write to a file or to the common area, you write at least 512 bytes of data. See Appendix B for the size of the common area on your operating system.

BLOCK WRITE FILE writes blocks to a file indicated by channel beginning at the block specified by *start*. **BLOCK WRITE FILE** writes as many blocks to a file as *variable* can hold. If string-variable A$ is dimensioned to 1,536 bytes, **BLOCK WRITE FILE** writes three blocks into it. However, if A$ is 1,535 bytes long, only two blocks are written.

For numeric arrays, each element holds 4 bytes in a double precision system, 6 bytes in a triple precision system, and 8 bytes in a quadruple precision system.

You can start at any block number and write sequentially from that point. The file position pointer (GPOS) is undefined after a **BLOCK WRITE FILE** statement.

BLOCK WRITE (without **FILE**) outputs the contents of *variable* to the common area. Figure 1–4 shows the number of blocks written from *variable*, depending on the number of bytes in the string or array variable.

BLOCK WRITE

continued

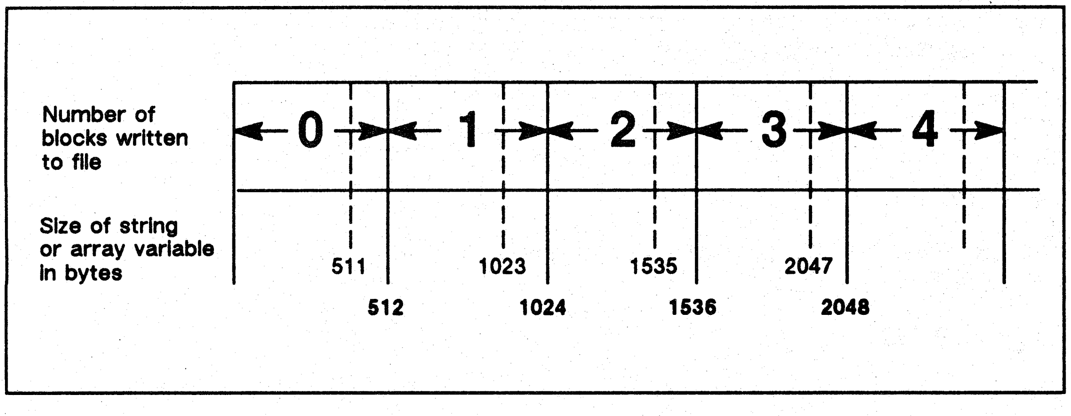

Number of blocks written to file
0
1
2
3
4
Size of string or array variable in bytes
511
1023
1535
2047
512
1024
1536
2048

Figure 1-4 BLOCK WRITE

BLOCK WRITE is equivalent to DG/RDOS block I/O (not buffered). A problem might occur if the same file is opened for another type of I/O that is buffered. This is not a Business BASIC problem, but rather a problem inherent with combining buffered and unbuffered I/O to the same file under DG/RDOS. The statements in Business BASIC that use buffered I/O are **READ FILE, WRITE FILE, LREAD FILE, LWRITE FILE, PRINT FILE,** and **INPUT FILE.** Do not mix the **BLOCK READ/WRITE FILE** statements with the buffered I/O statements.

How to Use It

To write blocks to a file, you must open the file and assign a channel number to it. You can specify where to begin the write operation in the file by indicating a block number in *start*. The default for *start* is the first block in the file.

The size of *variable* determines the number of blocks written. If the variable is 512 bytes or more in length, but less than 1,024 bytes, only one block is written (i.e., only the first 512 bytes of *variable* are written). If variable is 1,024 bytes or more, but less than 1,536 bytes, only two blocks are written, etc.

The information you send to the common area with a **BLOCK WRITE** stays in the common area until you do another **BLOCK WRITE** to the common area or until you log off. The common area is always open and ready for use. Many utilities and CLI commands use **BLOCK WRITE** and thus overwrite the common area.

Examples

1. Line 30 sends X$ into the common area, where **OPEN** finds it. **OPEN** uses a **BLOCK READ** to read X$, then puts a new X$ back into the common area with a **BLOCK WRITE**. Line 50 reads the new X$.

```
00005  REM--WRITE X$ TO COMMON
00007  REM--WHERE OPEN MAY FIND IT.
00010  DIM X$(512)
00020  LET X$="SUB1,5,SUB2,5,PHYS,6",FILL$(0)
00030  BLOCK WRITE X$
00040  SWAP "OPEN"
00050  BLOCK READ X$
```

2. This example reads 2 blocks from **TEMP** and writes them to **FINAL**, using N for a block counter.

```
00010  OPEN FILE (0,0),"TEMP"
00020  OPEN FILE (1,0),"FINAL"
00030  LET N=1
00040  DIM X$(1024)
00050  BLOCK READ FILE(0,N),X$
. . .
00200  BLOCK WRITE FILE(1,N),X$
00210  LET N=N+2
00220  GOTO 00080
. . .
```

BREAK

Statement

Terminates the DO loop currently executing.

AOS/VS	UNIX

Format

BREAK

What It Does

The **BREAK** statement terminates the most recently started **DO** loop and transfers control to the statement following the end of the loop. If no matching **END LOOP**, **WHILE**, or **UNTIL** statement terminates the loop, `Error 96 – DO with no matching END LOOP, WHILE or UNTIL` is raised.

How to Use It

You can code the **BREAK** statement within or, by using a **GOTO** statement, outside the **DO** loop you want to terminate. In either case, execution resumes at the statement following the end of the loop. Both methods are illustrated below.

Examples

1. The **BREAK** statement is within the **DO** loop.

```
00010  X=1
00020  DO
00030    PRINT "I'm in the DO loop"
00040    LET X=X+1
00050    IF X=5 THEN BREAK
00060  WHILE (X<10)
00070  PRINT "This line prints after the BREAK statement."
00080  STOP

* RUN
I'm in the DO loop
I'm in the DO loop
I'm in the DO loop
I'm in the DO loop
This line prints after the BREAK statement.
Stop at 00080
```

continued **BREAK**

2. Here **BREAK** is used to terminate a **DO** loop that reads single characters from a data file.

```
00010 DIM A$[1],ER$[80]
00020 OPEN FILE[0],"example"      : OPEN data file
00030 ON ERR THEN BREAK           : BREAK out of loop when we get
                                  : an error
00040 DO                          : \
00050   READ FILE[0],A$           :  \  DO loop
00060   WRITE A$                  :  /
00070 END LOOP                    : /
00080 IF SYS(7)<>-6 THEN          : Report any errors other than
                                  : end of file
00090   LET ER$=ERM$(SYS(7))
00100   PRINT "Error: ";ER$
00110   STOP
00120 ELSE
00130   PRINT "End of file reached"
00140   CLOSE FILE[0]
00150 END IF
00160 END

* RUN
Here is line 1 of the file
Here is line 2 of the file
Here is line 3 of the file
Here is line 4 of the file
Here is line 5 of the file
End of file reached

*
```

BYE

Statement and Command

Logs a user out of Business BASIC.

AOS/VS	DG/RDOS	UNIX

Format

BYE

What It Does

BYE terminates a Business BASIC session. In DG/RDOS, typing **BYE** returns you to the logon banner. In AOS/VS and UNIX systems, **BYE** terminates your current Business BASIC process. For example, if you go to Business BASIC from the AOS/VS CLI, **BYE** returns you to the AOS/VS CLI. If you log on to Business BASIC directly, **BYE** logs you off. On UNIX systems, **BYE** returns you to the calling shell process.

How to Use It

Type **BYE** as a keyboard command, or precede it with a line number as a program statement.

CHAIN

Statement and Command

Executes a utility or another program.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{CHAIN}\left\{\begin{matrix}\textit{string-expression}\\ \textit{“filename”}\end{matrix}\right\}\text{THEN}\left[\left\{\begin{matrix}\text{GOTO } \textit{line-number}\\ \text{CON}\end{matrix}\right\}\right]$$

Arguments

filename	A literal for a saved file but not for an ASCII listing or source file. The file must contain a Business BASIC program.
string-expression	A string variable, string literal, or string array element (UNIX only) you've already dimensioned and to which you have assigned the value of *filename*.
line-number	A valid line number in the program you are chaining to. Execution begins at the line number rather than the beginning of the program, and the program's variables retain the values they had when the program was saved.

What It Does

Use **CHAIN** to execute another program from the program that is running.

CHAIN searches your directory for *filename*; if the file is not found, it searches the library directory (in AOS/VS and UNIX systems, it follows your search path). If Business BASIC finds the new program, it clears your currently running program from working storage, loads the new program into your working storage, and executes the new program. If it does not find the new program, your currently running program remains in working storage, and you get `Error 10 - File does not exist.`

By default, the program chained to runs from the lowest line number in the program, and all variables are cleared as if a **RUN** had occurred. If you specify **THEN GOTO** *line-number* or **THEN CON** in a **CHAIN** statement, Business BASIC acts as if you used the **CON** command, and all variables retain the values they had when the program was saved. **CHAIN** does not change the status of files. Opened files remain open and current file position pointers are maintained.

Note: During the execution of a **CHAIN** statement, keyboard interrupts are ignored. This means that you may not be able to interrupt a series of short programs executed using **SWAP** or **CHAIN** statements.

CHAIN

continued

How to Use It

CHAIN may be a program statement or a keyboard mode command. To start the new program at the beginning, use **CHAIN** without **THEN CON** or **THEN GOTO**. To resume execution of the new program from its point of interruption, use **CHAIN THEN CON**. To start execution at a certain line number, use **CHAIN THEN GOTO** *line-number*. If you want the new program to execute and then return to the original calling program, use **SWAP** instead of **CHAIN**.

Examples

1. Executes program **PROG102**.

```
00090  CHAIN "PROG102"
```

2. Executes **SCRATCH** from where it stopped before it was saved.

```
*CHAIN "SCRATCH" THEN CON
```

3. Executes **PROG3** starting at line 100.

```
00010  DIM NAME$(10)
00020  LET NAME$="PROG3"
00030  CHAIN NAME$ THEN GOTO 00100
```

CHR$

Function

Puts the binary value of a number into a string.

AOS/VS	DG/RDOS	UNIX

Format

CHR$(*expression*[,*byte*])

Arguments

expression	Any numeric expression.
byte	Optional number of bytes up to and including the maximum number of bytes allowed by your operating system (for example, triple precision). If you do not include *byte*, a one-byte string is formed.

What It Does

The **CHR$** function transfers bytes directly from a numeric expression to a string variable. Each byte of a string contains the ASCII value of the character it represents; the several bytes of a numeric variable contain a binary number that translates directly into the decimal value displayed by **PRINT**. Numeric variables hold data in four bytes in a double precision system, six bytes in triple precision, and eight bytes in quadruple precision. See Appendix B to determine the highest precision allowed by your operating system. Thus, the maximum number of bytes of *expression* that **CHR$** can place into a string variable depends on the precision you are using. If the *byte* argument is not specified, only one byte is transferred. If the number of bytes to be transferred is less than the size of the numeric expression, the low-order bytes of the numeric are used.

How to Use It

Use the **CHR$** function in a **LET** statement or command. The string variable may be a string or substring. For example, if you say **LET A$=123**, Business BASIC creates a three-byte string with digits 1, 2, and 3 as ASCII characters in the string. However, if you say **LET A$=CHR$(123)**, Business BASIC creates a one-byte string containing the character represented by the decimal ASCII value 123. See ASC for information about extracting binary values from strings.

CHR$

continued

Examples

1. The comma between arguments is a required concatenation operator. This statement makes a string concatenated to two bytes of C%, four bytes of 0, four bytes of the value R1*50 and two bytes of the value 0.

```
00080  LET D$ = CHR$(C%,2),CHR$(0,4),CHR$(R1*50,4),CHR$(0,2)
```

2. This example displays the character value of the numbers 65 to 90.

```
00010  DIM X$(1)
00020  FOR A = 65 TO 90
00030  LET X$ = CHR$(A)
00040  PRINT X$;" ";
00050  NEXT A
* RUN
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

3. Y$ is a four-byte string containing the binary value "5" in its fourth byte. Its ASCII value is printed.

```
00010 DIM Y$(4)
00020 LET Y$=CHR$(5,4)
00030 LET V=ASC(Y$)
00040 PRINT V
5
*
```

4. This example puts the record length (length of A$ plus four bytes to hold the value of the length) at the beginning of the record.

```
00100 LET RECORD$=CHR$(LEN(A$)+4,4),A$
```

CLOSE

Statement and Command

Closes all open files or a specific file.

AOS/VS	DG/RDOS	UNIX

Format

CLOSE [FILE(*channel*)]

Arguments

channel	Channel number of an open file you want to close. If channel is 16, it refers to all open files at your terminal and is equivalent to a **CLOSE** with no arguments.

What It Does

CLOSE (by itself) closes all open files, thus freeing their channel numbers. **CLOSE FILE** *channel* closes the file specified by *channel* and frees the channel for you to use again. **CLOSE FILE(16)** is equivalent to **CLOSE**.

On AOS/VS systems, the **CLOSE** instruction without an argument also causes all open INFOS II channels to be closed.

In addition, **CLOSE** frees the physical channel numbers and closes physical files that were opened using the **LOPEN** statement or command. However, it does not reinitialize LFTABL$ so that another logical file can be opened using the same logical file number.

How to Use It

Most programs close files when they finish using the files because a job is limited to a predetermined number of channels (see Appendix B for the maximum number of channels for your operating system). If you forget to close a file and you try to open it, you get an error message that says the file is already open. To solve this problem, issue a **CLOSE** or **CLOSE FILE** command while in keyboard mode and then open the file.

CLOSE

continued

Example

Consider the following program:

```
00010  REM--OPEN "DATA" ON 2
00020  OPEN FILE(2,0),"DATA"
00030  REM--PROCESS FILE
. . .
00490  REM--CLOSE "DATA"
00500  CLOSE FILE(2)
. . .
00999  REM--CLOSE ALL FILES
01000  CLOSE
```

If an interrupt, error, or **STOP** statement occurred before line 500 below and you tried to run the program again from the beginning, Business BASIC would return:

```
Error 42 - File already opened
```

since the **CLOSE** statement at line 500 had not been executed and you were trying to execute line 20 again. You could then type the command:

***CLOSE**

or

***CLOSE FILE(2)**

to resume execution of the program.

COMP

Function

Performs a one's complement of an expression.

AOS/VS	UNIX

Format

COMP(*expression*)

Arguments

expression	Numeric expression or variable to be complemented.

What It Does

The **COMP** function is used to flip the bits of a binary expression. If a bit is set to 1 in the expression, that bit is set to 0 in the result, and vice versa.

How to Use It

Use the **COMP** function to mask bits in an expression. For example, to clear a bit in an expression, perform an **AND** comparison with the one's complement of the value with that bit set.

Figure 1–5 shows how to clear the ninth bit of the value 192 by comparing it with the one's complement of 64.

```
AND (192, COMP(64))
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Power of 2
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	$expr_1$ =192
1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	$expr_2$ =COMP(64)
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	RESULT =128

Figure 1–5 COMP Function in an AND Comparison

COMP *continued*

Example

The COMP function is used to clear the ninth bit of X.

```
00010 INPUT "Initial value of X: ",X
00020 PRINT "Value of AND(X,COMP(64)): ",AND(X,COMP(64))

* RUN
Initial value of X: 192
Value of AND(X,COMP(64)): 128
```

CON

Command

Continues execution of a stopped program.

AOS/VS	DG/RDOS	UNIX

Format

CON

What It Does

CON continues the execution of a program from the point where it last stopped. **CON** does not affect either the values of variables or the status of files (i.e., open files remain open, file position pointers stay where they are, and locks are still active).

CON starts execution at the line immediately following the line at which the program was interrupted or stopped. It is equivalent to **RUN** *line-number*, where *line-number* is the number of the first line not executed after a **STOP** or interruption.

How to Use It

Use **CON** as a keyboard mode command only. If your program stops for any reason other than normal termination, you can edit it, test variables for their values, etc., then type **CON** to resume execution of the program.

Example

In this example, the program stops if an invalid age is found in the data. The user prints the count to verify the age against the original figure, then changes the age value as needed. **CON** is used to continue the program from the point at which it stopped.

```
00010 DATA 2,4,0,6,8,34,76,89,233,77,34,101,765,-1
00020 LET TOTAL=0
00030 LET COUNT=0
00040 READ AGE
00050 IF AGE<0 OR AGE>100 THEN GOTO 00200
00060 LET COUNT=COUNT+1
00070 LET TOTAL=TOTAL+AGE
00080 GOTO 00040
00100 LET AVGAGE=TOTAL/COUNT
00110 PRINT "TOTAL AGES= ";TOTAL,"NUMBER OF AGES= ";COUNT, "
AVERAGE AGE= ";AVGAGE
00120 END
00200 IF AGE=-1 THEN GOTO 00100
00210 PRINT "INVALID AGE = ";AGE
00220 STOP
00230 GOTO 00060
```

CON *continued*

```
*RUN
INVALID AGE =  233

STOP AT 00220
* PRINT COUNT 8
* AGE = 23
* CON
INVALID AGE =  101

STOP AT 00220
* PRINT COUNT 11
* CON
INVALID AGE =  765

STOP AT 00220
* PRINT COUNT 12
* AGE = 75
* CON
TOTAL AGES= 529            NUMBER OF AGES= 13            AVERAGE AGE= 40
```

CRM$

Function

Crams every three bytes of a string into two bytes.

AOS/VS	DG/RDOS	UNIX

Format

string-variable1=**CRM$**(*string-variable2*)

Arguments

string-variable1	A string variable or substring that receives the crammed string.
string-variable2	A string variable or substring you want to cram into *string-variable1*; it can be the same as *string-variable1*.

What It Does

Every three bytes of *string-variable2* are crammed into two bytes of *string-variable1*. You can cram only the following 40 characters: uppercase A to Z, 0 to 9, and four special characters. You can set the four special characters by using **STMA 10**, or you can use the default characters—space, comma (,), minus sign (–), and decimal point (.). Any character in *string-variable2* not in this set is converted to the zero value character, and when *string-variable1* is uncrammed, the zero value character becomes a space.

Cramming is done by assigning a number to each character in the string to be crammed. The table below is an example of that assignment.

```
0 = space                14 = A
1 = comma (,)            15 = B
2 = minus sign (-)       16 = C
3 = decimal point (.)    17 = D
4 = 0                    18 = E
5 = 1                    19 = F
6 = 2                    20 = G
.                        .
.                        .
.                        .
13 = 9                   39 = Z
```

This example illustrates CRM$ and UCM$:

```
00010 DIM X$(6), Y$(4)
00020 X$="ABCD"
00030 Y$=CRM$(X$)
00040 X$=UCM$(Y$)
```

CRM$ *continued*

During the execution of line 30, the following calculation takes place:

```
"A"            "B"            "C"
14 * 40^2 + 15 * 40^1 + 16 = 23016
```

The binary value 23016 is placed into the first two bytes of Y$ or Y$(1,2).

```
"D"
17 * 40^2 + 0 * 40^1 + 0 = 27200
```

The binary value 27200 is placed into the second two bytes of Y$ or Y$(3,4).

How to Use It

Use the **CRM$** function to shorten the amount of space needed to hold a string. You can use the **CRM$** function only in **LET** statements and commands because you have to assign the three bytes to two bytes of a string. You can make *string-variable1* and *string-variable2* the same string variable. You must use **UCM$** on a crammed string before you can print the string or perform other string functions on it.

Examples

1. This example shows that a string is unchanged by the cramming and uncramming process.

```
00010  DIM X$(6),Y$(9)
00020  LET Y$="ABCDEFGHI"
00030  LET X$=CRM$(Y$)
00040  LET Y$=""
00050  LET Y$=UCM$(X$)
00060  PRINT Y$
*RUN

ABCDEFGHI
```

2. This example shows the difference in length between an uncrammed string and a crammed string.

```
*DIM X$(6)
*LET X$="SIXCHR"
*PRINT LEN(X$) 6
*LET X$=CRM$(X$)
*PRINT LEN(X$) 4
```

DATA

Statement

Specifies values for variables in READ statements.

AOS/VS	DG/RDOS	UNIX

Format

DATA *value*[,*value*...]

Arguments

value	A numeric constant or a string literal in quotation marks.

What It Does

The values in all **DATA** statements in a program form a single data list. The first value in the list is the first value in the **DATA** statement with the lowest line number and the last value in the list is the last value in the **DATA** statement with the highest line number. Each item in the variable list of a **READ** statement picks up a value from the data list: the first variable in a **READ** picks up the first value in the data list, the next variable picks up the next value, etc. You can make a **READ** statement start at a particular line in the data list by using **RESTORE**.

How to Use It

Use **DATA** statements with **READ** statements to provide values for variables quickly and easily. You can have many **DATA** statements in a program, and many values of mixed types (numeric and string) in a single **DATA** statement, separated by commas. **DATA** statements can appear anywhere in a program, even after **STOP** statements, because Business BASIC does not execute **DATA** statements. You cannot use **DATA** after **THEN** in conditional statements such as **IF...THEN**, **ON ERR THEN**, or **ON IKEY THEN**.

Example

This example shows both numeric constants and string literals used as **DATA** values.

```
00005 DIM A$(20),B$(20)
00010 READ  X,Y,A$,B$
00020 PRINT X,Y,A$,B$
00050 END
00060 DATA 2,4,"HELLO","GOODBYE"
*RUN

2            4              HELLO            GOODBYE
```

DEF

Statement

Defines your own function.

AOS/VS	DG/RDOS	UNIX

Format

DEF FN*a*(*variable*)=*expression*

Arguments

a	A single letter in the range A to Z that serves as the function name with the **FN** prefix.
variable	A dummy numeric variable to be replaced with a numeric expression when you refer to the function. For UNIX only, you can specify more than one variable delimited by commas. See Appendix B for the number of variables you can specify on your operating system.
expression	A numeric expression for the calculation you want to perform.

What It Does

DEF allows you to define a function that you can refer to in a program without having to repeat the function specification. This is called a user-defined function. When you refer to a defined function by its **FN**-name, the value you supply for the variable is used in the function's calculations.

How to Use It

When you define a function, you create a name for it: use a single letter with the prefix **FN** (i.e., **FNB**). You also supply a variable in parentheses and a numeric expression that is the calculation you want your function to perform. *variable* can also be used in *expression*.

When you refer to the function by its **FN**-name, you supply a value for the variable (such as **FNA(12)** or **FNB(X+Y)**). This value replaces the variable in *expression*, and the function performs the calculations using this value. If the variable is not in the expression, the value you supply is not used, but a value must be supplied.

You can use a function's **FN**-name with a supplied value for the variable in any statement in which you can have numeric expressions. Your expression can include one or more previously defined functions. This is called nesting. See Appendix B for the number of nesting levels allowed on your operating system. All functions defined in a program exist only for the program in working storage. **DEF** limits you to single-line formulas. For longer formulas, use subroutines.

Examples

1. This example defines two functions. The second user-defined function uses the first function in its definition.

```
00010 DIM A$[1]
00020 INPUT "Enter three numbers and a character:",X ,Y,Z,A$
00030 DEF FNA(NUM) = X*Y+NUM+ASC(A$)
00040 DEF FNB(NUM) = 2*FNA(3*NUM)
00050 PRINT FNA(Z)
00060 PRINT FNB(Z)

* RUN
Enter three numbers and a character: 10,20,30,B
296
712
```

2. This example uses more than one variable.

```
00010 DEF FNA(A,B,C,D)=A+B+C+D
00020 M=2 \ N=10 \ O=3 \ P=4
00030 X=FNA(M,N,O,P)
00040 PRINT X

* RUN
19
```

DELAY

Statement and Command

Delays execution of the next program statement or command.

AOS/VS	DG/RDOS	UNIX

Format

DELAY *expression*

Arguments

expression	A numeric expression that represents the number of 10ths of a second that the program will be delayed. The maximum value allowed for expression is 32767.

What It Does

DELAY holds up the execution of the next program statement.

How to Use It

You supply *expression* to define how many 10ths of a second the program should delay. If you press the interrupt key during a delay, you stop the delay and return to keyboard mode.

For AOS/VS or UNIX, **DELAY 0** is valid and results in immediate output to the terminal. This statement is similar to **STMA 8,5**.

Examples

1. This statement causes a two-second delay.

```
00010 DELAY 20
```

2. The delay between the printing of each number increases proportionately by the value of that number.

```
00010 FOR K = 10 TO 30 STEP 5
00020 DELAY K
00030 PRINT K;
00040 NEXT K
*RUN

10 15 20 25 30
```

DELETE

Statement and Command

Deletes a file.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{DELETE } [\textit{error-code,}] \begin{Bmatrix} \textit{string-expression} \\ \textit{“filename”} \end{Bmatrix}$$

Arguments

error-code	An optional variable that receives any error code generated as a result of the execution of the **DELETE** statement. If an error occurs, the number returned in *error–code* will be a negative number that refers to a DG/RDOS error. Use the **ERM$** function to retrieve the error message associated with the error returned. It will return a value of 0 if a successful deletion occurs. This argument must be initialized (i.e., set to some value) before it is used.
string-expression	A string variable, string literal, substring, or string array element (UNIX only) that represents *filename*.
filename	A string literal in quotation marks that is the name of a file in the current directory or the directory or subdirectory specified.

What It Does

DELETE searches the current directory for *filename* (or for the file represented by the string expression) and erases *filename* and the file itself. This frees up space used by the file. The optional variable *error-code* receives the error code if any problem occurs during the deletion. The value in *error–code* is 0 if a successful deletion occurs. If a problem occurs and the error code argument is not specified, a default error trap results.

NOTE: The use of the *error-code* argument suppresses execution of the default error trap (which would cause the program to halt) or any ON ERR condition. It returns instead to *error-code* the same error code as would have been supplied by the applicable SYS error function. Therefore, you must check the *error-code* value to determine whether an error has occurred.

How to Use It

In DG/RDOS systems, you can specify a directory or subdirectory in *filename*, but you must follow the DG/RDOS naming conventions. If you use *string-expression,* it must already be dimensioned and assigned a value of *filename*. In AOS/VS and UNIX systems, use your operating system's naming conventions for filenames. In all systems, if *filename* is a link, the resolution file is deleted and the link remains.

DELETE

continued

NOTE: On UNIX systems, you must use a Business BASIC link.

The *error-code* argument is useful for setting up your own error routines when a deletion error results. This argument receives a negative number representing a DG/RDOS error code if any problem occurs with the deletion; it is 0 if the deletion is successful. If a problem occurs and *error-code* is not present, then an error trap occurs.

Examples

1. This example deletes a file.

   ```
   * DELETE "FILE101.LS"
   ```

2. Note that the error routine used in this example is a very simple one. Many types of error routines can be used once the value of ERCODE is checked.

   ```
   00010 LET ERCODE=0
   00020 DELETE ERCODE, "TEST"
   00030 IF ERCODE<>0 THEN GOTO 00060
   00040 PRINT "FILE DELETED"
   00050 GOTO 00070
   00060 PRINT "UNSUCCESSFUL DELETION, ERROR CODE VALUE:",ERCODE
   00070 END
   ```

   Below are the results of running program 2, first with and then without the existence of the file named "TEST".

   ```
   * RUN
     FILE DELETED

   * RUN
     UNSUCCESSFUL DELETION, ERROR CODE VALUE:    -10
   ```

3. This example is a modification of example 2. It uses **DELETE** without the ERCODE argument.

   ```
   00010 DELETE "TEST"
   00020 END

   * RUN

   * RUN

   I/O ERROR 10 AT 00010 - File does not exist
   ```

   The first run of this program illustrates a successful deletion. The second run causes Business BASIC to generate a default error message.

continued

DELETE

4. This error routine uses **ERM$**.

```
00010 LET ERCODE=0
00020 DELETE ERCODE,"XXX"
00030 IF ERCODE=0 THEN
00040   PRINT "FILE DELETED"
00050 ELSE
00060   DIM A$[100]
00070   LET A$=ERM$(ERCODE)
00080   PRINT "ERROR ";ERCODE;" ";A$
00090 END IF
```

```
* RUN

ERROR -10 File does not exist
```

DELREC

Statement and Command

Deletes a record from a linked-available-record file and adds it to the deleted-record chain.

AOS/VS	DG/RDOS	UNIX

Format

DELREC *logical-file-number,record-number*

Arguments

logical-file-number	A numeric expression yielding the logical file number of a data file opened using **LOPEN FILE**.
record-number	The relative record number (1 to end of file) of the record that is to be deleted.

What It Does

DELREC deletes data records in dynamically allocated files to recover dynamic space. **DELREC** flags as deleted the record specified by *record-number* in the file specified by *logical-file-number* and adds the record to the deleted-record chain. **DELREC** flags the record by changing the value of the first two bytes in the record (the status bytes) to zero.

NOTE: This statement is preferable to the **DELREC.SL** subroutine since it performs automatic locking, is faster, and frees the code space normally occupied by the **DELREC.SL** subroutine. However, the **DELREC** statement does not work with files in the PARAM file database structure. For PARAM files, you must use the **DELREC.SL** subroutine instead unless you use the physical format of LOPEN.

How to Use It

Before using **DELREC** to delete a record, you should check the status bytes to make certain the record has not been deleted. Deleting the same record twice corrupts the deleted-record chain. The **LRELINK** utility program can correct the problem, but you can avoid using it by ensuring that the record being deleted is the one you want to delete. Record 0 of a linked available record file is automatically locked during the execution of the **DELREC** statement.

Example

```
. . .
00130 DIM LFTABL$(78),T9$(544)
00140 LET LFTABL$=FILL$(0)
00150 LOPEN FILE[2,T9$],"MYINDEX"    :Open MYINDEX as an index file.
00160 LOPEN FILE[3,T9$],"MYDATA"     :Open MYDATA as database file.
00170 LET RECNO=-1
. . .
00300 INPUT "CUSTOMER # :",CNUM      :Get customer number as key.
00310 LET KEY$=CHR$(CNUM,4)          :Convert to key string.
00320 KFIND 2,T9$,KEY$,RECNO         :Find this customer.
00330 IF RECNO<=0 THEN GOTO 04000    :Didn't find him.
00340 LOCK 1,3,RECNO                 :Lock it before reading.
00350 LREAD FILE[3,RECNO],MYDATA$    :Read his record.
. . .                                :Validate record for deletion.
00360 UNPACK "J",MYDATA$,STATUS%
00370 IF STATUS%<=0 THEN GOTO 05000 :Record already deleted.
. . .
00500 KDEL 2,T9$,KEY$,RECNO          :Delete key from MYINDEX.
00510 DELREC 3,RECNO                 :Delete data record from MYDATA.
00520 UNLOCK 1                       :Unlock data record now.
00530 END
04000 PRINT "CUSTOMER NOT FOUND"
. . .
```

DIM

Statement and Command

Sets dimensions for arrays and/or strings.

AOS/VS	DG/RDOS	UNIX

Format

AOS/VS and DG/RDOS Systems:

$$\mathrm{DIM}\left\{\begin{array}{l}\textit{string-variable}(n)\\ \textit{numeric-array}(\textit{column})\\ \textit{numeric-array}(\textit{row},\textit{column})\end{array}\right\}\ldots$$

UNIX Systems:

$$\mathrm{DIM}\left\{\begin{array}{l}\textit{string-variable}(n)\\ \textit{numeric-array}(\textit{column})\\ \textit{numeric-array}(\textit{row},\textit{column})\\ \textit{numeric-array}(\textit{dimen0},\ldots,\textit{dimen7})\\ \textit{string-array}(n;\textit{dimen0},\ldots,\textit{dimen7})\end{array}\right\}\ldots$$

Arguments

string-variable	A string variable name ending with a $ symbol.
numeric-array	Variable name for an array.
string-array	A string array name ending with a $ symbol.
n	The maximum string length of *string-variable* or of an element of *string-array*. It can be a variable or a numeric expression.
column	A numeric expression or a variable that specifies the number of elements in a one-dimensional array or the array position of the last element in a two-dimensional array. Start with 0 in determining the count to specify in *column*.
row	A numeric expression or variable that specifies the array position of the last row in a two-dimensional array. Start with 0 in determining the count to specify in *row*.
dimen	The number of elements in a *numeric-array* or *string-array* dimension, beginning with dimension 0. *dimen* can be a numeric expression or a variable. You can specify up to eight dimensions. (UNIX only)

What It Does

Business BASIC uses the information you supply in a **DIM** statement or command to allocate storage. The value of *column* defines how many column positions a one-dimensional array needs with one element per column position. If you specify *row* and *column*, these values determine how many row and column positions a two-dimensional array can contain. All numeric arrays start at column position 0. For example, a one-dimensional array with 16 elements starts at column position 0 and ends at column position 15. This array is dimensioned using "15" as the value of *column*.

For string arrays on UNIX systems, use *dimen* to specify the number of elements in each dimension. Code one *dimen* parameter for each dimension. Each string array dimension starts at position 1.

How to Use It

If you use a numeric array but do not declare it in a **DIM** statement, Business BASIC sets aside 11 elements (column positions 0 to 10) for a one-dimensional array and 121 elements (columns 0 to 10 and rows 0 to 10) for a two-dimensional array. An undeclared one-dimensional array cannot have more than 11 elements, and an undeclared two-dimensional array cannot have more than 121 elements. If you need more space, dimension the array with DIM. The number of elements you can place in an array is restricted only by the amount of available memory.

You must dimension string variables and string arrays; no default exists. The value of *n* defines how many characters you will have in a string. For string variables and string arrays, *n* begins with 1, not 0, so A$(10) has 10 characters.

You can dimension a numeric array and a string variable or several numeric arrays and string variables in the same **DIM** statement in any order. On UNIX systems, you can combine string arrays with string variables and numeric arrays. You can also use separate **DIM** statements. Use parentheses or square brackets to enclose your subscripts. Express the values for *row* and *column* as variables or numeric expressions; if you use variables, they must have pre-assigned values. Your variable for an array must follow the naming rules for Business BASIC variables.

Your *string-variable* must follow the rules governing string variables. The value you supply for *n* (string length) can be a variable or a numeric expression. Again, use either parentheses or square brackets to enclose your subscripts.

See Appendix B for the maximum string length, variable name length, and number of program variables allowed on your operating system.

To refer to any element of an array or any character of a string, use the name of the array or the string variable and supply the identifying subscript.

On UNIX systems, to refer to the entire string in a string array element, do not specify a length (*n*), but do include a semicolon. For example, the command **X$=A$(;1,1)** refers to the entire string at element 1,1. You can refer to a substring of a particular element's string by specifying the range of bytes before the semicolon. For example, the command **X$=A$(4,9;3,4)** places bytes 4-9 of the string at element 3,4 into X$.

DIM

continued

Use **DIM** as either a program statement or a keyboard mode command. If you use subscripted variables in a **DIM** statement or command, you must supply values for the variables before executing **DIM**. If used as a command, **DIM** applies only to a program already executing (i.e., you must use **DIM** before using **CON**), because a **RUN** resets all program variables.

You can redimension a previously dimensioned array or string variable during execution of a program, but you cannot alter the size of the storage space already allocated for it. Numeric arrays, string arrays, and string variables can be redimensioned only to the same number of elements or fewer; if fewer, you cannot use or refer to locations in the original array that are not used in the redimensioned array. Use redimensioning primarily to change the subscripts of arrays with two or more dimensions.

Examples

1. SALES is a 6x7 element two-dimensional array. C is a 21-element one-dimensional array. STRING$ is a 30-character string variable.

```
00010 DIM SALES(5,6),C(20),STRING$(30)
```

2. This statement redimensions SALES to a 7x6 two-dimensional array, redimensions one-dimensional array C of 21 elements to be a 3x7 two-dimensional array, and changes the length of STRING$ to 28 characters.

```
00020 DIM SALES(6,5),C(2,6),STRING$(28)
```

3. This example stops a program and dimensions arrays using variables with pre-assigned values.

```
STOP AT 00070
* DIM SALES(X,Y),C(J),STRING$(Z)
* CON
```

4. This statement dimensions a 1x2 two-dimensional string array with 80-byte elements.

```
* 10 DIM SA$(80;0,1)
```

5. This UNIX example uses string arrays to determine which offices are occupied in two buildings. Each building has three halls with 20 offices on each hall.

```
00010 DIM LOCAT$[25;1,2,19],NAME$[25]
00020 INPUT "Enter Employee Name or 'END': ",NAME$
00030 IF NAME$[1,3]="END" THEN GOTO 00090
00040 INPUT "Enter Building #: ",B,"   Hall #: ",H," Office #: ",O
00050 PRINT
00060 LET B=B-1 \ H=H-1 \ O=O-1
00070 LET LOCAT$[;B,H,O]=NAME$
00080 GOTO 00020
00090 REM Display only the offices that are occupied
00100 FOR B=0 TO 1
00110   PRINT
00120   PRINT "Building ";B+1;
```

```
00130   FOR H=0 TO 2
00140     PRINT TAB(5),"Hall ";H+1
00150     FOR O=0 TO 19
00160       IF LOCAT$[1,1;B,H,O]="" THEN GOTO 00180
00170       PRINT TAB(23);"Office ";O+1;LOCAT$[1,25;B,H,O]
00180     NEXT O
00190   NEXT H
00200 NEXT B
```

```
* RUN
Enter Employee Name or 'END': Barry
Enter Building #: 1   Hall #: 1   Office #: 1

Enter Employee Name or 'END': Wendy
Enter Building #: 1   Hall #: 1   Office #: 2

Enter Employee Name or 'END': Keith
Enter Building #: 1   Hall #: 1   Office #: 3

Enter Employee Name or 'END': Steve
Enter Building #: 1   Hall #: 2   Office #: 1

Enter Employee Name or 'END': Diana
Enter Building #: 1   Hall #: 2   Office #: 2

Enter Employee Name or 'END': Priscilla
Enter Building #: 1   Hall #: 3   Office #: 1

Enter Employee Name or 'END': Caroline
Enter Building #: 1   Hall #: 3   Office #: 2

Enter Employee Name or 'END': Debbie
Enter Building #: 2   Hall #: 1   Office #: 1

Enter Employee Name or 'END': END

Building  1   Hall  1
                        Office 1 Barry
                        Office 2 Wendy
                        Office 3 Keith
              Hall 2
                        Office 1 Steve
                        Office 2 Diana
              Hall 3
                        Office 1 Priscilla
                        Office 2 Caroline

Building  2   Hall  1
                        Office 1 Debbie
              Hall 2
              Hall 3

*
```

DIR

Statement and Command

Displays the current directory or moves to another directory.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{DIR} \left[\left\{ \begin{matrix} \textit{"pathname"} \\ \textit{string-variable} \end{matrix} \right\} \right]$$

Arguments

pathname	The name of a directory or a path to a directory expressed as a string literal in quotation marks. For rules about AOS/VS pathnames, see the *AOS/VS CLI User's Manual.* For rules about UNIX pathnames, see your programmer's reference manual. In DG/RDOS, this is a directory specifier. See *Using the DG/RDOS Command Line Interpreter* for more information.
string-variable	Variable to represent pathname or to receive a value of pathname.

What It Does

DIR without an argument displays the current directory's name. **DIR** with a *pathname* argument (or a string variable representing a pathname) changes the current directory to the directory specified in *pathname*. However, if the string variable is a null string, it receives a value of the pathname of the current directory. Only "AA" accounts can use a **DIR** statement or command, unless your Business BASIC system has been generated to allow all users to use privileged statements and commands. In DG/RDOS, **DIR** changes the system directory as well as the user's current directory.

How to Use It

Use **DIR** as a keyboard mode command or a program statement. To specify a directory that is within the current directory (i.e., a subdirectory), specify the directory's name. To specify a directory that is not within the current directory, specify a pathname. To find out the current directory in a **DIR** program statement, use a null string variable as an argument (as shown in the example below).

Example

Line 20 of this example gets the current directory and assigns it to A$. Line 30 displays the current directory name. Line 40 moves to directory **DEPT43**. Line 50 swaps to and executes the program **ACCT.PY**. Line 60 moves to the previous directory (its name is still in A$). A subdirectory is necessary in AOS/VS and UNIX systems.

```
00010  DIM A$(20)
00020  DIR A$
00030  PRINT A$
00040  DIR "DEPT43"
00050  SWAP "ACCT.PY"
00060  DIR A$
```

DO WHILE/UNTIL/END LOOP

Statement

Defines a program loop.

AOS/VS	UNIX

Formats

1. Condition at the top or condition not used.

```
DO [WHILE expression]
 .
 .    (Block of executable statements)
 .
END LOOP
```

2. Condition at the bottom.

```
DO
 .
 .    (Block of executable statements)
 .
{ WHILE }
{ UNTIL } expression
```

Arguments

expression	A Boolean expression that is evaluated each time the loop is executed. If *expression* is true, the loop is executed; if it is false, the loop is exited.

What It Does

If you code a **WHILE** or **UNTIL** condition, Business BASIC performs the block of executable statements either while *expression* is true or until *expression* is true. When you use **WHILE** or **UNTIL** at the bottom of the loop, the **END LOOP** statement is not required.

When **WHILE** *expression* appears at the top of the loop, a matching **END LOOP** statement is required. If no matching **END LOOP** statement is found, `Error 96 – DO with no matching END LOOP, WHILE or UNTIL` is raised. You cannot code an **UNTIL** condition at the top of a **DO** loop.

If *expression* appears at the top of the loop, it is evaluated before any of the loop body is executed. If *expression* is false when it is first evaluated, the loop is not executed. As long as *expression* is true, the loop is executed. When *expression* becomes false, control is transferred to the statement following the matching **END LOOP** statement, and the loop becomes inactive.

continued

DO WHILE/UNTIL/END LOOP

If **WHILE** *expression* or **UNTIL** *expression* appears after the body of executable statements, the loop is executed at least one time before *expression* is evaluated. If Business BASIC encounters a **WHILE** statement at the top of the loop and a **WHILE** or **UNTIL** statement at the bottom of the same loop, it returns `Error 98 – DO with conditional at top and bottom.`

An **END LOOP, WHILE** or **UNTIL** statement without a corresponding **DO** statement causes `Error 97 – END LOOP, WHILE or UNTIL with no matching DO.`

How to Use It

You can nest up to 32 levels of **DO** loops.

You can enter and exit **DO** loops without affecting loop execution. When you exit a loop (using a **GOTO** or **GOSUB** statement), the loop remains active. To terminate the loop before exiting, use the **BREAK** statement or **STMA 8,6**.

If a Business BASIC program ends while one or more **DO** loops are still active (that is, their **END LOOP** statements have been omitted), no error is generated but the loops remain active.

You can code a **DO** loop without a condition at the top or bottom of the loop. This creates an endless loop unless you include code that allows you to break out of the loop.

Examples

1. This program is an endless **DO** loop.

```
00010 DIM NAME$[30],NAME2$[30]
00020 OPEN FILE[0,1],"VISITOR_LOG"
00030 DO
00040   PRINT @(-30)
00050   PRINT @(3,25);" V I S I T O R ´ S   L O G"
00060   INPUT @(13,14),"ENTER YOUR NAME:  ",NAME$
00070   IF NAME$="END OF DAY" THEN BREAK
00080   INPUT @(15,10),"PERSON TO MEET WITH: ",NAME2$
00090   PRINT FILE[0],"VISITOR NAME: ";NAME$
00100   PRINT FILE[0],"MET WITH: ";NAME2$
00110   PRINT FILE[0],"DATE: ";SYS(2);"/";SYS(1);"/";SYS(3)
00120   PRINT FILE[0],"TIME: ";SYS(14);":";SYS(13)
00130   PRINT FILE[0]
00140 END LOOP
00150 CLOSE FILE[0]
00160 END
```

DO WHILE/UNTIL/END LOOP

2. This **DO WHILE** loop has the condition at the bottom of the loop. The loop is executed once before the condition is checked.

```
00010 LET X=5
00020 DO
00030  PRINT "This line prints once before the condition is
checked."
00040 WHILE (X < 5)

* RUN
This line prints once before the condition is checked.
```

3. In this example, the **WHILE** condition is at the top of the loop. Because the **WHILE** condition is evaluated before the loop is executed, an error occurs when this program is run.

```
* 10 DATA 1,2,3,0,4
* 15 X=-1
* 20 DO WHILE X<>0
* 30 READ X
* 40 Y = 10/X
* 50 PRINT Y
* 60 END LOOP
* RUN
10
5
3
Error 16 at 40 - Arithmetic
```

4. This example uses an **UNTIL** condition at the bottom of the loop.

```
* 10 LET X=1
* 20 LET POWERS_OF_TWO=0
* 30 DO
* 35   LET POWERS_OF_TWO=POWERS_OF_TWO+1
* 40   LET X=SHFT(X,1)
* 70 UNTIL X=512
* 80 PRINT "512 = 2 raised to the";POWERS_OF_TWO;"th power"
* run
512 = 2 raised to the 9th power
```

END

Statement

Terminates program execution.

AOS/VS	DG/RDOS	UNIX

Format

END

What It Does

END terminates program execution. Without an **END** or **STOP** statement, the program terminates after its highest numbered statement.

END does not cause a message to be written to the terminal, but a **STOP** statement does.

How to Use It

Use **END** only as a program statement. It does not have to be the last statement in your program, but it does terminate the program when it is executed.

END LOOP

Statement

Terminates DO loop execution.

AOS/VS	UNIX

Format

END LOOP

What It Does

END LOOP terminates execution of the last active **DO** loop.

How to Use It

Use **END LOOP** only as a program statement. It should follow the **DO** loop that it terminates. For more information, see **DO WHILE/UNTIL/END LOOP**.

ENTER

Statement and Command

Merges program source statements with working storage.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{ENTER}\left\{\begin{array}{l}\textit{"filename"}\\ \textit{string-variable}\end{array}\right\}$$

Arguments

filename	The name of a disk file or device, expressed as a string literal in quotation marks, containing BASIC source statements in character (ASCII) format.
string-variable	A string variable already dimensioned and assigned a filename value.

What It Does

ENTER is the command to merge programs. Any program that you list to a file is in ASCII (readable) format and can be entered. **ENTER** searches the current directory for *filename*; if it does not find it, it searches the library directory (in AOS/VS and UNIX systems, it follows your search path). If it does not find *filename* there, it gives you an error message. If it does find *filename*, **ENTER** brings the new program statements into working storage. When a line number of the entered program matches a line number of the program currently in working storage, the entered statement replaces the current program statement. For line numbers that do not have a match, Business BASIC inserts the entered statement into the proper sequence of current program statements. If you do not want to merge the entered file with working storage, then you must execute a **NEW** statement before the **ENTER** statement.

ENTER does not restore **DATA** statements; to restore **DATA** statements, use **RESTORE**. In a partially executed program that you have listed, variable assignments and file status do not remain fixed when you enter the program.

How to Use It

Use **ENTER** either as a keyboard mode command or a program statement. Your filename is a source file created by listing a program to a file or by using an editor. In addition, the source file can be a file, in character format, that was created outside the Business BASIC system.

Because a subroutine is a source file (with an .SL extension) and not a program file, you must enter a Business BASIC subroutine into your program to use it. See **LIST** for more information about character format programs, listing files, and source files.

ENTER

continued

Example

In this example, two programs are merged. Note that line 10 is overwritten by the second program.

```
* NEW
10 PRINT "THIS IS LINE 10."
20 PRINT "THIS IS TEST1."
30 PRINT
* LIST "TEST1"

* NEW
10 PRINT "THIS IS THE NEW LINE 10."
100 PRINT "THIS IS TEST2."
* LIST "TEST2"

* NEW
* ENTER "TEST1"
* ENTER "TEST2"
* LIST
00010 PRINT "THIS IS THE NEW LINE 10."
00020 PRINT "THIS IS TEST1."
00030 PRINT
00100 PRINT "THIS IS TEST2."
```

EOF

Function

Checks for end of file.

AOS/VS	DG/RDOS	UNIX

Format

EOF(*channel*)

Arguments

channel	A numeric expression for the channel number of a file opened for reading (input or input/output).

What It Does

The **EOF** function returns 1 if it detects an end of file; otherwise it returns 0. The **EOF** function cannot detect the end of a subfile, only the end of a physical file. **POSITION FILE** resets the end-of-file flag.

NOTE: If you use **EOF** for an unused channel (i.e., one without an open file), -1 is returned in **EOF**.

How to Use It

EOF is used frequently in **IF...THEN** statements to make a conditional transfer at the end of a file. An **IF** EOF(*channel*) **THEN** *statement* executes *statement* when the **EOF** returns 1. Put your **EOF** statement after your **READ FILE** or **INPUT FILE** statement. You must keep track of a subfile's length to detect its end, since **EOF** does not work with subfiles. Use the status word described in **MTDIO** to detect the end of a tape file.

The **EOF** flag is set on the first attempt to read or input beyond the end of the file.

Example

```
00010  OPEN FILE(1,3),"INPUTFILE"
00020  READ FILE(1),A,B,C,D,E
00030  IF EOF(1) THEN GOTO 00200
00040  PRINT A,B,C,D
00050  GOTO 00020
00200  PRINT "END OF FILE"
00210  CLOSE FILE(1)
```

ERASE

Statement and Command

Erases program statements.

AOS/VS	DG/RDOS	UNIX

Format

ERASE *linenumber1*, [*linenumber2*]

Arguments

linenumber	A valid line number in your program or a valid range of lines (*linenumber* doesn't have to be an exact line number in the program).

What It Does

ERASE removes *linenumber1* through *linenumber2* inclusively from your program. If no lines exist in your program in the range *linenumber1* to *linenumber2*, you get an error message. If *linenumber1* and/or *linenumber2* do not exist but there are lines with numbers in between them, Business BASIC erases those lines. If a *linenumber1* is followed by a comma, all lines from *linenumber1* to the end of the program are erased.

How to Use It

Use **ERASE** either as a program statement or a keyboard command. As a command you do not have to type the keyword **ERASE**. When using **ERASE** as a statement, give the line number twice to erase a single line.

Examples

1. This erases lines 1500 through 1900.

   ```
   00010  ERASE 1500,1900
   ```

2. This command erases line 100 of the program currently in working storage.

   ```
   * ERASE 100,100
   ```

3. This command erases from line 600 to the end of the program currently in working storage.

   ```
   * ERASE 600,
   ```

ERM$

Function

Retrieves an error message.

AOS/VS	DG/RDOS	UNIX

Format

LET *string-variable*=ERM$(*number*)

Arguments

string-variable	A string variable, substring, or string array element (UNIX only) used to receive the error message; it must be dimensioned large enough to contain the error message.
number	The record number in the **BASIC.ER** file for the error message you want.

What It Does

ERM$ retrieves the error message specified by *number*. The number typically comes from **SYS(7)**, **SYS(40)**, or **SYS(41)**.

NOTE: **SYS(40)** and **SYS(41)** are available only on AOS/VS and UNIX systems.

How to Use It

When you trap errors in your program (by using **ON ERR**), you can have the program print the appropriate error message. Also, you can generate your own unique error messages by adding messages to the end of the **BASIC.ER** file and then using *number* to specify which message you want to retrieve.

You can use **ERM$** only in **LET** statements or commands because you have to assign the error message to *string-variable*. The largest error message is 64 bytes long.

See **SYS**, **AERM$**, and **UERM$** for more information about using error functions.

ERM$

continued

Examples

1. When Business BASIC encounters an error in this program, control passes to line 500, where the appropriate error code is selected and the error message associated with that error code is printed.

```
00010  ON ERR THEN GOTO 00500
00020  DIM ER$(64)
 . . .
00500  REM ERROR ROUTINE
00510  IF SYS(7)=-60 THEN
00520    LET ER = SYS(31)
00530    LET ER$ = AERM$(ER)
00540  ELSE
00550    LET ER = SYS(7)
00560    LET ER$ = ERM$(ER)
00570  END IF
00580  PRINT "ERROR # ";ER;"= ";ER$
00590  END
 . . .
```

2. This example uses **ERM$**, **AERM$**, and **UERM$** to retrieve error messages from **SYS(41)**, **SYS(42)**, and **SYS(43)**.

```
01000 REM * error handler
01010 IF SYS(41)=-60 THEN              :Same as SYS(7) and SYS(40)
01012   IF SYS(42)=-276 THEN
01014     LET ER$=UERM$(SYS(43))
01016   ELSE
01020     LET ER$=AERM$(SYS(42))       :Same as SYS(31)
01025   END IF
01030 ELSE
01040   LET ER$=ERM$(SYS(41))          :Same as SYS(7) and SYS(40)
01050 END IF
```

EXTRACT

Statement

Extracts the next field from a string.

AOS/VS	DG/RDOS	UNIX

Format

EXTRACT *field,input-string,delimiters,position*

Arguments

field	A string variable receiving the field extracted from *input-string*.
input-string	A string expression containing the field to be extracted.
delimiters	A string expression containing characters or nulls to be recognized as delimiters for the fields in *input-string*.
position	A numeric variable pointing to the last delimiter processed in *input-string*. It must be initialized to 0 before the initial EXTRACT statement. When EXTRACT reaches the end of *input-string* or when EXTRACT ends without finding a match, *position* is set to -1.

What It Does

EXTRACT scans *input-string* for each character listed in *delimiters*. Scanning starts with the relative location indicated by *position* plus 1 (skipping leading blanks). The substring beginning with the first nonblank character and continuing through the last nonblank character that is not the set indicated by *delimiters* is placed in *field*. The relative location of the terminating character (a blank or a character in delimiters) is placed in *position*. When EXTRACT reaches the end of the *input-string* or when EXTRACT ends without finding a match, *position* is set to -1.

How to Use It

Enter EXTRACT as a program statement. Make sure the position argument is initialized to 0 before EXTRACT executes.

Example

This program prompts the user to enter an input string and the field delimiters. It then uses EXTRACT to remove the fields from the input string until EXTRACT reaches the end of string. Each time it executes EXTRACT, the program displays the position of the delimiter that marked the end of the substring and the substring that was moved into the *field* argument (C$). When the end of the string is reached in the first pass through the program, the *position* argument is set to -1; the substring is Z. The program then returns to line 00020 and prompts to user to enter another input string and delimiters. This continues until the interrupt key is pressed.

EXTRACT

```
00010 DIM A$[100],B$[100],C$[20]
00020 INPUT A$," DELIMITERS: ",B$
00030 LET P=0
00040 EXTRACT  C$,A$,B$,P
00050 PRINT "P= ";P," ARG=";C$
00060 IF P>0 THEN GOTO 00040
00080 GOTO 00020
00090 REM USE ESCAPE OR OTHER INTERRUPT KEY TO EXIT
```

```
? ABCDE1FGHIJ2KL3MNOP4XY6Z   DELIMITERS: 124567
P=  6   ARG=ABCDE
P=  12  ARG=FGHIJ
P=  20  ARG=KL3MNOP
P=  23  ARG=XY
P= -1   ARG=Z
```

This time the user enters an input string but no delimiters. Since there are no blanks in the input string and no other delimiters were entered, the substring that is moved to the *field* argument (C$) is identical to the input string, and the *position* argument (**P**) is set to -1. The following examples show the results you get based on other input strings and delimiter combinations.

```
?  123456567789  DELIMITERS:
P= -1   ARG=123456567789

?  123 4567 890  DELIMITERS:
P=  4   ARG=123
P=  9   ARG=4567
P= -1   ARG=890

?  12.45 -64.34  DELIMITERS:.
P=  3   ARG=12
P=  6   ARG=45
P=  10  ARG=-64
P= -1   ARG=34

?RUN  DELIMITERS:
P= -1   ARG=RUN


? 12/31/84  DELIMITERS: / -
P=  3   ARG=12
P=  6   ARG=31
P= -1   ARG=84

? 31/12/84  DELIMITERS: / -
P=  3   ARG=31
P=  6   ARG=12
P= -1   ARG=84
?
IKEY AT 00020
```

FILL$

Function

Fills a string or substring with a value.

AOS/VS	DG/RDOS	UNIX

Format

string-variable=**FILL$**(*ascii*[,*bytes*])

Arguments

string-variable	The string variable or substring you want filled.
ascii	The numeric expression for the decimal ASCII value of the character with which you want to fill the string.
bytes	An optional numeric expression for the number of bytes in *string-variable* that you want to fill.

What It Does

FILL$ without *bytes* fills *string-variable* from the current position in the string variable (or the beginning byte of the substring represented by the string variable) to the dimensioned length of the string variable (or the ending byte of the substring represented by the string variable). The string variable is filled with the ASCII character whose decimal code is represented in the field labeled *ascii.*

FILL$ with *bytes* fills the string variable from the current position in the string variable (or the beginning byte of the substring) to the number of bytes you specify. For example, if the current position of the string variable is 30 and you specify 10 bytes, **FILL$** fills the string with *ascii* from position 30 to position 39 inclusive unless the dimensioned length of the string variable is less than 39, in which case it only fills the string to its dimensioned length.

How to Use It

Use **FILL$** only with **LET** statements and commands. You can concatenate several string functions and string expressions in one assignment statement. Your *string-variable* can be a regular string variable, a string variable with a subscript that refers to a byte position within its dimensioned length, or a substring whose subscripts refer to the beginning and ending bytes of a substring.

FILL$

continued

Examples

1. The string A$ is filled. Notice the difference in length before and after the FILL$.

```
00005  REM--FILL A$ TO END WITH BLANKS
00010  DlM A$(512)
00020  LET A$="ABCDE"
00030  PRINT LEN(A$)
00040  LET A$=A$,FILL$(32)
00050  PRINT LEN(A$)
* RUN
5
512
```

2. This example uses substring designations to indicate the positions to be filled.

```
* DIM B$(512)
* LET B$=A$(1,5)
* PRINT LEN(B$)
5
* LET B$(6,15)=FILL$(32,4)
* PRINT LEN(B$)
9
* LET B$(9,15)=FILL$(32)
* PRINT LEN(B$)
15
* LET B$(0)=FILL$(32,5)
* PRINT LEN(B$)
20
* LET B$(0)=FILL$(32)
* PRINT LEN (B$)
512
```

NOTE: In Business BASIC, B$(0) refers to the position following the current length (returned by LEN) up to the dimensioned length of B$.

FOR...NEXT

Statement

Defines a program loop.

AOS/VS	DG/RDOS	UNIX

Format

FOR *control-variable=expression1* **TO** *expression2* [**STEP** *expression3*]

.
. (Block of executable statements)
.

NEXT *control-variable*

Arguments

control-variable	A simple, nonsubscripted numeric variable that controls the loop by being increased or decreased. You must use the same variable in **FOR** and **NEXT** statement pairs.
expression1	The first value assigned to the *control-variable* when a loop is encountered. It can be a numeric expression or a variable.
expression2	The last value assigned to *control-variable* and the terminating value of the loop. It can be a numeric expression or a variable.
expression3	The amount by which you want to increase or decrease *control-variable* after each loop execution. A positive (+) value increases *control-variable*, while a negative (–) value decreases it. If you do not specify *expression3*, the default value is +1.

What It Does

FOR...NEXT statements define a loop. The **FOR** statement defines how many loops to perform and the **NEXT** statement marks the end of the loop. A **FOR...NEXT** loop can execute Business BASIC statements any number of times (including zero times). The loop can assign increasing or decreasing values to variables and perform a variety of tasks.

When Business BASIC first encounters a **FOR** statement, it evaluates *expression1*, *expression2*, and *expression3*. If you omit **STEP** *expression3*, the assumed step value is +1. The *control-variable* is set to the value of *expression1*, and loop operation proceeds as follows:

1. The *control-variable* is tested: if *expression3* is positive (increasing) and *control-variable* is greater than *expression2*, the loop immediately terminates. Control then passes to the statement following the corresponding **NEXT** statement. *control-variable* retains the last value it had during loop execution.

   If *expression3* is negative (decreasing) and *control-variable* is less than *expression2*, then the loop immediately terminates. Control then passes to the statement following the corresponding **NEXT** statement. *control-variable* retains the last value it had during loop execution.

FOR...NEXT

continued

If *control-variable* passes the loop execution test, the following steps are performed:

2. Business BASIC executes the block of executable statements in the **FOR...NEXT** loop.
3. When Business BASIC reaches the corresponding **NEXT** statement *control-variable* is set to the value of *control-variable* + *expression3*. Business BASIC then returns to step 1 above and repeats the loop.

How to Use It

You cannot use **FOR...NEXT** statements as keyboard mode commands.

You can execute **FOR...NEXT** loops repeatedly by nesting one loop within another. Every **FOR** statement must have a matching **NEXT** statement and vice versa, or you get an error message. See Appendix B for the maximum nesting depth allowed on your operating system.

To properly nest a loop within another loop, make sure that the **FOR** and **NEXT** boundaries of the outer loop contain the **FOR** and **NEXT** boundaries of the inner loop. For example:

Legal Nesting

```
FOR X =
  FOR Y =
    FOR Z =
    NEXT Z
  NEXT Y
NEXT X
```

Illegal Nesting

```
FOR X =
  FOR Y =
NEXT X
  NEXT Y
```

If you use numeric variables for your expressions, you must assign values to them before using them. Give *expression1*, *expression2*, and *expression3* positive or negative values, but note that *expression3* cannot be 0. If you want to loop one time, just set *expression2* equal to *expression1*.

The values of *expression2* and *expression3* are calculated before the loop begins, so if you use any variables in these expressions and change their values in an instruction within the loop, termination of the loop is not affected in any way. To alter the loop's termination, change the value of the *control-variable* with an instruction within the loop.

If Business BASIC exits from a loop before completing the loop (using **GOTO** or **GOSUB**), the **FOR** statement of that loop remains active until Business BASIC executes another **FOR** statement with the same *control-variable*. You can return to the loop if no other loop in the program has the same *control-variable* or if another loop with the same *control-variable* has not executed yet. When another loop with the same control variable is executed, the first loop is discarded. In this case, the **NEXT** statement for the discarded **FOR** statement causes `Error 22 – NEXT – no FOR` to be generated.

Branching in and out of a **FOR...NEXT** loop is possible, but if you skip a **NEXT** statement and return to the loop without restarting it, you get unpredictable results. If you branch out of a loop and then want to return and resume execution of the loop, you should return to the **NEXT** statement and execute it. If you do not, and you try to execute a **FOR** statement, it might be interpreted as a new nested loop within your unfinished loop and could cause problems.

Examples

1. The control variable J equals last value assigned during execution of loop, before *expression2* was exceeded.

```
* 10  FOR J = 1 TO 10 STEP 2
* 20    PRINT J,
* 30  NEXT J
* 40  PRINT
* 50  PRINT"J NOW HAS THE VALUE:";J
* RUN
1 3 5 7 9
J NOW HAS THE VALUE:9
```

2. The loop counter begins at 10 and is decremented by 2 until the counter reaches 2.

```
* 10  FOR J=10 TO 2 STEP-2
* 20    PRINT J,
* 30  NEXT J
* 40  PRINT
* 50  PRINT "LAST VALUE OF J: ";J
* RUN
10 8 6 4 2
LAST VALUE OF J: 2
```

FOR...NEXT

continued

3. The loop counter begins at 12 and is incremented by 1 until 18 is reached.

```
* LIST
00010 LET X=12
00020 LET Y=18
00030 FOR NUMBER=X TO Y
00040   PRINT NUMBER,
00050 NEXT NUMBER
00060 STOP
* RUN
12 13 14 15 16 17 18
STOP AT 00060
```

GETREC

Statement and Command

Gets an available record from a data file opened using LOPEN FILE.

AOS/VS	DG/RDOS	UNIX

Format

GETREC *logical-file-number,record-number*

Arguments

logical-file-number	A numeric expression that represents the logical file number of a data file opened using **LOPEN FILE**.
record-number	The relative (1 to end of file) record number returned as available by **GETREC**. This argument must be initialized.

What It Does

GETREC retrieves the next available record number in a type L (linked-available-record) logical data file. **GETREC** first examines record 0 of the data file to determine if any records have been deleted. If records have been deleted, the next available record number is that of the last record in the deleted-record chain. If there are no deleted records, **GETREC** returns the next available record number. Once the record number has been determined, **GETREC** updates record 0 to reflect the new status of the file. If a record of -1 is returned, no available records exist, and the file is full.

NOTE: This statement is preferable to the **GETREC.SL** subroutine since it performs automatic locking, is faster, and frees the code space normally occupied by the **GETREC.SL** subroutine. However, the **GETREC** statement does not work with files in the PARAM file database structure. For PARAM files, you must use the **GETREC.SL** subroutine instead unless you use the physical format of LOPEN.

How to Use It

Enter **GETREC** as a program statement. **GETREC** can only be used with files in the logical database file structure that have been opened with the **LOPEN FILE** statement. After the *record-number* has been retrieved, you should make sure that the file is not full. If a valid *record-number* was returned, the record can be written to the file using the **LWRITE FILE** statement.

GETREC *continued*

Examples

1. Allocate a record.

```
00200 LOPEN FILE[3,T9$],"MYDATA"      :Open MYDATA as database
                                      :file
. . .
00460 RECNO = 0                       :Initialize record number
00470 GETREC 3,RECNO                  :Allocate a record in MYDATA
00475 IF RECNO<0 THEN GOTO 01000      :No available records, so
                                      :go to end program routine.
00480 LWRITE FILE[3,RECNO],MYDATA$    :Add new record to file
```

2. Compose a keyed record.

```
00010 DIM RECNO(0)
00020 DIM LFTABL$(78),CNAM$(20),BUF$(544),KEYID$(6),RECORD$(24)
00030 LET LFTABL$=FILL$(0)           :Initialize localfile table.
00040 LOPEN FILE (2,BUF$),"CUDATA"   :Open customer data file.
00050 LOPEN FILE (3,BUF$),"INDEX"    :Open index file.
00060 INPUT "CUSTOMER ID NUMBER: ",CNUM
00070 LET KEYID$=CHR$(CNUM,4)              :Compose key string.
00090 REM ADD RECORD
00100 INPUT USING "","COMPANY NAME:     ",CNAM$
00110 PACK "ZJJA20",RECORD$,1,CNUM,CNAM$   :Compose RECORD$ from
                                           :CNUM and CNAM$.
00120 GETREC 2,RECNO           :Allocate record in CUDATA file.
00125 IF RECNO<0 THEN GOTO 00200     :No available records, so
                                     :go to end of program.
00130 LWRITE FILE (2,RECNO),RECORD$  :Write it out.
00140 LET KEYID$=CHR$(CNUM,3)        :Add customer number in
                                     :CNUM to index.
00150 KADD 3,BUF$,KEYID$,RECNO
00190 CLOSE
00200 END
```

GOSUB...RETURN

Statement

Transfers control to and from a subroutine within a program.

AOS/VS	DG/RDOS	UNIX

Format

```
GOSUB line-number1
  .
  .
line-number1
  .
  . (subroutine statements)
  .
RETURN
```

Arguments

line-number1	The line number entry point of a subroutine; it must exist as a program line number.

What It Does

GOSUB immediately transfers control to the line number specified by *line-number1*. Business BASIC executes the statements of the subroutine one by one, beginning at *line-number1* until Business BASIC reaches the **RETURN** statement. The **RETURN** statement returns control to the statement immediately following the **GOSUB** statement that provided an entry point for the subroutine.

How to Use It

Enter a subroutine only by using **GOSUB**. Without **GOSUB**, the **RETURN** statement causes an error when executed.

Many **GOSUB** statements can branch to the same subroutine. A **RETURN** occurring in that subroutine returns control to the statement immediately following the specific **GOSUB** that last branched to the subroutine.

You can also use more than one **RETURN** statement in a subroutine if program logic requires the subroutine to terminate at several different places depending on a condition.

You can nest subroutines. See Appendix B for the maximum nesting depth allowed on your operating system. Nesting occurs when a subroutine is called during the execution of another subroutine. Upon execution of the first **RETURN** statement, control passes to the statement immediately following the **GOSUB** statement last executed. The next **RETURN** statement passes control to the next to last executed **GOSUB** statement and so on.

GOSUB...RETURN

continued

Business BASIC also provides special prewritten subroutines to handle special functions. You may take advantage of these subroutines by using **ENTER** to merge them with your program. Be careful, however, because these subroutines have predefined line numbers! Use **GOSUB** to transfer to the predefined entry points of these subroutines. All line numbers for Business Basic subroutines fall between 7500 and 9999.

Examples

1. Use **GOSUB** to enter the same **FOR...NEXT** loop twice.

```
00010 DIM REPLY$[1]
00020 INPUT "INPUT ANY NUMBER BETWEEN 1 - 10: ",A
00030 GOSUB 00100
00040 LET A=A+5
00050 GOSUB 00100
00060 INPUT "RUN THIS EXAMPLE AGAIN (Y OR N) ?: " , REPLY$
00070 IF REPLY$="N" THEN STOP  ELSE GOTO 00020
00100 FOR I=1 TO A STEP 2
00110   PRINT I;
00120 NEXT I
00130 PRINT
00140 RETURN
* RUN
INPUT ANY NUMBER BETWEEN 1 - 10: 5
1 3 5
1 3 5 7 9
RUN THIS EXAMPLE AGAIN (Y OR N) ?: N
STOP AT 00050
```

2. Nested subroutines.

```
* LIST
00500 GOSUB 00530
00510 PRINT "EXAMPLE"
00520 STOP
00530 PRINT "NEST";
00540 GOSUB 00570
00550 PRINT "TINE ";
00560 RETURN
00570 PRINT "ED ";
00580 GOSUB 00610
00590 PRINT "ROU";
00600 RETURN
00610 PRINT "SUB";
00620 RETURN
* RUN
NESTED SUBROUTINE EXAMPLE
STOP AT 00520
```

3. This example shows how to use the prewritten subroutine—**DELREC.SL**. First you must use **ENTER "DELREC.SL"** to merge it with your current program. The subroutine takes up line numbers 8600 through 8695, but 8600 is the only entry point. **DELREC.SL**, like all the prewritten subroutines, already has a **RETURN** statement—you need not provide one. F% holds the logical file number, and R1 holds the record number.

```
* ENTER "DELREC.SL"
* LIST 1200,1220
01200 LET F%=2
01210 LET R1=X3
01220 GOSUB 08600 : \ DELREC.SL
*
```

GOTO

Statement

Goes to a specific statement.

AOS/VS	DG/RDOS	UNIX

Format

GOTO *line-number*

Arguments

line-number	An existing program statement line number.

What It Does

GOTO transfers control to the statement specified by *line-number*. If the statement at that line number is executable, then the statement and those following it are executed in sequential order. If the statement at *line-number* is a nonexecutable statement (like **DATA** or **REM**), then Business BASIC executes the first executable statement following the statement at *line-number*.

How to Use It

If *line-number* is not a line number in the program, an error occurs when you execute the **GOTO** statement. If *line-number* is the same as the line number of the **GOTO** statement, the program will be in an infinite loop (a **GOTO** should never "go to" itself). See Appendix B for the range of line numbers allowed on your operating system.

Example

In this example, control passes back to line 40 until all values for AGE in the file **AGES** have been read. Then the average age is calculated.

```
00010 LET TOTAL=0
00020 LET COUNT=0
00030 OPEN FILE[0,3],"AGES"
00040 READ FILE[0],AGE
00050 IF EOF(0) THEN GOTO 00100
00060 IF AGE<1 THEN GOTO 00200
00060 LET COUNT=COUNT+1
00070 LET TOTAL=TOTAL+AGE
00080 GOTO 00040
00100 LET AVGAGE=TOTAL/COUNT
00110 PRINT TOTAL,COUNT,AVGAGE
00120 END
00200 PRINT "ILLEGAL AGE = ";AGE
00210 GOTO 00040
```

GPOS

Function

Returns the current file pointer position.

AOS/VS	DG/RDOS	UNIX

Format

GPOS(*channel*)

Arguments

channel	A numeric expression for the channel number of a file opened for sequential or random access.

What It Does

GPOS returns the relative byte number in the file (specified by *channel*) to which the file pointer is positioned. Note that the last access to the file sets the file pointer's current position. **GPOS** returns -1 if the file specified by *channel* is not open. Since the byte position after a **BLOCK READ FILE** is undefined, you cannot use **BLOCK I/O** with this function.

How to Use It

Use **GPOS** to determine the byte to which the file pointer is positioned after a file access. An input or output to a file causes the file pointer to move to the byte immediately following the last byte read or written. **GPOS** indicates where the next input or output starts.

Use **GPOS** as a numeric expression wherever numeric expressions are permitted. For example, you may use it with **IF...THEN** statements to conditionally transfer to another routine, depending on the value of the file pointer.

On UNIX systems, in order to obtain a 0 when using **GPOS** on a device file or a queue file, you must place the device or queue filename in the **DEVICE_MAP** file.

Examples

1. **GPOS** refers to an unopened file.

```
00030 LET B = GPOS(0)
00040 PRINT B
00050 END
* RUN
-1
*
```

GPOS

continued

2. **GPOS** references an open file.

```
* LIST
00010 DIM X$(512)
00020 OPEN FILE(1,0),"JUNK"
00030 WRITE FILE (1), "THIS IS A JUNK FILE"
00040 CLOSE
00050 OPEN FILE(1,0)"JUNK"
00060 PRINT "THIS IS THE GPOS OF FILE 1 AFTER THE OPEN: ";GPOS(1)
00070 READ FILE(1),X$
00080 PRINT "THIS IS THE GPOS OF FILE 1 AFTER THE READ: ";GPOS(1)
* RUN
THIS IS THE GPOS OF FILE 1 AFTER THE OPEN: 0
THIS IS THE GPOS OF FILE 1 AFTER THE READ: 19
*
```

IF...THEN...ELSE

Statement

Transfers to or executes a statement if a condition is true.

AOS/VS	DG/RDOS	UNIX

Formats

1. **Single-Line Format**

   IF *expression* [**THEN**] *statement* [**ELSE** *statement*]

2. **Block Structured Format**

```
IF expression THEN
•
• (Block of executable statements)
•

    [ELSE
    •
    • (Block of executable statements)
    •]

END IF
```

Arguments

expression	A numeric expression or variable, but not a string. If the expression is 0, the condition is false; if the expression does not equal 0, the condition is true.
statement	Any legal Business BASIC statement except **CON**, **DATA**, **DO/WHILE/UNTIL**, **END**, **END LOOP**, **FOR**, **MSG**, **NEXT**, **ON ERR**, **ON IKEY**, **REM**, and **RFORM**. You can use **ON GOTO**, **ON GOSUB**, **DIM**, and any commands you use as statements.
line-number	Any line number that exists in your program.

What It Does

IF makes a decision based on the value of an expression or the logical answer to the relational expression. For example, if *expression* equals 0, the condition is false; if the expression is non-zero, the condition is true. If the relational expression A>B is true, then the condition is true; if not, then the condition is false. If the condition is true, the statement after **THEN** is executed; if it is false, control passes to the statement following the **IF** statement. Also note that the statement

IF...THEN...ELSE

continued

IF *variable* **THEN GOTO** *line-number*

is the same as

IF *variable* **<>** 0 **THEN GOTO** *line-number*

How to Use It

Use **IF** only as a program statement. You can use *line-number* or **GOTO** *line-number* after **THEN**: it's still a simple **GOTO** transfer. To make a **GOSUB** transfer, you must specify both **THEN** and **GOSUB**. Otherwise, you can use most Business BASIC statements after **THEN**, including another **IF** statement. If you have a **GOSUB** transfer, the subroutine's **RETURN** statement returns control to the statement immediately following the **IF**.

The expressions in the relational expression can be numeric or string, but you can only compare numeric to numeric and string to string—never numeric to string. When comparing strings or substrings using the equal (=) sign, **IF** matches the strings character by character until it finds a difference, if any. If you're comparing strings using the greater than (>) or less than (<) signs, **IF** compares the ASCII code value of the first characters of each string. To be equal, both strings must have the same characters in the same order and must have the same length. Even one extra null at the end of one of two otherwise identical strings makes the strings unequal.

The rules for **IF...THEN...ELSE** are the same as for **IF** plus the following:

1. Each **ELSE** must have a corresponding **IF**.
2. **ELSE** refers to the most recent **IF** condition before the **ELSE**.

CAUTION: When nesting **IF...THEN...ELSE** statements, you can make subtle logic errors that will not give you syntax errors. For example, suppose you wish to print GREATER when A is greater than 0 and EQUAL when A equals 0. If you enter:

```
* 10 IF A <> 0 THEN IF A > 0 THEN PRINT "GREATER"  ELSE
 PRINT  "LESS"
```

LESS prints when A is less than zero because **ELSE** refers to the most recent **IF**. Business BASIC processes **ELSE** when the **IF** condition is false. In this example, the false condition of **IF A > 0** is A <= 0. You will not get a syntax error but your results will be incorrect. For the correct results enter:

```
* 10 IF A = 0 THEN PRINT "EQUAL" ELSE IF A > 0 THEN
PRINT "GREATER"
```

IF...THEN...ELSE

Block Structured Format

The block structured **IF...THEN** and **IF...THEN...ELSE** statements allow groups of statements to be conditionally executed where the group is bounded by the **END IF** statement. This format requires the **IF** (*expression*) **THEN** sequence to be alone on a line (not the object of a simple **IF**) and the word **THEN** is not optional. The **ELSE** and **END IF** statements must appear alone on lines. The **END IF** statement is always expected to mark the end of the group of statements controlled by a block **IF**. When *expression* is true, subsequent statements are executed until a matching **ELSE** statement or an **END IF** statement is encountered. Encountering an **ELSE** causes subsequent statements to be skipped until a matching **END IF** statement is found.

If *expression* is not true, subsequent statements are skipped until a matching **ELSE** or **END IF** statement is found. Encountering an **ELSE** causes subsequent statements to be executed until a matching **END IF** statement is found. Both simple **IF** statements and block **IF** statements can be used inside of the blocks of statements bounded by **THEN** and **ELSE** or **THEN** and **END IF** or **ELSE** and **END IF**. Mismatching **IF...THEN** and **END IF** pairs or **IF...THEN...ELSE** and **END IF** sets may result in runtime errors. Extra **END IF** statements are ignored. An unexpected **ELSE** (no active **IF** block) causes an error. Reaching the end of a program when an **ELSE** or **END IF** is expected (false block **IF**) causes an error.

Examples

1. This example compares two strings.

```
* LIST
00010 DIM A$[29], B$[29]
00020 LET A$="ABCDEFGHIJKLMNOPQRS"
00030 LET B$="ABCDEFGHIJKLMNOPQRS"
00040 IF A$=B$ THEN PRINT "TRUE" ELSE PRINT "FALSE "
00050 END
* RUN
TRUE
```

2. This example uses the Boolean operator **AND** in an **IF** statement. For other examples see **AND**, **NOT**, and **OR**.

```
* LIST
00010  LET A=1
00020  LET B=0
00030  IF A AND B THEN GOTO 00060
00040  PRINT "EITHER A OR B IS ZERO"
00050  STOP
00060  PRINT "BOTH A AND B ARE NON-ZERO"
00070  STOP
* RUN
EITHER A OR B IS ZERO
STOP AT 00070
```

IF...THEN...ELSE

continued

3. This **IF** statement uses **ON GOTO** statements.

```
* LIST
00005  INPUT A,B,C
00010 IF A=1 THEN ON B THEN GOTO 00110, 00120 ELSE ON C THEN
GOTO 00130, 00140
00040 PRINT "PART OF LINE 00010 IS FALSE"
00050 PRINT "EITHER B > 2  OR  C > 2"
00060 PRINT "B = ";B,"C = ";C
00070 STOP
00110 PRINT "TRUE A = 1   B = 1"
00115 STOP
00120 PRINT "TRUE A = 1   B = 2"
00125 STOP
00130 PRINT "FALSE A <> 1   C = 1"
00135 STOP
00140 PRINT "FALSE A <> 1   C = 2"
00145 STOP
* RUN
? 2 ? 2 ? 2
FALSE A <> 1  C = 2
STOP AT 00145
* RUN
? 2 ? 1 ? 4
PART OF LINE 00010 IS FALSE
EITHER B > 2  OR  C > 2
B = 1     C = 4
STOP AT 00070
```

4. If A=1 then Business BASIC sets B=2. If A <> 1 then Business BASIC checks if A=2. If A=2 then BASIC sets C=4; if A <> 2 then Business BASIC sets C=99.

```
* LIST
00010 LET A,B,C=0
00020 INPUT "ENTER VALUE FOR A: ",A
00030 IF A=1 THEN LET B=2 ELSE IF A=2 THEN LET  C=4 ELSE LET C=99
00040 PRINT "VALUE FOR A IS NOW ";A
00050 PRINT "VALUE FOR B IS NOW ";B
00060 PRINT "VALUE FOR C IS NOW ";C
00070 END
* RUN
ENTER VALUE FOR A: 3
VALUE FOR A IS NOW 3
VALUE FOR B IS NOW 0
VALUE FOR C IS NOW 99
```

continued **IF...THEN...ELSE**

5. Block **IF** statement.

```
* LIST
00010 LET X,Y,Z=0
00020 INPUT "ENTER VALUE FOR X: ",X
00030 INPUT "ENTER VALUE FOR Y: ",Y
00040 IF X=Y THEN             :Block IF executes statements 00050
00050   LET Z=7               :and 00060 (and associated GOSUB
00060   GOSUB 00200 :  X EQUALS Y   :statement) when X equals Y
00070 END IF
00080 END
00200 PRINT "X IS EQUAL TO Y, Z IS EQUAL TO: ";Z
00220 RETURN
```

6. This example is a modification of example 4 to show the use of **ELSE** in a block **IF**.

```
* LIST
00010 LET X,Y,Z=0
00020 INPUT "ENTER VALUE FOR X: ",X
00030 INPUT "ENTER VALUE FOR Y: ",Y
00040 IF X=Y THEN             :Block IF executes statements 00050
00050   LET Z=7               :through 00060 when X equals Y
00060   GOSUB 00200                    :X EQUALS Y
00070 ELSE                             :and statements 00080
                                       :through 00090
00080   LET Z=3               :when X does not equal Y
00090   GOSUB 00300                    :X DOES NOT EQUAL Y
00100 END IF
00110 END
00200 REM  X EQUALS Y
00210 PRINT "X IS EQUAL TO Y, Z IS EQUAL TO:";Z
00220 RETURN
00300 REM  X DOES NOT EQUAL Y
00310 PRINT "X IS NOT EQUAL TO Y, Z IS EQUAL TO :";Z
00320 RETURN
```

INPUT

Statement and Command

Enters data-sensitive input from a terminal or file.

AOS/VS	DG/RDOS	UNIX

Format

INPUT { FILE(*channel*) | [@(*number*),]["*prompt*",] } *variable*[,*variable*...][;]

Arguments

channel	The channel number of a file if inputting from a file; the file must be in character format.
@(number)	Cursor positioning and terminal control expressions (@ expressions) are described with **PRINT**.
prompt	A string literal in quotation marks that is output as a prompt for an input request from a terminal. This prompt replaces the question mark (?) prompt. On AOS/VS and DG/RDOS systems, you can enclose ASCII values in angle <> brackets within the string literal. **Note:** Not all terminals use the same ASCII values to perform the same functions.
variable	A numeric variable, string variable, or array variable, depending on the values you want to input. You can also use subscripted string variables and array variables. Dimension string variables and arrays before using them. Any combination of the above is allowed.
;	A semicolon at the end of an **INPUT** positions the cursor to the space immediately following the input prompt.

What It Does

INPUT enters numeric and string data from a terminal or file. **INPUT** and **INPUT FILE(16)** perform terminal input; **INPUT FILE**(*channel*) performs file input.

INPUT reads text that is terminated by special characters and is meant to be used with **PRINT** on text files and for terminal I/O. To perform binary I/O on data and other types of files, use **READ** and **WRITE**.

How to Use It

When terminal input is requested, the prompt argument is displayed on the screen and Business BASIC waits until the input is terminated, usually by an unpend key. If no prompt is specified, a question mark is used. Respond to the prompt by entering values.

continued **INPUT**

If the **INPUT** statement has only one variable, enter only one value. If the **INPUT** statement has more than one variable and you enter only one value, Business BASIC requests more values by outputting more prompts (**INPUT USING** gives an error in this case). You can supply values by answering each prompt, or you can type all the values at once by separating them with commas. If the input is of the wrong type (i.e., a string response to a numeric input), a bell is rung and a prompt (\?) is displayed to indicate a request for a valid response (during file input or **INPUT USING**, an error occurs).

Prompt arguments and terminal control functions are ignored by **INPUT FILE** (except **INPUT FILE(16)**).

The terminators for **INPUT FILE** are null, form feed, Carriage Return, and New Line. The primary and secondary unpend keys defined by **STMA 4** are used by **INPUT** and **INPUT FILE(16)**. Also, several terminal control functions are available to position on the screen, to specify the maximum length of an input, to define whether echo is allowed, etc. (Refer to **PRINT** for a description of these functions.)

All forms of **INPUT** ignore leading blanks and assume commas indicate the end of one variable and the beginning of the next variable. To input a string containing leading blanks or commas, enclose the string in quotation marks or use **INPUT USING**. **INPUT** converts values enclosed in angle brackets (< >) to the ASCII character indicated by the value.

Examples

1. Use **INPUT** to request data.

```
00010 INPUT A,B,C
00020 PRINT A+B,B+C
* RUN
?1,2,3
3      5
*
```

2. Use **INPUT** to display a prompt requesting data.

```
00010 INPUT "A,B,C:",A,B,C
00020 PRINT A+B,B+C
* RUN
A,B,C:1,2     ?3
3      5
*
```

3. This example shows use of the semicolon at the end of the **INPUT** statement to allow the next output to appear on the same line.

```
00010 INPUT A,B,C;
00020 PRINT "SAME LINE"
* RUN
? A\ ? 1 ? 2,3 SAME LINE
*
```

INPUT

4. This example uses triple precision and **INPUT FILE.**

```
00010 DIM X$[32]
00020 OPEN FILE [0,0],"JUNK"
00030 POSITION FILE [0,0]
00040 INPUT "Double Precision: ",A
00050 INPUT "Triple Precision: ",B#
00060 INPUT "Single Precision: ",C%
00070 PRINT FILE[0],A,B#,C%
00080 LET X=GPOS(0)
00090 PRINT "file position after PRINT FILE: ";X
00100 POSITION FILE [0,0]
00110 INPUT FILE[0],X$
00120 PRINT "data after INPUT FILE"
00130 PRINT X$
* RUN
Double Precision: 20
Triple Precision: 300
Single Precision: 1
file position after PRINT FILE:  32
data after INPUT FILE
20        300      1
```

5. This is another example using triple precision.

```
00020 OPEN FILE [0,0],"JUNK"
00030 POSITION FILE [0,0]
00040 INPUT "Double Precision: ",A
00050 INPUT "Triple Precision: ",B#
00060 INPUT "Single Precision: ",C%
00070 PRINT FILE [0], A
00080 PRINT FILE [0],B#
00090 PRINT FILE [0],C%
00100 LET X=GPOS(0)
00110 PRINT "file position after PRINT FILE: ";X
00120 POSITION FILE [0,0]
00130 INPUT FILE [0],D,E#,F%
00140 PRINT "DATA AFTER INPUT FILE"
00150 PRINT "D IS ";D
00160 PRINT "E# IS ";E#
00170 PRINT "F% IS ";F%
00180 STOP
```

continued **INPUT**

```
* RUN
DOUBLE PRECISION: 20
TRIPLE PRECISION: 300
SINGLE PRECISION: 1
FILE POSITION AFTER PRINT FILE:  15
DATA AFTER INPUT FILE
D IS 20
E# IS 300
F% IS 1
STOP AT 00180
*
```

INPUT USING

Statement and Command

Inputs character data from a terminal or file allowing an error and a delimiter trap.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{INPUT [FILE(}\mathit{channel}\text{)] USING"",[}\mathit{prompt}\text{,]} \left\{ \begin{array}{l} \mathit{variable}\text{[,}\mathit{variable}\text{...]} \\ \mathit{string\text{-}variable1}\text{[,}\mathit{string\text{-}variable2}\text{]} \end{array} \right\} \text{[;]}$$

Arguments

channel	The channel number of an opened file if inputting from a file; file must be in character format.
prompt	A string literal in quotation marks that is output as a prompt for an input from the terminal (replaces the question mark (?) prompt). You can enclose ASCII values in angle brackets (<>), and you can use unquoted @ cursor positioning and terminal control expressions (described with **PRINT**). **Note:** Not all terminals use the same ASCII values to perform the same functions.
variable	A numeric variable or array (already dimensioned) depending on the values you want to input. Any combination of the above is allowed if you repeat the variable.
string-variable1	Only one string variable or substring is allowed; a second string variable or substring would automatically be treated as *string-variable2*; a third would cause an error. String variables must be dimensioned.
string-variable2	A special string variable or substring after *string-variable1* that receives the value of the delimiter.
;	A semicolon at the end of an **INPUT** positions the cursor to the space immediately following the input prompt.

What It Does

INPUT USING works similarly to **INPUT FILE**, with the following differences:

- All values must be input at once. Multiple numeric values must be separated by commas.
- If the input value to *variable* cannot be converted into a number, **INPUT USING** causes a normal error that you can trap with an **ON ERR** statement. If you use *string-variable1*, the entire input list of values is transferred unedited to *string-variable1* (i.e., punctuation marks included), up to the maximum that *string-variable1* can hold.

- If *string-variable2* is included in the **INPUT USING** statement, Business BASIC places the terminating character in *string-variable2*, which can be a string expression or a string variable dimensioned to at least one byte.

How to Use It

Use **INPUT USING** just like **INPUT FILE**, with these exceptions:

- If you do not satisfy the input request, a normal error occurs that you can trap with an **ON ERR** statement.
- If you use *string-variable2*, it receives the delimiter.
- For AOS/VS systems, when you are using terminal type 8 and the AOS characteristic FKT is on, the two-byte sequence generated by a function key is placed in *string-variable2*. This is also true for UNIX systems when you are using terminal type 8 and you include the **-F** option on the command line when you execute Business BASIC.

Examples

1. This example demonstrates **INPUT USING** from a file. A$(LEN(A$)+1) is the expression for *string-variable2*. It puts the delimiter immediately after the rest of the data in A$.

```
00010  DIM A$(132)
00020  OPEN FILE (0,0), "ASCIIDATA"
00030  INPUT FILE (0), USING "", A$, A$(LEN(A$)+1)
00040  IF EOF(0) GOTO 00300
00050  OPEN FILE (1,1), "NEWFILE"
00060  PRINT FILE (1),A$
00070  GOTO 00030
00300  CLOSE
00310  STOP
```

2. In this example, the input is provided by the user. Notice that the numeric input values are separated by commas.

```
00010  DIM X$(50),A$(1)
00015  INPUT USING "","INPUT NUMBERS FOR A, B AND  C: ",A,B,C
00020  INPUT USING "","TYPE ANYTHING HERE ! ",X$
00025  PRINT "A IS ";A,"B IS ";B,"C IS ";C
00030  PRINT X$
* RUN
INPUT NUMBERS FOR A, B AND C: 1
ERROR 46 AT 00015 - INPUT INVALID
* RUN
INPUT NUMBERS FOR A, B AND C: 1,2,3
TYPE ANYTHING HERE ! ASDFGHJKL;?!@#$ %&*+=
A IS 1     B IS 2     C IS 3
ASDFGHJKL;?!@#$%&*+=
```

INT

Function

Truncates a number to make it an integer.

AOS/VS	DG/RDOS	UNIX

Format

INT(*expression*)

Arguments

expression	A numeric expression or variable.

What It Does

Since Business BASIC allows only integer data, this function has no effect. Business BASIC supplies the INT function to enhance the compatibility of the product.

INT truncates the expression to form the greatest integer not larger than the expression. It does not round. For example, if the value of an expression is 8.9999, the INT function returns 8, not 9.

How to Use It

Use the INT function as a numeric expression wherever you are permitted to use numeric expressions.

Example

```
* PRINT INT(7/4) 1
```

KADD

Statement and Command

Adds a key to an index file.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{KADD}\left\{\begin{array}{l}\textit{descriptor-string}\\ \textit{logical-file-number}\end{array}\right\}\textit{,buffer-string,key-string,record-number}$$

Arguments

descriptor-string	A regular or subscripted string variable. To be able to lock records, dimension this variable to, or provide a substring of, at least 18 bytes. The variable or substring contains the channel number, a byte offset, an automatic lock flag, and the name of the index file. If you do not need to lock records, the string variable or substring must be at least 8 bytes long.
logical-file-number	A numeric expression representing the logical file number of an index file that has been opened with the **LOPEN FILE** statement.
buffer-string	A regular or subscripted string variable dimensioned to, or having a substring length of, at least 544 bytes. This argument is used as a buffer to hold an index block while performing I/O to an index file.
key-string	A regular or subscripted string variable that is the key entry. It must be dimensioned to, or have a substring length of, exactly the same number of bytes as the key (or the length of the key when crammed, if you are using the **CRM$** function). The maximum key length is 122 bytes.
record-number	A numeric variable for a data record number greater than zero to which you want the key to point. The data file need not exist yet.

What It Does

Use **KADD** to add key entries to an index file. **KADD** adds *key-string* and *record-number* to the index file described in *descriptor-string* or represented by *logical-file-number*. **KADD** pulls index blocks into *buffer-string* and searches them to find the proper location for the key entry. The index is sorted in ascending order. If you attempt to add a −1 key to the index, the key is not added and *record–number* is returned with a value of zero. If duplicate keys are not allowed and **KADD** finds a duplicate key, the new key is not added and *record-number* is returned with a value of zero. If you are using duplicate keys, the key is added. You select duplicate key usage when you create the index file.

KADD *continued*

NOTE: Because **KADD** does not check for duplicate record numbers, duplicate keys can specify the same record number. The programmer supplies the record number.

How to Use It

You must create and open an index file before adding keys to it.

NOTE: There are two restrictions on using a 2048-byte index block. First, all opens on 2048-byte indexes must be in a shared mode (4 or 5). You cannot access a 2048-byte index that you have opened exclusively. If you open this kind of index file exclusively and then try to execute a **KADD** statement, `Error 89 – Illegal file type` is displayed. You can get exclusive access to a 2048-byte block index by using Access Control Lists (AOS/VS) or permissions (UNIX) to the file. Second, if you use the logical file approach to Business BASIC file usage or have subfiles within a master file, you *must* ensure that the index file begins on a sector boundary that is a multiple of four in the physical file. (Under the 512-byte index structure, it is only necessary that the index file begin on a sector boundary.) Place all 2048-byte index files at the beginning of the physical file to avoid wasting unused sectors between a data file and an index file.

For Accessing Logical Index Files

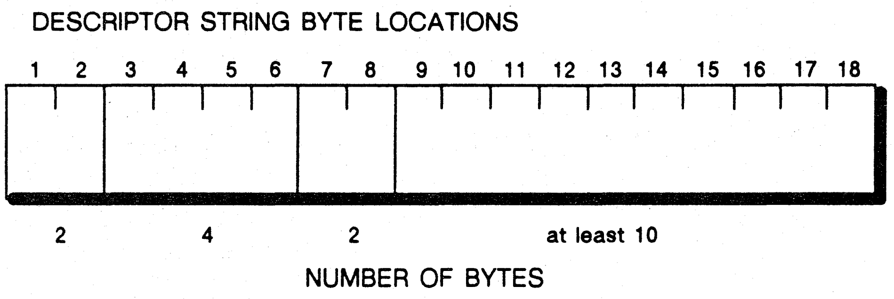
To use **KADD** to access an index file that is part of a logical file database structure, you must have a *logical-file-number*, dimension *buffer-string* to 544 bytes, and supply *key-string* and *record-number*.

For Accessing Index Files Defined in PARAM

To use **KADD** to access an index file that is part of a PARAM file database structure, you must have a *descriptor-string*, dimension *buffer-string* to 544 bytes, and supply *key-string* and *record-number*. You can build one *descriptor-string* in your program and use it for any **KADD**, **KDEL**, **KFIND**, or **KNEXT**. Your *descriptor-string* must include the following 18 bytes of information:

- The physical channel number of the index file (two bytes).
- The byte offset to block zero of the index file in the physical file, or zero if index file is not a subfile (four bytes).
- A zero when you want automatic locking of the index file to occur or a one for no automatic locking (two bytes). If you request automatic locking, then block 0 of the index is locked.
- The logical filename for the index file (not needed if you don't use automatic locking) padded with nulls to the end of the string (ten bytes).

Figure 1–6 defines each byte location of the index file descriptor string.



where byte locations:

1-2	= physical channel number of index file
3-6	= byte offset to block 0 of index in physical file if index is a subfile, or 0 if index is not a subfile
7-8	= flag for index auto-lock is 0 if lock is to occur and you need the filename; 1 if no lock and you do not need the filename
9-18+	= logical filename (string), padded with nulls

Figure 1-6 Index File Descriptor String

All Uses of KADD

For all uses of **KADD**, you must dimension *key-string* to the maximum number of bytes per key entry for the particular index file. The maximum legal dimension for *key-string* is 122 bytes. Key entries to the same index file always have the same maximum number of bytes. Keys can be alphanumeric, concatenated subkeys, crammed strings (using the **CRM$** function), or any combination of these assigned to *key-string*. (See **CHR$** for information on putting the binary value of a numeric key into *key-string*.)

Add keys after you write their data records. This prevents other users from trying to read the record to which the keys point before that data record is written. In multiple-user systems, allow automatic locking of the index file and lock the data file. The **GETREC** statement automatically locks record 0 of the data file. If your program does not use the **GETREC** statement, use **LOCK** for this purpose.

KADD

continued

Examples

1. This example illustrates the use of the **KADD** statement to add a key to an index file that is part of a PARAM file database structure.

```
00010 DIM D$[18],B$[544],KEYID$[10]   :For descriptor, buffer, key
00020 DIM X$[512],C1[1,3],RECORD$[28] :For C1 array, record, OPEN
00030 LET X$="DATA1,6,INDEX1,6",FILL$(0):Files to open in mode 6
00040 BLOCK WRITE X$                 :String to send to OPEN
00050 SWAP "OPEN"                    :OPEN opens files.
00060 BLOCK READ X$                  :Returns C1 information.
00070 LET K=1                        :Initialize pointer.
00080 FOR I=0 TO 1                   :For each row of C1,
00090   FOR J=0 TO 3                 :and for each column,
00100     LET C1[I,J]=ASC(X$[K,K+3])  :get information for C1.
00110     LET K=K+4                   :Increment pointer.
00120   NEXT J
00130 NEXT I
00140 REM *fill descriptor string*
00150 LET D$=CHR$(C1[1,0],2),CHR$(C1[1,1]4):Channel and offset,
00160 LET D$[0]=CHR$(0,2),"INDEX1",FILL$(0):auto-lock of INDEX1.
00170 INPUT "Type key: ",KEYID$       :Input key.
00180 INPUT "Type record: ",RECORD$   :Input record.
00190 LET F%=0                        :Logical file 0 is DATA1.
00200 LOCK 2,"DATA1",0,C1[0,3]        :Lock record 0 of DATA1.
00210 GOSUB 08400:  GETREC           :Get available record of DATA1.
00220 UNLOCK 2                       :Unlock record 0 of DATA1.
00230 GOSUB 09610:    POSFL  :Position using R1 from GETREC.SL.
00240 WRITE FILE[C%],RECORD$:Write record using C% from:POSFL.SL.
00250 KADD D$,B$,KEYID$,R1  :Add new key, R1 is record pointer.
00260 IF R1<=0 THEN GOTO 00500       :If KADD not successful, go
                                     :to 500. (R1<=0 if error
                                     :occurred, or if duplicate
                                     :keys are not allowed and
                                     :you gave KADD one.
                                     :Otherwise, R1 equals the
                                     :record in DATA1).
```

continued **KADD**

2. This example illustrates the use of the **KADD** statement to add a key to an index file that is part of a logical file database structure.

```
00005 DIM KEYID$[10],RECORD$[28],T9$[544],RECNO[0]
00010 DIM LFTABL$[52]
00020 LET LFTABL$=FILL$(0)          :Initialize local file table.
00030 LOPEN FILE[1,T9$],"DATA1"     :Open  logical data file.
00040 LOPEN FILE[2,T9$],"INDEX1"    :Open logical index file.
00050 INPUT "Type key: ",KEYID$     :Input key.
00060 INPUT "Type record: ",RECORD$ :Input  record.
00080 GETREC 1,RECNO                :Get the next available
00090 LWRITE FILE[1,RECNO],RECORD$  :record in DATA1, write the
                                    :record.
00100 KADD 2,T9$,KEYID$,RECNO       :Add key-string to INDEX1.
00110 IF RECNO<=0 THEN GOTO 00500 ELSE GOTO 00600
00500 PRINT "KADD WAS NOT SUCCESSFUL"         :If key was not
                                              :added, recno<=0.
00600 CLOSE
00610 END
```

KDEL

Statement and Command

Deletes a key from an index file.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{KDEL} \left\{ \begin{array}{l} \textit{descriptor-string} \\ \textit{logical-file-number} \end{array} \right\} \textit{,buffer-string,key-string,record-number}$$

Arguments

descriptor-string	A regular or subscripted string variable. To be able to lock records, dimension this variable to, or provide a substring of, at least 18 bytes. The variable or substring contains the channel number, a byte offset, an automatic lock flag, and the filename of the index file. If you do not need to lock records, the string variable or substring must be at least 8 bytes long.
logical-file-number	A numeric expression that represents the logical file number of an index file opened using **LOPEN FILE**.
buffer-string	A regular or subscripted string variable dimensioned to, or having a substring length of, at least 544 bytes. This argument is used as a buffer to hold an index block while performing I/O to an index file.
key-string	A regular or subscripted string variable, that is the key entry. It must be dimensioned to, or have a substring length of, exactly the same number of bytes as the key (or the length of the key when crammed, if you are using the **CRM$** function). The maximum key length is 122 bytes.
record-number	An initialized numeric variable that receives the record pointer. You need to specify *record-number* only if you are using duplicate keys in the index file. *record-number* is returned by **KDEL** with the value of the record pointer if **KDEL** was successful or zero if **KDEL** did not find the key.

What It Does

KDEL searches for *key-string* and then, if duplicate keys are allowed, for *record-number*. If it does not find a match for *key-string* or for *key-string* and *record-number*, it returns *record-number* with a value of zero; otherwise, Business BASIC deletes *key-string* and *record-number* entries in the index and returns *record-number* with the record pointer.

How to Use It

To use **KDEL**, put the index file information into *descriptor-string* or the logical file represented by *logical-file-number* (see **KADD**), dimension *buffer-string* to 544 bytes, supply a value for *key-string*, and, if your index file has duplicate keys, supply a value for *record-number*. Otherwise, supply an initialized variable for *record-number* so that it can return with a value for you to check.

NOTE: There are two restrictions on using a 2048-byte index block. First, all opens on 2048-byte indexes must be in a shared mode (4 or 5). You cannot access a 2048-byte index that you have opened exclusively. You can get exclusive access to a 2048-byte block index by using Access Control Lists (AOS/VS) or permissions (UNIX) to the file. Second, if you use the logical file approach to Business BASIC file usage or have subfiles within a master file, you *must* ensure that the index file begins on a sector boundary that is a multiple of four in the physical file. (Under the 512-byte index structure, it is only necessary that the index file begin on a sector boundary.) Place all 2048-byte index files at the beginning of the physical file to avoid wasting unused sectors between a data file and an index file.

When you delete the index key for a data record, first lock the record to make sure no one is using it and to prevent anyone from locking it. Then delete the index entry and the data record respectively so that no one can use the key to access the record.

Examples

1. This example illustrates the use of the **KDEL** statement to delete a key from an index file that is part of a PARAM file database structure.

```
00010 DIM D$[18],B$[544],KEYID$[10]    :DIM for descriptor,
                                       :buffer and key.
00020 DIM X$[512],C1[1,3],RECORD$[128]:DIM for C1 array, record,
                                       :and OPEN.
00030 LET X$="DATA1,6,INDEX1,6",FILL$(0)       :Files to open in
                                       :mode 6.
00040 BLOCK WRITE X$                   :String to send to OPEN.
00050 SWAP "OPEN"                      :OPEN opens files,
00060 BLOCK READ X$                    :Returns C1 information.
00070 LET K=1                          :Initialize pointer.
00080 FOR I=0 TO 1                     :For each row of  C1,
00090   FOR J=0 TO 3                   :and for each column
00100     LET C1[I,J]=ASC(X$[K,K+3])  :extract info for C1.
00110     LET K=K+4                            :Bump pointer.
00120   NEXT J
00130 NEXT I
00140 REM *fill descriptor string*
00150 LET D$=CHR$(C1[1,0],2),CHR$(C1[1,1],4)    :Channel and
                                                :offset,
```

KDEL *continued*

```
00160 LET D$[0]=CHR$(0,2),"INDEX1",FILL$(0)     :auto-lock of
                                                :INDEX1.
00170 INPUT "Type key to be deleted: ",KEYID$   :Input key to be
                                                :deleted.
00180 LET R=0                 :Initialize R (no duplicates).
00190 KDEL D$,B$,KEYID$,R     :Delete key in key-string, return R.
00200 IF R=0 THEN GOTO 00400:If key not found, go to 400.
00210 PRINT "Deleted key was for record #";R
. . .
```

2. This example shows an example of a delete with locking. This example only works for a logical index file with no duplicate keys allowed.

```
. . .
00130 DIM LFTABL$(78),T9$(544)
00140 LET LFTABL$=FILL$(0)
00150 LOPEN FILE[2,T9$],"MYINDEX"    :Open MYINDEX as an index file.
00160 LOPEN FILE[3,T9$],"MYDATA"     :Open MYDATA as database file.
00170 LET RECNO=-1
. . .
00300 INPUT "CUSTOMER # :",CNUM      :Get customer number as key.
00310 LET KEY$=CHR$(CNUM,4)          :Convert to key string.
00320 KFIND 2,T9$,KEY$,RECNO         :Find this customer.
00330 IF RECNO<=0 THEN GOTO 04000    :Didn't find him.
00340 LOCK 1,3,RECNO                 :Lock it before reading.
00350 LREAD FILE[3,RECNO],MYDATA$    :Read his record.
. . .                                :Validate record for deletion.
00360 UNPACK "J",MYDATA$,STATUS%
00370 IF STATUS%<=0 THEN GOTO 05000 :Record already deleted.
. . .
00500 KDEL 2,T9$,KEY$,RECNO          :Delete key from MYINDEX.
00510 DELREC 3,RECNO                 :Delete data record from MYDATA.
00520 UNLOCK 1                       :Unlock data record now.
00530 END
04000 PRINT "CUSTOMER NOT FOUND"
. . .
```

KFIND

Statement and Command

Finds a key in an index file.

AOS/VS	DG/RDOS	UNIX

Format

KFIND $\left\{\begin{array}{l}\textit{descriptor-string}\\ \textit{logical-file-number}\end{array}\right\}$*,buffer-string,key-string,record-number*

Arguments

descriptor-string	A regular or subscripted string variable. To be able to lock records, dimension this variable to, or provide a substring of, at least 18 bytes. The variable or substring contains the channel number, a byte offset, an automatic lock flag, and the filename of the index file. If you do not need to lock records, the string variable or substring must be at least 8 bytes long.
logical-file-number	A numeric expression representing the logical file number of an index file that has been opened using the **LOPEN FILE** statement.
buffer-string	A regular or subscripted string variable dimensioned to, or having a substring length of, at least 544 bytes. This argument is used as a buffer to hold an index block while performing I/O to an index file. *buffer-string* must be unique to this index file if you plan to use **KNEXT** statements.
key-string	A regular or subscripted string variable that is the key entry. It must be dimensioned to, or have a substring length of, exactly the same number of bytes as the key (or the length of the key when crammed, if you are using the **CRM$** function). The maximum key length is 122 bytes.
record-number	An initialized numeric variable that receives the record pointer.

What It Does

KFIND searches the index file described by *descriptor-string* or *logical-file-number* by reading blocks into *buffer-string*. If **KFIND** locates an exact match for *key-string*, *record-number* contains the record pointer associated with that key. If **KFIND** reaches a value greater than *key-string* (i.e., no exact match is found), **KFIND** returns that value in *key-string*. To indicate that this match is approximate rather than exact, *record-number* returns the record pointer for that key as a negative number. If **KFIND** reaches the end of the index without finding either an exact match or an approximate match, it returns –1 in *key-string* and 0 in *record-number*.

KFIND

NOTE: If you are going to use **KNEXT** after **KFIND**, do not delete *buffer-string*. **KNEXT** uses the same buffer string as **KFIND**.

How to Use It

Use **KFIND** to find any key in an index file or to find the first key in an index file. You can then use **KNEXT** to read the index file sequentially from the key found by **KFIND**.

NOTE: There are two restrictions on the use of the 2048-byte index block. First, all opens on 2048-byte indexes must be in a shared mode (4 or 5). You cannot access a 2048-byte index that you have opened exclusively. If you open this kind of index file exclusively and then try to execute a **KFIND** statement, `Error 89 - Illegal file type` is displayed. You can get exclusive access to a 2048-byte block index by using Access Control Lists (AOS/VS) or permissions (UNIX) to the file. Second, if you use the logical file approach to Business BASIC file usage or have subfiles within a master file, you *must* ensure that the index file begins on a sector boundary that is a multiple of four in the physical file. (Under the 512-byte index structure, it is only necessary that the index file begin on a sector boundary.) Place all 2048-byte index files at the beginning of the physical file to avoid wasting unused sectors between a data file and an index file.

You must build a *descriptor-string* or supply a value for *logical-file-number*, dimension *buffer-string*, and supply a value for *key-string*. After using **KFIND**, check *record-number*. If *record-number* is negative, an approximate match was found. You can use **KNEXT** to get the next key in sequence. If *record-number* is zero, nothing like *key-string* was found in the index; if *record-number* is a positive record pointer, then you found the key you were looking for.

Examples

1. This example illustrates the use of the **KFIND** statement to find a key in an index file that is part of a PARAM file database structure.

```
00010 DIM D$[18],B$[544],KEYID$[10]    :For descriptor, buffer,
                                       :and key.
00020 DIM X$[512],C1[1,3],RECORD$[28]  :For C1 array, record,
                                       :and OPEN.
00030 LET X$="DATA1,6,INDEX1,6",FILL$(0):Files to open in mode 6.
00040 BLOCK WRITE X$                   :String to send to OPEN.
00050 SWAP "OPEN"                      :OPEN opens files.
00060 BLOCK READ X$                    :Returns C1 information.
00070 LET K=1                          :Initialize pointer.
00080 FOR I=0 TO 1                     :For each row of C1,
00090   FOR J=0 TO 3                   :and for each column,
00100     LET C1[I,J]=ASC(X$[K,K+3])   :extract information for C1.
```

continued **KFIND**

```
00110      LET K=K+4                     :Bump pointer.
00120    NEXT J
00130 NEXT I
00140 REM *fill descriptor string*
00150 LET D$=CHR$(C1[1,0],2),CHR$(C1[1,1],4):Channel and offset,
00160 LET D$[0]=CHR$(0,2),"INDEX1",FILL$(0):auto-lock of INDEX1.
00170 INPUT "Type key: ",KEYID$        :Input key to find.
00190 LET R=0                          :Initialize record pointer.
00200 KFIND D$,B$,KEYID$,R             :Find key return R1.
                                       :If R is 0,
00210 IF R=0 THEN GOTO 00800           :nothing found, go to 800.
                                       :If R is negative,
00220 IF R<0 THEN GOTO 00400           :approximate match found,
                                       :go to 400.
00230 PRINT "record is: ";R            :KFIND successful--print R.
00240 LET R1=R                         :Set R1 to R for POSFL.SL
                                       :routine.
00250 LET F%=0                         :Logical file 0 in C1 array
                                       :(for data file).
00260 GOSUB 09610:    POSFL
. . .
```

2. This example illustrates the use of the **KFIND** statement to find a key in an index file that is part of a logical file database structure.

```
00005 DIM KEYID$[10],RECORD$[28],T9$[544],RECNO[0]
00010 DIM LFTABL$[52]
00020 LET LFTABL$=FILL$(0)   :Initialize local file table.
00030 LOPEN FILE[1,T9$],"DATA1"        :Open logical data file.
00040 LOPEN FILE[2,T9$],"INDEX1"       :Open logical index file.
00050 INPUT "Type key: ",KEYID$                 :Input key.
00060 INPUT "Type record: ",RECORD$    :Input record.
00080 KFIND 2,T9$,KEYID$,RECNO:Find key, return record-number.
00090 IF RECNO=0 THEN GOTO 00800       :If record not found
                                       :go to 800.
00100 IF RECNO<0 THEN GOTO 00400       :Approximate match found
                                       :go to 400.
00110 PRINT "record is: ";RECNO        :Key was found.
. . .
```

KNEXT

Statement and Command

Locates the next key in an index file after a KFIND.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{KNEXT}\left\{\begin{array}{l}\textit{descriptor-string}\\ \textit{logical-file-number}\end{array}\right\}\textit{,buffer-string,key-string,record-number}$$

Arguments

descriptor-string	A regular or subscripted string variable. To be able to lock records, dimension this variable to, or provide a substring of, at least 18 bytes. The variable or substring contains the channel number, a byte offset, an automatic lock flag, and the filename of the index file. If you do not need to lock records, the string variable or substring must be at least 8 bytes long.
logical-file-number	A numeric expression representing the logical file number of an index file that has been opened using **LOPEN FILE**.
buffer-string	A regular or subscripted string variable dimensioned to, or having a substring length of, at least 544 bytes. This argument is used as a buffer to hold an index block while performing I/O to an index file. *buffer-string* must be unique to this index file.
key-string	A regular or subscripted string variable that is the key entry. It must be dimensioned to, or have a substring length of, exactly the same number of bytes as the key (or the length of the key when crammed, if you are using the **CRM$** function). The maximum key length is 122 bytes.
record-number	An initialized numeric variable that receives the record pointer.

What It Does

KNEXT returns, in *key-string*, the next key after a **KFIND** and, in *record-number*, the key's associated record pointer from the index file described in *descriptor-string* or *logical-file-number*. **KNEXT** pulls an index block into *buffer-string* and uses *buffer-string* each time **KNEXT** is executed, so do not disturb *buffer-string* between KNEXT statements. When **KNEXT** reaches the last index entry, a subsequent **KNEXT** returns *record-number* with a 0 value; any additional **KNEXT** statements cause an end-of-file error.

How to Use It

To use **KNEXT**, first use **KFIND**. You must have *descriptor-string* or *logical-file-number*, *buffer-string*, and *key-string* dimensioned to the maximum length of your keys in the index and *record-number* initialized to some value. **KNEXT** returns *key-string* and *record-number*, but you should check *record-number* to see if you reached the end of the index. **KNEXT** must have a separate *buffer-string* for each index because it needs to preserve the information in *buffer-string* for the next **KNEXT** to the same index.

NOTE: There are two restrictions on the use of the 2048-byte index block. First, all opens on 2048-byte indexes must be in a shared mode (4 or 5). You cannot access a 2048-byte index that you have opened exclusively. You can get exclusive access to a 2048-byte block index by using Access Control Lists (AOS/VS) or permissions (UNIX) to the file. Second, if you use the logical file approach to Business BASIC file usage or have subfiles within a master file, you *must* ensure that the index file begins on a sector boundary that is a multiple of four in the physical file. (Under the 512-byte index structure, it is only necessary that the index file begin on a sector boundary.) Place all 2048-byte index files at the beginning of the physical file to avoid wasting unused sectors between a data file and an index file.

Use **KNEXT** after **KFIND** to find the next key in sequence, and use subsequent **KNEXT** statements (or a **KNEXT** loop) to read the index sequentially from that point. A common use of **KNEXT** is to find duplicate keys or approximate matches; another is to process a data file in a sorted order by reading an index sequentially from the beginning.

When using **KNEXT** after a **KFIND** on a 512-byte block index, it is important to realize that the pointer for the **KNEXT** is set when the **KFIND** is done. The **KNEXT** pointer is not reset by adding or deleting an intervening key.

In AOS/VS, when using **KNEXT** after a **KFIND** on a 2048-byte block index, the pointer for the **KNEXT** always returns the next key in the index. This includes the keys added by others after the user did his original **KFIND**. In addition, **KNEXT** used with 2048-byte index blocks does not return keys deleted by another user after the original **KFIND**.

To process an entire index sequentially, call **KFIND** with a null key, thus returning the first key in the index, and then access the remainder of the index with **KNEXT**.

KNEXT

continued

Examples

1. This example illustrates the use of the **KNEXT** statement to access a key in an index file that is part of a PARAM file database structure.

```
00010 DIM D$[18],B$[544],KEYID$[10]    :For descriptor,
                                       :buffer and key
00020 DIM X$[512],C1[1,3],RECORD$[128]       :For C1 array,
                                       :record and OPEN.
00030 LET X$="DATA1,6,INDEX1,6",FILL$(0):Files to open in mode 6.
00040 BLOCK WRITE X$                   :String to send to OPEN.
00050 SWAP "OPEN"                      :OPEN opens files,
00060 BLOCK READ X$                    :Returns C1 information.
00070 LET K=1                          :Initialize pointer.
00080 FOR I=0 TO 1                     :For each row of C1,
00090   FOR J=0 TO 3                   :and for each column,
00100     LET C1[I,J]=ASC(X$[K,K+3])   :extract info for C1.
00110     LET K=K+4                    :Increment pointer.
00120   NEXT J
00130 NEXT I
00140 REM *fill descriptor string*
00150 LET D$=CHR$(C1[1,0],2),CHR$(C1[1,1],4):Channel and offset,
00160 LET D$[0]=CHR$(0,2),"INDEX1",FILL$(0):auto-lock of INDEX1.
00170 INPUT "Type key: ",KEYID$        :Input key to find.
00180 LET R=0                          :Initialize record pointer.
00190 KFIND D$,B$,KEYID$,R  :Find key, return R. If R is 0,
                            :nothing was found and goto 800.
00200 IF R=0 THEN GOTO 00800:If R is negative, an approximate
00210 IF R<0 THEN GOTO 00400:match was found, and go to 400.
00220 PRINT "record is: ";R
00230 LET R1=R              :Set R1 to R for POSFL.SL routine.
00240 LET F%=0              :Logical file 0 in C1 array (for
                            :data file).
00250 GOSUB 09610 :  POSFL  :Go to POSFL.SL to position to
                            :record R1.
. . .
00400 PRINT "Approximate Match! Key found is: ";KEYID$
                            :Approx. key's record negative.
00410 PRINT "Record associated with approximate key: ";ABS(R)
00420 INPUT "Keep looking? YES (0), No (1) :",N
00430 IF N THEN GOTO 00800
00440 KNEXT D$,B$,KEYID$,R  :Find next key and return R.
00450 IF R=0 THEN GOTO 00800:If end-of-index go to 800.
00460 PRINT "Key found is: ";KEYID$,"It's record is" ;R
                            :R is positive.
00470 INPUT "Is this it? yes(0), no(1): ",N
00480 IF N THEN GOTO 00420
00490 PRINT "OK. YOU GOT IT."
00500 GOTO 00230            :Go back to position routine.
. . .
```

continued

KNEXT

2. This example illustrates the use of **KNEXT** to access an index file that is part of a logical file database structure. This example assumes the data records are being read only, with no updating of the record. If the records were to be updated, then the record should be locked prior to reading and unlocked after rewriting.

```
00400 LET K$=""              :Null key to retrieve first entry.
00410 KFIND 2,BUF2$,K$,RNO   :Logical file 2
00420 LET RNO=ABS(RNO)       :Negative (not found) pointer
                             :expected.
00430 GOTO 00510             :Process key and pointer for first
                             :record before calling KNEXT.
00500 KNEXT 2,BUF2$,K$,RNO   :Get next entry in index.
00510 IF RNO<=0 THEN GOTO 01900       :End of index?
00520 LREAD FILE(3,RNO),RECORD$       :Read data record on logical
                                      :file 3. Process data record
01000 GOTO 00500             :Loop back for another record.
. . .
```

KPREV

Statement and Command

Locates the preceding key in an index file after a KFIND.

AOS/VS	UNIX

Format

$$\text{KPREV}\left\{\begin{array}{l}\textit{descriptor-string}\\ \textit{logical-file-number}\end{array}\right\}\textit{,buffer-string,key-string,record-number}$$

Arguments

descriptor-string	A regular or subscripted string variable. To be able to lock records, dimension this variable to, or provide a substring of, at least 18 bytes. The variable or substring contains the channel number, a byte offset, an automatic lock flag, and the filename of the index file. If you do not need to lock records, the string variable or substring must be at least 8 bytes long.
logical-file-number	A numeric expression representing the logical file number of an index file that has been opened using **LOPEN FILE**.
buffer-string	A regular or subscripted string variable dimensioned to, or having a substring length of, at least 544 bytes. Business BASIC uses this buffer to hold an index block while performing I/O to an index file. *buffer-string* must be unique to this index file.
key-string	A regular or subscripted string variable that is the key entry. It must be dimensioned to, or have a substring length of, exactly the same number of bytes as the key (or the length of the key when crammed, if you are using the **CRM$** function). The maximum key length is 122 bytes.
record-number	An initialized numeric variable that receives the record pointer.

What It Does

KPREV returns in *key-string* the value of the key preceding the one you just located with a **KFIND** command. And in *record-number*, **KPREV** returns a pointer to the record that contains *key-string*. **KPREV** pulls an index block into *buffer-string* and uses *buffer-string* each time **KPREV** is executed, so do not disturb *buffer-string* between **KPREV** statements. When **KPREV** reaches the first index entry, a subsequent **KPREV** returns 0 in *record-number*; any additional **KPREV** statements cause an end-of-file error.

How to Use It

To use **KPREV**, first use **KFIND**. Then, on your **KPREV** command line, include (1) a *descriptor-string* or *logical-file-number* that identifies the index from which you want to read, (2) a *buffer-string*, (3) a *key-string* dimensioned to the maximum length of the keys in the file, and (4) a *record-number*. **KPREV** returns *key-string* and *record-number*. After each **KPREV**, you should check *record-number* to see if you have reached the beginning of the index. **KPREV** must have a separate *buffer-string* for each index because it needs to preserve the information in *buffer-string* for the next **KPREV** involving the same index.

NOTE: There are two restrictions on the use of the 2048-byte index block. First, all opens on 2048-byte indexes must be in a shared mode (4 or 5). You cannot access a 2048-byte index that you have opened exclusively. You can get exclusive access to a 2048-byte block index by using Access Control Lists (AOS/VS) or permissions (UNIX) to the file. Second, if you use the logical file approach to Business BASIC file usage or have subfiles within a master file, you *must* ensure that the index file begins on a sector boundary that is a multiple of four in the physical file. (Under the 512-byte index structure, it is only necessary that the index file begin on a sector boundary.) Place all 2048-byte index files at the beginning of the physical file to avoid wasting unused sectors between a data file and an index file.

KPREV is valid only with 2048-byte block indexes.

Use **KPREV** after **KFIND** to find the immediately preceding key. Use subsequent **KPREV** statements (or a **KPREV** loop) to read the index sequentially from that point. You can use **KPREV** to find duplicate keys or approximate matches; or to process a data file in a sorted order by reading an index sequentially from the end.

When you use **KPREV** after a **KFIND**, the pointer for the **KPREV** always returns the preceding key in the index. This includes the keys added by others after you did your original **KFIND**. In addition, **KPREV** will not return keys deleted by another user after the original **KFIND**.

Examples

1. This example illustrates the use of the **KPREV** statement to access a key in an index file that is part of a PARAM file database structure. (Compare this example with example 1 for the **KNEXT** statement.)

```
00170 INPUT "Enter a Key: ",KEYID$     :Input key to be found.
00180 LET R=0                          :Initialize record pointer.
00190 KFIND D$,B$,KEYID$,R             :Find key, return R.
00200 IF R=0 THEN GOTO 00800           :If R is 0, then nothing was
                                       :found.
00210 IF R<0 THEN GOTO 00400           :If R<0, an approximate
                                       :match was found.
00220 PRINT "RECORD IS: ";R            :Match was found.
```

KPREV

continued

```
. . .
00400 PRINT "Approximate Match!  Key found is: ";KEYID$
00420 INPUT "Keep Looking?  YES (0), NO (1) :",N
00430 IF N THEN GOTO 00800
00433 LET R=ABS(R)
00435 INPUT "Search index for NEXT (0), or PREVIOUS (1) key :",N
00437 IF N=0 THEN
00440    KNEXT D$,B$,KEYID$,R         :Find next key and return R.
00442    GOTO 00455
00445 ELSE
00447    KPREV D$,B$,KEYID$,R         :Find previous key and
                                      :return R.
00450 END IF
00455 IF R=0 THEN GOTO 00800          :If end/beginning of index
                                      :go to 00800.
00460 PRINT "Key found is: ";KEYID$,"Its record is";R
. . .
```

2. This example illustrates the use of the **KPREV** statement to access a key in an index file that is part of a logical database structure. (Compare this example with example 2 for the **KNEXT** statement.)

```
00170 INPUT "Enter a Key: ",KEYID$    :Input key to be found.
00180 LET R=0                         :Initialize record pointer.
00190 KFIND 2,B$,KEYID$,R             :Find key in logical file 2
                                      :and return R.
00200 IF R=0 THEN GOTO 00800          :If R is 0, then nothing was
                                      :found.
00210 IF R<0 THEN GOTO 00400          :If R<0, an approximate
                                      :match was found.
00220 PRINT "RECORD IS: ";R           :Match was found.
. . .
00400 PRINT "Approximate Match!  Key found is: ";KEYID$
00420 INPUT "Keep Looking?  YES (0), NO (1) :",N
00430 IF N THEN GOTO 00800
00433 LET R=ABS(R)
00435 INPUT "Search index for NEXT (0) or PREVIOUS (1) key :",N
00437 IF N=0 THEN
00440    KNEXT 2,B$,KEYID$,R          :Find next key and return R.
00442    GOTO 00455
00445 ELSE
00447    KPREV 2,B$,KEYID$,R          :Find previous key and
                                      :return R.
00450 END IF
00455 IF R=0 THEN GOTO 00800          :If end/beginning of index
                                      :go to 00800.
00460 PRINT "Key found is: ";KEYID$,"Its record is";R
. . .
```

LEN

Function

Finds the current length of a string.

AOS/VS	DG/RDOS	UNIX

Format

LEN(*string-variable*)

Arguments

string-variable	A string variable, substring, or string array element (UNIX only).

What It Does

Although a string has a maximum dimensioned length, its dynamic length (the number of characters currently in the string) may be less than its dimensioned length. **LEN** returns the number of characters currently in *string-variable*, including spaces, punctuation, special characters and nulls. If nothing is in the string (including no nulls), **LEN** returns 0.

How to Use It

Use **LEN** as a numeric expression wherever Business BASIC allows numeric expressions. If a string is padded with nulls or spaces (after the data), you can use the **TRUN$** function (see **TRUN$**) and then **LEN** to determine where the data stops in the string.

Examples

1. **LEN** checks the length of a string.

```
00005 DIM A$(100)
00010 INPUT "TYPE 4 CHARACTERS:",A$
00020 IF LEN(A$)=4 THEN GOTO 00040
00025 PRINT "YOU BLEW IT, TRY AGAIN."
00030 GOTO 00010
00040 END
```

2. **LEN** checks the length of a dimensioned string.

```
*DIM A$(512)
*PRINT LEN(A$)
0
```

3. **LEN** checks the length of an empty string.

```
*LET A$= ""
*PRINT LEN(A$)
0
```

LEN

continued

4. **LEN** checks the length of a string assignment.

```
*LET A$=12345
*PRINT A$
12345
*PRINT LEN(A$)
5
```

5. **LEN** checks the length of a filled string.

```
*LET A$=FILL$(0,30)
*PRINT LEN(A$)
30
```

6. **LEN** checks the length of a completely filled string.

```
*LET A$=FILL$(0)
*PRINT LEN(A$)
512
```

LET

Statement and Command

Assigns a value to a variable.

AOS/VS	DG/RDOS	UNIX

Format

Simple Assignments

[LET] *variable=expression*

Multiple Assignments

[LET] *variable*[,*variable...*]=*expression*

Multiple Statement/Command

[LET] *variable*[,*variable...*]=*expression* [,*variable*[,*variable...*]=*expression*]

Arguments

variable	A numeric or string variable that can be subscripted. If you use a numeric variable, **LET** initializes the variable. If you use a string variable, you must have already dimensioned it.
expression	A numeric or string expression. If *variable* is numeric, *expression* must be numeric. Numeric expressions and relational expressions can contain subscripted numeric variables, numeric functions, user-defined functions, and constants. If *variable* is a string, *expression* can include string variables, string array elements (UNIX only), string functions, string literals in quotation marks, and even numeric expressions without quotation marks in order to make a string of digits.

What It Does

You can use **LET** to create a numeric variable and assign a value to it. For multiple assignments, the variables on the left side of the **LET** statement must be of the same data type. You can also use **LET** statements to create an 11-element, one-dimensional array or a 121-element, two-dimensional array without having to dimension the arrays. However, do not assign a string value to a string variable without dimensioning the string variable first.

Use an ampersand (&) suffix as part of your variable name to indicate a quadruple precision variable (such as FOUR&), a pound sign (#) to indicate a triple precision variable (such as THREE#), and a percent sign (%) to indicate a single precision variable (such as ONE%). No suffix is required for double precision variables. If your variable holds a value that occupies four words, use a quad precision variable to transfer the value to or from a file. If the value occupies three words, use a triple precision variable. If your system does not use triple precision, use only double and single precision variables. See *Learning Business BASIC* or *Programming with Business BASIC* for the range of values allowed with each precision.

LET *continued*

How to Use It

Use **LET** as a keyboard command to perform quick calculations or assignments when debugging a program. Entering the keyword **LET** is optional; the system inserts any missing **LET** statements when the program is listed.

Numeric expressions can have constants, variables, subscripted variables, numeric functions, user-defined functions, relational expressions, and arithmetic symbols.

String expressions can have string literals in quotation marks, string variables, subscripted string variables, string functions, string operators, and string array elements (UNIX only). See **DIM** for the syntax to use for string array elements.

On UNIX systems, to refer to the entire string in a string array element, do not specify a length (*n*), but do include a semicolon. For example, the command **X$=A$(;1,1)** refers to the entire string at element 1,1. You can refer to a substring of a particular element's string by specifying the range of bytes before the semicolon. For example, the command **X$=A$(4,9;3,4)** places bytes 4-9 of the string at element 3,4 into X$.

You can assign a numeric expression to a string variable without using quotation marks (e.g., LET A$=1234, LET B$=3+4: A$ contains the string 1234 and B$ contains the string 7). The assignment creates a string of digits that you can convert back to a number with the **VAL** function or the **VALUE** statement.

Your variable must always be on the left of the equal sign (=) and the value you assign to it must always be on the right.

See Appendix B for the maximum number of characters allowed in a variable name and the maximum number of variables allowed in one program on your operating system.

Any place that an expression is valid, you can use a relational operator. For example:

```
00340 LET A= (X>10)
00350 LET A= 10*(A$>B$)+20*(A$=B$)+30*(A$<B$)
```

All relational operations are evaluated as true or false and reduced to a value of 1 or 0. Thus, line 340 results in A=1 if the variable X is greater than 10; otherwise, A=0. If you carry this one step further, line 350 results in A=10 if A$>B$ is true, A=20 if A$=B$ is true, and A=30 if A$<B$ is true.

You can append the contents of string B$ to string A$ using this command:

LET A$(0)=B$

Examples

1. When using **LET** as a command, the word "LET" is optional.

```
* X=2
* Y=1495
* LET X=X*Y
```

continued **LET**

2. Determine the length of a filled and an unfilled string.

```
00010 DIM A$(10),B$(20),C$(20)
00020 LET A$="ABC"
00030 PRINT LEN (A$)
00040 LET B$="ABC",A$,FILL$(0)
00050 PRINT B$;
00060 PRINT LEN (B$)
00070 LET C$=542
00080 PRINT C$
00090 END
* RUN
3
ABCABC 20
542
```

3. Be careful with multiple string assignments because expressions are evaluated individually for each variable.

```
* DIM A$(10),B$(10)
* LET A$="AB"
* LET B$="CD"
* LET A$,B$=A$,B$
* PRINT A$ ABCD
* PRINT B$ ABCDAB
```

4. In this example B(3) is set to 7 (not B(7)).

```
00010 LET A=3
00020 LET A,B(A),C=7
```

5. Subscript errors are checked before the **LET** is performed.

```
* DIM I(10)
* A=100
* LET A, I(A)=2
Error 31 - Subscript
```

6. Using **LET** for multiple assignments:

```
10 LET A,B,C=3
```

is equivalent to

```
10 LET C=3 \ B=3 \ A=3
```

7. Using **LET** with relational operators:

```
10 LET A=B=C=0
```

is equivalent to

```
10 IF B=C THEN LET A=0 ELSE A=1
```

LIST

Command

Lists the character contents of a program to a file or a terminal.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{LIST}\left[\left\{\begin{array}{l} \textit{line-number} \\ [\text{TO } \textit{line-number2}] \\ \textit{line-number1} \left[\left\{\begin{array}{c}\text{TO}\\ ,\end{array}\right\} \textit{line-number2}\right]\end{array}\right\}\right] \quad [\text{ ``}\textit{filename}\text{'' }]$$

Arguments

line-number	A line number (when you want only one line listed).
line-number1	The line number of the first line in a range of lines you want listed. If it does not exist, the range begins at the next higher line number.
line-number2	The line number of the last line in a range of lines you want listed. If it does not exist, the range terminates at the next lower line number.
filename	The name of a disk file or device expressed as a string literal in quotation marks. If the filename already exists, you get an error message.

What It Does

LIST writes all of a program or the lines specified to the disk file or device specified by *filename*. If you do not specify a filename, **LIST** displays the program or the lines specified on your terminal. If a program is listed to a disk file, *filename* is put in your directory, and you can retrieve it by using *filename* with the **ENTER** command. If *filename* is a device, the output goes to the device in character format.

How to Use It

To send part or all of a program to a disk file, specify *filename* as the new file. To output part or all of the current program to the line printer, specify its device name as your filename. To specify part or all of the current program in working storage, use the following formats:

LIST	This outputs the entire program starting at the lowest numbered line.
LIST *line-number*	This outputs only that line.
LIST TO *line-number2*	This outputs all lines from the lowest line number through *line-number2*.

continued **LIST**

LIST *line-number1* TO *line-number2* or LIST *line-number1, line-number2*	These both output all lines starting at *line-number1* through *line-number2*. *Line-number2* must be greater than *line-number1*.

When you list only one line, Business BASIC automatically puts it in the edit buffer so that you can use keyboard editing commands on it (all keyboard editing commands are in this manual). When you list a range of lines or all the lines, the last line listed remains in the edit buffer.

Examples

1. List from one line number to another.

```
*LIST 20,40
00020 INPUT BAL
00030 INPUT PAY
00040 LET BAL=BAL-PAY
```

2. List from the beginning.

```
*LIST TO 3O
00010 LET X=10
00020 INPUT BAL
00030 INPUT PAY
```

3. List a particular line to change. ".C" is a keyboard editing command.

```
*LIST
00010 LET X=10
*.C/X=10/X=20
00010 LET X=20
```

4. You can list a Business BASIC program on a line printer. This example shows how to list a program on a DG/RDOS system printer.

```
*LIST TO 30
00010 LET X=20
00020 INPUT BAL
00030 INPUT PAY
*LIST "$LPT"
```

5. List part of a program to a disk file.

```
*LIST 10 TO 100 "JUNK.LS"
```

LISTH

Command

Lists the character contents of a program with header information to a file or a terminal.

AOS/VS	DG/RDOS	UNIX

Formats

$$\text{LISTH}\left[\begin{Bmatrix} \textit{line-number} \\ [\text{TO } \textit{line-number2}] \\ \textit{line-number1}\left[\begin{Bmatrix}\text{TO}\\ ,\end{Bmatrix}\textit{line-number2}\right]\end{Bmatrix}\right] \quad [\text{ “}\textit{filename}\text{” }]$$

Arguments

line-number	A line number (when you want only one line listed).
line-number1	The line number of the first line in a range of lines you want listed. If it does not exist, the range begins at the next higher line number.
line-number2	The line number of the last line in a range of lines you want listed. If it does not exist, the range terminates at the next lower line number.
filename	The name of a disk file or device expressed as a string literal in quotation marks. If the filename already exists, you get an error message.

What It Does

LISTH works like **LIST**, except **LISTH** prints a header containing the user's account name, the date, the time, and the name of the program. If the program is not a saved program, its name will be **SCRATCH**.

How to Use It

See LIST.

Example

LISTH generates a header before displaying the text of the file.

```
*LOAD "test"
*LISTH

:ACCOUNT      DATE        TIME        PROGRAM
:PHIL 6     22-APR-91   14:57:23       TEST

00010  REM THIS IS A TEST
00020  INPUT X
00030  PRINT X^2
00040  END
```

LOAD

Command

Retrieves a saved program.

AOS/VS	DG/RDOS	UNIX

Format

LOAD *"filename"*

Arguments

filename	A filename in quotation marks for a program file created by a previous **SAVE** or **REPLACE**.

What It Does

LOAD clears the current program and data from working storage (like the **NEW** command) and retrieves the program named by *filename*. The new program becomes the current program in your working storage. **LOAD** searches first your directory for *filename*, and then the library directory; if it does not find *filename*, you get an error message. Under UNIX and AOS/VS systems, **LOAD**'s search follows your search path if the program cannot be found in either your current directory or your library directory.

Do not confuse this **LOAD** with the BASIC CLI command **LOAD**, which is used to restore files from a magnetic tape backup.

How to Use It

Use **LOAD** to retrieve a saved or replaced program. Use **LOAD** only as a keyboard mode command. You must specify *filename*. Load only programs that were created by a **SAVE** or **REPLACE**, or Business BASIC utility programs that are already saved. **RUN** *filename* and **CHAIN** *filename* perform a **LOAD** automatically as part of their execution.

LOCK/UNLOCK

Statement and Command

Locks/unlocks files and records.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{LOCK } \textit{identifier,}\left\{\begin{array}{l}\textit{"filename", start,record-size}\\ \textit{logical-file-number,record-number}\end{array}\right\}[\textit{,time}]$$

UNLOCK [*identifier*]

Arguments

identifier	A numeric expression used to identify a specific lock in a program. This is local to a given job. The maximum identifier is 32,700. Negative numbers are not permitted as values for *identifier*. Note: The interpreter does not prevent you from using lock IDs higher than 32,700; however, to avoid conflicts with lock IDs generated by Business BASIC, you should observe this restriction.
filename	A string literal in quotation marks that is the logical or physical name of the file you want locked. All users must use the same filename convention. In other words, all must specify physical files or logical files, or use some other convention established for your system. Do not use logical and physical filenames that are the same.
start	A numeric expression for the logical or physical starting byte number of the lock area: the convention you use (logical or physical) must be consistent with every user.
record-size	A numeric expression for the size (in bytes) of the record in the lock area (i.e., the record you want to lock).
logical-file-number	A numeric expression that evaluates to a file number for a data file of type "D" or "L" that has been previously opened using **LOPEN FILE**.
record-number	The record number of a logical record to be locked in the data file represented by *logical-file-number* that has been opened using **LOPEN FILE**.

LOCK/UNLOCK

time	A numeric variable (optional) containing the amount of time (specified in tenths of a second) to wait for an unlock before abandoning the lock attempt. In a successful lock, time is set to -1; on an unsuccessful lock, an error code is returned in *time*. The maximum value for *time* is 32767.

What It Does

LOCK/UNLOCK is essentially a communications device—it doesn't prevent anyone from accessing the file area. **LOCK** prevents anyone else from using a **LOCK** on the same file area. If all users follow the convention of trying to lock a record before accessing it, **LOCK** prevents another user from using a lock on the same file area by warning the user not to access the record. You should not use **LOCK** for record 0 of a data file that you will access with the **GETREC** statement. **GETREC** provides automatic locking of record 0.

The **LOCK** statement/command locks the number of bytes that contain the record. **LOCK** uses the numeric expression represented by *start* to get the starting byte number and *record-size* as the number of bytes that define an area to be locked. **LOCK** then examines the lock table (a list of locks for the entire system) to see if any other job locked that filename (the first 10 characters of *filename* are compared to filenames in the lock table). If another job also locked *filename*, Business BASIC compares the starting and ending block numbers of the lock area you supplied with the starting and ending block numbers of the lock area already locked. If there is an overlap, your **LOCK** will not be successful.

Your job is suspended either for the value put in *time*, until the other user unlocks that filename, or until an interrupt occurs. In all cases, if your **LOCK** is suspended, the *time* variable returns with a value of 57. If there is no overlap or if *filename* does not exist in the lock table, then your **LOCK** is successful and the filename with your lock area is recorded in the lock table.

UNLOCK removes your filename and lock area from the lock table so that someone else can lock that file. **UNLOCK** with an identifier refers only to the **LOCK** with the same identifier; **UNLOCK** without an identifier unlocks all locked files in your program.

Lock identifiers are unique to each job; for example, you can have 1 through 32,700 identifiers, each of which is unique to your terminal. Another terminal's identifier of the same value does not refer to the identifier of your **LOCK/UNLOCK**. Business BASIC reserves lock identifiers 32,701 through 32,767; however, no error occurs if you use an identifier in this range in a **LOCK** or **UNLOCK** statement. To avoid potential conflicts, though, stay within the 1 to 32,700 range. On AOS/VS and DG/RDOS systems, to find out which locks you have set at your terminal, run the **LOCKS** utility program. Use the **LOCKS** command on UNIX systems to display the current locks on your system. `Error 56 - Attempt to LOCK same area twice` occurs if you try to perform two **LOCK** statements at your terminal with the same identifier without an intervening **UNLOCK**. You can lock the same or overlapping areas as long as you use different lock identifiers. If you supply a time variable, the value 56 is returned in *time*.

On AOS/VS systems, RLS2 must be running and must be on your search path. On UNIX systems, **rlsx** must be running and must be on your search path. `Error 91 - Attempt to issue LOCK/UNLOCK without lock server running` occurs if a lock is attempted when the lock server is not running.

How to Use It

A **LOCK** only prevents another user from locking the same file area, so to control file access using file locking, everyone must agree to the same lock conventions and must also agree to attempt to lock a record before accessing the record. You should use only one type of filename in a **LOCK** statement: physical or logical, preferably logical filenames. Otherwise, if someone tries to lock a file area in a physical file and another user tries to lock the same file area using the logical (subfile) filename instead, both **LOCK** statements are successful, although you only want one of them to be successful.

You must decide which convention to use for *start*: the logical starting byte within a logical file (this can be a subfile) or the physical starting byte in a physical file, whether or not subfiles exist. Otherwise, the same area can be represented differently and locked by different users, which defeats the purpose of **LOCK**.

Use *time* to specify how many tenths of a second a **LOCK** should wait for an **UNLOCK** or interrupt. When that amount of time has elapsed, control returns to your program, and you should check the value in *time*. A 57 means an **UNLOCK** has not yet occurred to unlock the file area you want to lock. Use *time* to avoid "deadly embraces" that occur when you want to lock an area that is already locked by another user and the user wants to lock an area you have already locked. Neither of you can perform an **UNLOCK** while your jobs are suspended.

`Error 56 - Attempt to LOCK same area twice` is issued if you attempt to lock more than one area with the same lock identifier. This error is not returned if you try to lock an area overlapped by an area that is already locked when different lock identifiers are specified. Also, an area that is already locked can be locked again when different lock identifiers are used.

`Error 93 - Maximum number of locks has been exceeded` occurs if no lock buffers are available. `Error 34 - Function argument` occurs if you use an identifier greater than 32767.

LOCK/UNLOCK

continued

Example

```
00010 DIM D$[18],B$[544],KEYID$[10]:For descriptor, buffer, and key.
00020 DIM X$[512],C1[1,3],RECORD$[28]:For C1 array, record, and OPEN
00030 LET X$="DATA1,6,INDEX1,6",FILL$(0):Files to open in mode 6.
.
.
.
00170 INPUT "Type key: ",KEYID$       :Input key.
00180 INPUT "Type record: ",RECORD$   :Input record.
00190 LET F%=0                        :Logical file 0 is DATA1.
00200 LOCK 2,"DATA1",0,C1[0,3]        :Lock record 0 of DATA1.
00210 GOSUB 08400: \GETREC            :Get available record of DATA1.
00220 UNLOCK 2                        :Unlock record 0 of DATA1.
00230 GOSUB 09610:  \POSFL            :Position using R1 from GETREC.
.
.
.
```

LOCKS

Command

Displays information about current file locks.

UNIX

Format

LOCKS

What It Does

The **LOCKS** command displays information about the file locks that have been set using the **LOCK** command.

Each file lock has a unique set of identifiers, including a lock ID. The **LOCKS** command displays the lock ID for each locked file area. The associated filename, process ID, and parameters for the locked area are also displayed. If you are running Business BASIC in DG mode (i.e., you included the -D option when you executed Business BASIC), then your screen clears and the lock information is displayed at the top of the screen. If you are running Business BASIC in non-DG mode, then the lock information is displayed at the right of the screen.

If an **UNLOCK** command is issued while you are viewing the **LOCKS** display, the lock associated with that **UNLOCK** command disappears from the display. (See the **LOCK/UNLOCK** command description.)

How to Use It

Use the **LOCKS** command in keyboard mode.

The column headings in the **LOCKS** display have the following meanings:

S	Status of the lock
	F = free
	I = invalid
	L = active lock
	W = waiting for access to an area that is already locked
	U = unlocked

PID	Process ID of the Business BASIC process that issued the lock
LID	Lock ID; the identifier used in the **LOCK** statement that issued the lock.

LOCKS

continued

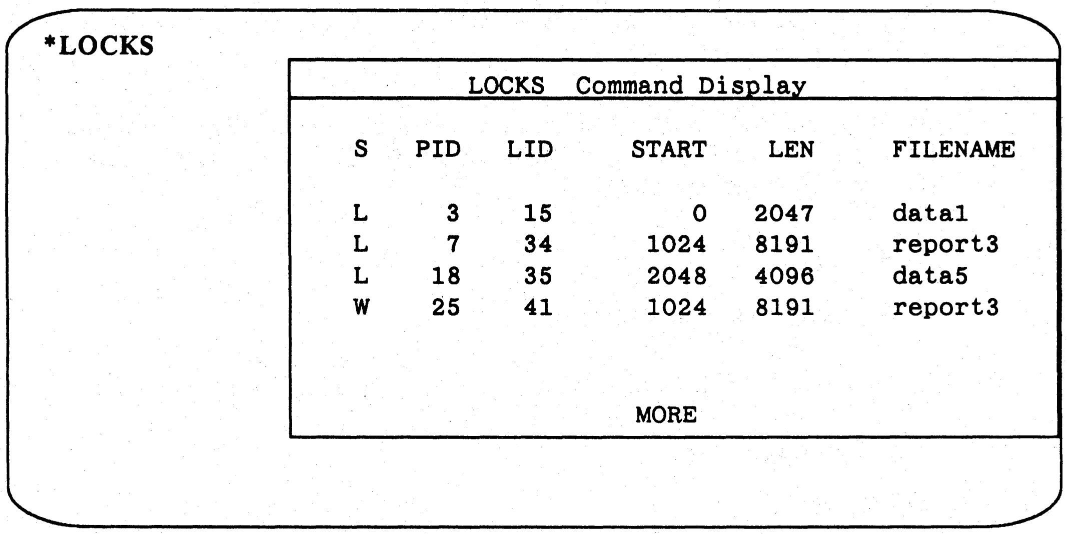
START	Starting byte of the locked area
LEN	Length (in bytes) of the locked area
FILENAME	Name of the file containing the area for which the lock has been issued.

When you are running **LOCKS** in non–DG mode, then if all the output from a **LOCKS** command cannot fit in the display window, the word MORE appears in the bottom border of the display. To scroll down one line in the output, press the New Line key. To scroll down a full screen, press the space bar. To quit the **LOCKS** display process and return to the prompt, press the interrupt key or the **q** key.

NOTE: An I or F in the status column indicates that there is a problem you need to check into.

Example

The **LOCKS** command produces the following display in non–DG mode. All of the processes that are attempting to lock the same area of a file will be displayed after the active record (indicated by an L or U in the status column).The lock ID is the identifier used in the **LOCK** statement/command.



```
*LOCKS
                         LOCKS  Command Display

                  S   PID   LID     START    LEN     FILENAME

                  L     3    15         0   2047     data1
                  L     7    34      1024   8191     report3
                  L    18    35      2048   4096     data5
                  W    25    41      1024   8191     report3


                                     MORE
```

LOPEN FILE

Statement and Command

Opens a logical file.

AOS/VS	DG/RDOS	UNIX

Format

LOPEN [*error-code*,]FILE(*logical-file-number*,*buffer-string*[,*flags*]),"*logical-file-name*"

or

LOPEN [*error-code*,]FILE(*logical-file-number*,*channel*[,*flags*]),"*logical-file-name*",
"*file-type*",*record-length*[,*last-record-number*[,*offset*]]

Arguments

error-code An optional numeric variable that receives any error code generated as a result of the execution of the **LOPEN FILE** statement. This argument must be initialized (i.e., set to zero) before it is used.

logical-file-number A numeric expression yielding the logical file number of a data file opened using **LOPEN FILE**. This number must be greater than or equal to 1.

buffer-string A regular or subscripted string variable dimensioned to (or with a substring length of) at least 512 bytes. It is used as a buffer while **LOPEN** scans the volume label.

flags An optional numeric expression used to signal whether automatic locking of linked (type L) and index (type I) files should occur and whether the physical file should be exclusively opened.

Bits	Mask Value	Effect if Set
0	1	Do not automatically lock this file
1	2	Exclusively open physical file
0 & 1	3	Both exclusive open and no lock

A value of 0 (default when *flags* is missing) signals locking should automatically occur for **GETREC**, **DELREC**, **KADD**, **KDEL**, and **KFIND** statements, and that the file is to be shared. When bit 0 is set, it signals that locking is not needed. When bit 1 is set, the physical file is exclusively opened, which is useful for updates where shared access of any kind is inappropriate.

logical-file-name The name of the logical file in quotation marks.

LOPEN FILE

last-record-number	A numeric expression which represents the last record number that should be used. This argument is required for the second form of the **LOPEN FILE** statement, when the file to be opened is type I, index file. If the file is an index file, you must specify a value that is less than or equal to 65535. If you are opening a file other than type I, this argument is optional and defaults to 16777215.
file-type	A string expression in quotation marks that defines the type of logical file being accessed: "D" for a direct (no automatic record allocation) file, "I" for an index file, and "L" for a linked available record file (automatic record allocation).
channel	The Business BASIC channel on which the physical file containing the logical file being defined is opened.
record-length	A numeric expression that defines the record length. When omitted, this argument defaults to 1. For an index file, this must be 512 or 2048 (AOS/VS and UNIX only).
offset	An optional numeric expression that defines the byte position in the physical file where the logical file begins. When omitted, this argument defaults to 0.

What It Does

The **LOPEN FILE** statement links a logical file name with an identifying logical file number for further I/O referencing.

For the first **LOPEN FILE** format, the volume label file is used by the **LOPEN FILE** statement to determine the characteristics of a logical file being opened. This information is then copied into LFTABL$, which controls the operation of the other logical I/O statements. LFTABL$ is described under "How to Use It."

The second format of the **LOPEN FILE** statement allows you to open a physical file. When using this format, you must first open the physical file. You supply all the information in the **LOPEN FILE** statement instead of reading the volume label file. The arguments you supply are then copied into LFTABL$.

In either format, after the **LOPEN FILE** has been performed, program execution proceeds in the same manner.

NOTE: The use of the *error-code* argument suppresses execution of the system's default error trap or any **ON ERR** condition (which causes the program to halt) and instead returns to *error-code* the same error code as would be supplied by the appropriate **SYS** error function. Therefore, you must check the *error-code* value to determine whether an error has occurred in opening a logical file.

How to Use It

You must use **LOPEN** to define and/or open logical files prior to their use in other logical I/O statements; i.e., **LREAD, LWRITE, GETREC, DELREC, LOCK, UNLOCK, KADD, KDEL, KFIND,** and **KNEXT.**

Use the first syntax for **LOPEN** when opening a database file created by the **LFU LCREATE** utility. Use the second syntax when you have opened a physical file and want the convenience of the logical I/O statements; therefore, you must provide the logical file definition.

Before executing the first **LOPEN** statement, dimension the string variable LFTABL$ and fill it with nulls to a current length of at least 26 times the highest logical file number to be used. LFTABL$ is used to obtain the definition of any logical file referenced by any logical I/O statement. Within LFTABL$ are entries of 26 bytes for each logical file. Each 26-byte entry contains the following fields at the specified relative positions:

Field	Position	Length
Channel number	1	2
Starting byte	3	4
Flags	7	2
Logical filename	9	10
Record length	19	2
Last record number	21	4
File type(D,L,or I)	25	1
Reserved	26	1

The **CLOSE** statement frees the physical channel numbers and closes physical files that were opened using the **LOPEN FILE** statement, but **CLOSE** does not reinitialize LFTABL$ so that another logical file can be opened using the same logical-file-number. However, you can reinitialize the 26 bytes allocated in LFTABL$ for the logical file numbers associated with the physical file that was closed.

The information in LFTABL$ can be accessed to obtain any information desired. This is supported by the subroutine **LFDATA.SL**. You should not change the information in LFTABL$ after its initial **FILL$**. Altering this information can cause logical I/O statements to produce unpredictable results.

Refer to your Business BASIC user's guide for the maximum number of open files allowed on your operating system.

LOPEN FILE

continued

Example

```
00010 DIM T9$[512],LFTABL$[52]      :Dimension required strings.
                                    :LFTABL$ is dimensioned to 26*2
                                    :because highest logical file
                                    :number is 2.
00020 LET LFTABL$=FILL$[0]          :Initialize logical file table.
00030 LOPEN FILE[1,T9$],"EMPDATA"   :Open employee data file.
00040 OPEN FILE[15,0],"TMP"         :Use open to open temporary file
                                    :for index.
00050 LOPEN FILE[2,15],"TMP","I",512,65000      :Open TMP as logical
                                    :index file. Assign values to
                                    :variables required by the
                                    :LINITINDEX.SL subroutine.
00100 GOSUB 07700                   :Call LINITINDEX.SL to initialize
                                    :TMP. Index is now ready to use.
                                    :Use KADD, KNEXT, etc. as
                                    :necessary.
```

LREAD FILE

Statement and Command

Reads a logical record.

AOS/VS	DG/RDOS	UNIX

Format

LREAD FILE [*logical-file-number,record-number,*]*string-variable*

Arguments

logical-file-number	A numeric expression yielding the logical file number of a data file opened using **LOPEN FILE**.
record-number	A numeric expression indicating a record number from 1 to the end of file for a type L file or 0 to the end of file for a type D file.
string-variable	A string variable of sufficient size to receive the record pointed to by *record-number*.

What It Does

The **LREAD FILE** statement causes data in binary format to be read from a file into the string variable listed in the statement.

NOTE: **LREAD FILE** will read logically deleted records without issuing an error message.

How to Use It

LREAD FILE reads from the file the number of bytes required to fill *string-variable* up to the defined logical record size specified in the previous **LOPEN**. A record string that is dimensioned longer than defined by record size will receive only the number of bytes defined by record size. Its current length will be set to this value. An error occurs if an attempt is made to issue an **LREAD** statement using a record string of insufficient length to contain the defined record.

In a random access file, reading a record that has not been written inputs a record of all 0s (null bytes).

LREAD FILE

continued

Example

This example shows an **LREAD** from a logical database file and a physical file.

```
.
.
.
00100 LOPEN FILE [1,B$],"TDATA"       :Open logical database file.
00110 FOR RECNO=1 TO 100              :Scan file.
00120    LREAD FILE [1,RECNO],RECORD$ :Read record from TDATA.
00130    UNPACK "JA20L",RECORD$,STAT%,NAME$,AMOUNT
                                      :Parse into fields.
00140    IF STAT%<=0 THEN GOTO 00800  :Check for deleted record.
.
.
.
00200 OPEN FILE [2,0], "MYFILE"       :Open the physical file.
00210 LOPEN FILE [5,2],"MYFILE","L",40            :Define in LFTABL$
.
.
.
00400 LREAD FILE [5,RNUM],MYREC$      :Read record from MYFILE.
```

LWRITE FILE

Statement and Command

Writes a logical record.

AOS/VS	DG/RDOS	UNIX

Format

LWRITE FILE [*logical-file-number,record-number,*]*string-variable*

Arguments

logical-file-number	A numeric expression yielding the logical file number of a data file opened using **LOPEN FILE**.
record-number	A numeric expression indicating a record number from 1 to the end of file for a type L file or 0 to the end of file for a type D file.
string-variable	A string variable large enough to receive the record pointed to by *record-number*.

What It Does

The **LWRITE FILE** statement outputs data in binary format to the logical record of the file referenced by the *logical-file-number*.

How to Use It

The number of bytes transferred to a file by **LWRITE** is the defined logical record size. If the record string is less than the defined record length, then the output record is padded with nulls. An error occurs if a record string with a current length greater than the logical record length is used.

LWRITE FILE

Example

In this example, a record is locked, read (using **LREAD FILE**), unpacked, updated, repacked, written (using **LWRITE FILE**), and unlocked.

```
.
.
.
00200 LOPEN FILE [1,T$], "DATA5"      :Open DATA5 as a logical
                                      :database file.
.
.
.
00400 LOCK 2,1,RECNO                  :Lock and
00410 LREAD FILE [1,RECNO], DATA5$    :Read record into DATA5$.
00420 UNPACK 00430,DATA5$,STAT%,NAME$,ADR$     :Break into fields.
00430 RFORM JA20A30                   :Format for above UNPACK and
                                      :the PACK below.
                                      :Update fields as needed.
.
.
.
00712 PACK 00430,DATA5$,STAT%,NAME$,ADR$       :Recompose record
                                               :from fields.
00720 LWRITE FILE[1,RECNO],DATA5$     :Rewrite record
00730 UNLOCK 2                        :and unlock it.
```

MAX

Function

Finds the largest expression.

AOS/VS	DG/RDOS	UNIX

Format

MAX*(expression1,expression2)*

Arguments

expression	Numeric expressions or variables that you want to compare. For UNIX only, you can compare more than two expressions; delimit them with commas.

What It Does

MAX returns the largest value of the expressions you specify. See Appendix B for the number of expressions you can specify on your operating system.

How to Use It

Use MAX as a numeric expression wherever numeric expressions are allowed.

Examples

1. Print the maximum of two variables.

```
00010 INPUT X, Y
00020 PRINT MAX(X,Y)
* RUN
?-3,7
7
```

2. Print the maximum of two expressions.

```
00010 DIM X$(100),Y$(100)
00020 INPUT X$,Y$
00030 PRINT MAX(LEN(X$),LEN(Y$))
* RUN
?ABC,ABCDEF
6
```

MIN

Function

Finds the smallest expression.

AOS/VS	DG/RDOS	UNIX

Format

MIN*(expression1,expression2)*

Arguments

expression	Numeric expressions or variables that you want to compare. For UNIX only, you can compare more than two expressions; delimit them with commas.

What It Does

MIN returns the smallest value of the expressions you specify. See Appendix B for the number of expressions you can specify on your operating system.

How to Use It

Use MIN as a numeric expression wherever numeric expressions are allowed.

Example

Determine the smaller of two values and display the value.

```
00010 INPUT X,Y
00015 IF X=Y THEN GOTO 00050
00020 IF MIN(X,Y)=X THEN GOTO 00040
00030 PRINT "Y(";Y;") is the minimum."
00035 STOP
00040 PRINT "X(";X;") is the minimum."
00045 STOP
00050 PRINT "X and Y are equal."
* RUN
?8,5
Y(5) is the minimum.
```

MOD

Function

Finds the remainder after dividing two expressions.

AOS/VS	DG/RDOS	UNIX

Format

MOD(*expression1*,*expression2*)

Arguments

expression1	A numeric expression or variable; the dividend.
expression2	A numeric expression or variable; the divisor.

What It Does

MOD divides *expression1* by *expression2* using this formula:

remainder= ABS(*expression1* – INT(*expression1*/*expression2*) * *expression2*)

For example, **MOD**(17,5) equals 2. If the remainder is 0, then *expression1* is a multiple of *expression2* (and *expression2* is a factor of *expression1*).

How to Use It

Use **MOD** as a numeric expression wherever numeric expressions are allowed.

Examples

1. **MOD** displays the remainder after dividing X by 10.

```
00010 INPUT X
00020 LET Y=10
00030 PRINT  "REMAINDER AFTER DIVIDING X BY 10:";
00040 PRINT MOD(X,Y)
* RUN
? 47
REMAINDER AFTER DIVIDING X BY 10:
7
```

2. For this example, the remainder is 0.

```
00010 X=4
00020 PRINT MOD(X,4)
* RUN
0
```

MOD

continued

3. For this example, the remainder is 0.

```
00010 X=8
00020 PRINT MOD(X,4)
* RUN
0
```

4. In this example, the user inputs values for the expressions to be divided in the **MOD** statement.

```
00010 INPUT X
00020 INPUT Y
00030 PRINT MOD(X,Y)
RUN
?-10
?-3
1
RUN
?10
?-3
1
RUN
?10
?3
1
```

MSG

Command

Sends a message to another terminal or process.

RDOS

Format

MSG, [*terminal,*] "*message*"

Arguments

terminal	The port number of the terminal to receive *message*. This can be a number, numeric expression or variable. You can find out which terminals are logged on and what their numbers are by using the Business BASIC CLI command **STAT**. If you do not specify *terminal*, the message is sent to the OP user with the lowest job number. If no OP user is logged on, `Error 51 - User not on system` is displayed.
message	The text of the message you want to send, expressed as either a string literal in quotation marks or a predefined string variable.

What It Does

MSG sends *message* to the terminal specified or to the OP user with the lowest job number if no terminal is specified. The receiving terminal displays the username of the sender along with the message, unless the receiving terminal set a no-message flag with **STMA 6,4**.

How to Use It

Use **MSG** as a keyboard command only. First, find out which terminals are logged on and their numbers by using the utility **STAT**. Your message may be a string literal in quotation marks or a previously dimensioned and assigned string variable.

Examples

***MSG,2,"WHAT ABOUT LUNCH?"**

This message would appear at terminal 2.

```
FROM 4:AATLB6 WHAT ABOUT LUNCH?
```

AATLB6 is the username for the message-sender, and 4 is AATLB6's terminal number. AATLB is the user that signed on at terminal 4; the appended "6" means that terminal 4 is a type 6 terminal.

***MSG,4,"YOUR TREAT? OK!"**

At terminal 4, the reply would come back from account ABSCR:

```
FROM 2:ABSCR6 YOUR TREAT?  OK!
```

MSG *continued*

```
*MSG, "LET'S GO EVERYBODY!"
```

This message would be received by the OP user with the lowest job number. Presumably the system operator would gather everyone for lunch.

MTDIO

Statement and Command

Directs I/O control of magnetic tape and cassette.

AOS/VS	DG/RDOS	UNIX

Formats

Format 1: Read from or write to a tape.

MTDIO *channel,function,string-variable,error-code*

Format 2: Skip records forward or backward.

MTDIO *channel,function,records,record-count,error-code*

Format 3: Rewind a tape.

MTDIO *channel,function,error-code*

Arguments

channel — A numeric expression for the channel number of a file opened in mode 7 (magnetic tape I/O).

function — A code number indicating one of the following functions:

Code	Function
1	Read a tape record into *string-variable* (format 1).
2	Rewind the tape to the beginning (format 3).
3	Not used.
4	Skip *records* number of records in tape, moving forward (format 2).
5	Skip *records* number of records in tape, moving backward (format 2).
6	Write a tape record from *string-variable* (format 1).
7	Write an end-of-file mark on the tape (format 3). (RDOS and UNIX only)
8	Read the tape drive status into *error-code* (format 3).

The default parity is odd. Make your function code negative if the tape requires even parity, but ask your system manager first.

Note: On UNIX systems, there is no difference between negative and positive codes.

string-variable — A string variable or subscripted string variable that contains data bound for output to tape or receives data input from tape. You must have already dimensioned *string-variable*.

MTDIO

continued

error-code	The numeric variable that receives the tape status or error code as bit flags in the right-most two bytes of the variable. This variable receives one of the error status codes listed in Figure 1–7. If some other error occurs, then a Business BASIC error is generated.
records	A numeric expression for the number of records to skip; it must be in the range of 0 to 4095. (If you specify 0, then 4096 records are skipped; otherwise, the number of records skipped equals the value you specify.)
record-count	A numeric variable that receives the count of the actual number of records skipped.

What It Does

MTDIO reads from and writes to magnetic tapes. You can read data, write data, skip records, and rewind the tape using **MTDIO** statements in your program or **MTDIO** commands at your terminal. **MTDIO** can handle any tape format, since it reads in records whose size you determine by dimensioning *string-variable*. Under AOS, AOS/VS, and RDOS systems, your record size can range from 2 to 8192 bytes. Under DG/RDOS systems only, this value can range from 2 to 16384 bytes. Under UNIX systems, the maximum record size is 32768.

On DG/RDOS systems only, the **.MTDRW** system call allows support for 16-Kbyte records. Use this feature by dimensioning the data string in the **MTDIO** read/write command to a value greater than 8192 and less than or equal to 16384.

UNIX, AOS/VS, and DG/RDOS tape formats require an even number of bytes because all tape transfers are word (two-byte) transfers. Also, on UNIX systems, a 150–MB cartridge tape only supports records that are a multiple of 512 bytes. If your tape is not in the proper format, you may get transfers with an odd number of bytes; this condition is indicated in *error-code*.

When **MTDIO** is reading from a tape and you have dimensioned *string-variable* to be larger than the number of bytes actually read, *string-variable* is truncated to the actual number of bytes found. If you have dimensioned *string-variable* to be less than the number of bytes in the tape record, *string-variable* is filled to capacity, and you lose the excess bytes (error-code will not indicate this).

When you are writing a tape record, the size of the record equals the dimensioned size of *string-variable* or the specified length of *string-variable* if *string-variable* is a substring, regardless of the current length of *string-variable*. Under DG/RDOS, dimensioning the string variable to a value greater than 8192 and less than or equal to 16384 bytes causes 16-Kbyte records to be used. When you are skipping tape records, a value of 0 for *records* causes the maximum of 4096 records to be skipped either forward or backward. Otherwise, a value from 1 to 4095 for *records* causes the specified number of records to be skipped, forward or backward. If the number of records you want to skip is greater than the number remaining in the file (moving forward), the tape stops when it encounters the end-of-file marker. You can test for an end of file by checking the status word returned in *error-code*.

After an **MTDIO** execution, *error-code* contains the status word for the tape drive unless an error occurs. An error returns the appropriate error code in *error-code*. End of file, end of tape, parity, and other DG/RDOS tape drive errors can only be determined by checking bits of the status word. These errors are not returned as Business BASIC errors. The status values as well as what each bit value signifies are shown in Figure 1-7.

VALUE	BIT POSITION																SIGNIFIES
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
32768	1	●	0	●	0	●	●	●	●	0	●	0	0	0	●	0	Error. (Bits 1, 3, 5, 6, 7, 8, 10 and 14 are the error bits; at least one is set.)
16384	●	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Data late; also sets bit 0 (RDOS and DG/RDOS only).
8192	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Tape rewinding (RDOS and DG/RDOS only).
4096	●	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Illegal command (also sets bit 0).
2048	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	High density (always set for cassettes & UNIX systems).
1024	●	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	Parity error (also sets bit 0).
512	●	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	End of tape (also sets bit 0).
256	●	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	End of file (also sets bit 0).
128	●	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	Beginning of tape (also sets bit 0).
64	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	Nine track tape (always set for cassettes & UNIX systems).
32	●	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	Bad tape (also sets bit 0).
16	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	Send clock (RDOS and DG/RDOS only).
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	First character (RDOS and DG/RDOS only).
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Write protected. Cannot be written to.
2	●	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	Odd character (odd byte) that also sets bit 0.
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	UNIT READY.

A "1" in a bit signifies that the bit is set, and the "value" is the value you use (with AND) to check if only that bit is set. Normally you only check one bit at a time, but if you want to check more than one bit, add the values together and use the total value to AND with er. A " ● " in a bit signifies that the bit in this word is set (signified by a "1"), at least one of the " ● " bits is also set (if one " ● " bit, it is set).

Figure 1-7 MTDIO Status Values

MTDIO

continued

How to Use It

To use **MTDIO**, first open the desired tape device in mode 7 on *channel*. When reading a tape with **MTDIO**, your string variable should be of sufficient size to contain the largest expected record. You will lose any bytes that do not fit. You must request an even number of bytes. If you know your tape ends on an odd byte, you should read in the number of bytes plus one to make it even and then discard the last byte (it will be a duplicate of the one before it).

Determine any error or tape drive status condition (like end of file, end of tape, or a parity error) by checking bits of the status word (see Figure 1-7). The variable *error-code* receives the status word, so you can use the **AND** function (**AND** *error-code)* with the value of the bit flag you want to check. To check more than one bit flag at once, add the values of the bit flags together and use the total value in the **AND** function with *error-code*. If you are reading a tape whose records were written by the DG/RDOS CLI **XFER** command or the Business BASIC CLI **XFER** command (**XFER** writes 512-byte blocks), you must dimension *string-variable* to 512 bytes for each **MTDIO** read. When reading into a substring, the substring must begin on an odd byte (1,3,5...) and must be an even number of bytes long.

When you write a tape record from *string-variable*, *string-variable* must be an even number of bytes in length. If *string-variable* is a substring, it must start on an odd byte (1,3,5...) and be an even number of bytes long. Remember to check for an end-of-tape condition in the status word in *error-code* when writing long tape files; otherwise, the tape could run off the reel. On DG/RDOS systems, after writing all your records to the tape, you should write an end-of-file mark using function 7 to prevent attempts to read beyond the end of file when the tape is used later. On AOS/VS systems, you must close the tape file to write an end–of–file mark. Business BASIC expects two end-of-file marks after the last file on a tape.

When skipping forward or backward to records on a tape, specify the records as a number from 1 to 4095. If you specify 0, the maximum number of records (4096) are skipped. Use 0 in *records* with a forward skip to get to the end of file quickly (the tape stops at the end-of-file marker), and use 0 in *records* with a backward skip to return to the exact beginning of the tape file (it stops at the beginning-of-tape marker). If there are multiple tape files, then the only time you go to the beginning-of-tape marker is when you space backward while in tape file 1. If you space backward while in any other tape file, it stops after crossing an end-of-file marker. Thus, to get to the beginning-of-tape marker, you need to use rewind.

When you rewind a tape in DG/RDOS Business BASIC, use a loop to check the status word in *error-code* to see if the tape is still rewinding or if the unit is ready. Tape drives are not as fast as **MTDIO**, and you do not want to try reading from or writing to a tape that is still rewinding.

NOTE: Even if the rewinding bit is never set on your tape drive, you should still check the ready bit.

In UNIX and AOS/VS Business BASIC, you cannot check the tape status as the tape rewinds, because the rewinding is handled by the operating system.

Example

This example uses **MTDIO** on DG/RDOS and UNIX with a magnetic tape.

Under AOS/VS Business BASIC, you cannot write EOF marks (as in lines 560 and 570). Also, you cannot check the tape status when rewinding the tape (as in lines 590 and 620). EOF marks and tape status are handled by AOS/VS.

```
00010 DIM RECORD$(80)        :Tape records are 80 bytes long.
00020 OPEN FILE (1,7),"MTO:0"        :Tape device MTO,file 0.
00025 LET ER%=0              :Initialize ER% to receive status.
00030 MTDIO 1,1,RECORD$,ER% :Read tape record into RECORD$,and tape
                             :status into ER%.
00040 IF AND(ER%,256) THEN GOTO 00300 :Test for an end-of-file
                             :process record; go read next record.
. . .
00300 MTDIO 1,5,0,NUMSKP,ER%:back up 4096 records or to beginning
                             :of file.
00310 INPUT "REC NUMBER OF NEW REC: ",NUM
00320 MTDIO 1,4,NUM-1,NUMSKP,ER% :go forward to record NUM.
00330 INPUT "TYPE NEW RECORD: ",RECORD$
00340 MTDIO 1,6,RECORD$,ER% :write new record.
. . .
00500 REM -- WRITE NEW LAST RECORD AND NEW EOF MARK
00510 INPUT "TYPE LAST NEW RECORD: ",RECORD$
00520 MTDIO 1,4,0,ER% :Go forward to the end-of-file.
00530 MTDIO 1,6,RECORD$,ER% :Overwrite end-of-file mark with new
                             :record.
00540 MTDIO 1,8,ER%          :Read tape drive status only into ER%
00550 IF AND(ER%,32768) THEN GOTO 01000 :if error, go to line 1000.
00560 MTDIO 1,7,ER%          :Write an end-of-file mark on the tape.
00570 MTDIO 1,7,ER%          :Write another end-of-file mark.
00580 MTDIO 1,2,ER%          :Rewind the tape.
00590 IF NOT(AND(ER%,8192)) THEN GOTO 00600     :Set up loop to check
                                                :if tape is still
                                                :rewinding.
00592 DELAY 50
00594 GOTO 00580
00600 CLOSE FILE (1) :Tape rewound, close the file
00610 STOP
00620 GOTO 00590             :This loop will continue until tape is
                             :rewound.
. . .
01000 IF AND(ER%,16384) THEN PRINT "DATA LATE"
01010 IF AND(ER%,512) THEN PRINT "END OF TAPE"
. . .
```

NEW

Statement and Command

Clears your working storage.

AOS/VS	DG/RDOS	UNIX

Format

NEW

What It Does

NEW clears all currently loaded statements and variables from working storage, leaving the working storage empty for new programs. It also closes any open file channels, and unlocks any locked files. **NEW** does not clear the common area.

The **NEW** instruction causes all open INFOS II channels to be closed.

How to Use It

Use **NEW** as a program statement or as a keyboard command. **NEW** clears your working storage so that you can load or enter a program, or you can prevent someone from listing the contents of the current program in working storage. You can make **NEW** a program statement that executes when the program finishes, preventing anyone from listing the program after it has run. You can also use **NEW** with **ON ERR** and **ON IKEY** to prevent the unauthorized listing of a program when an error or interrupt occurs.

Examples

1. **NEW** as a keyboard command.

   ```
   * NEW
   ```

2. **NEW** prevents the listing of a program on an interrupt.

   ```
   00010 ON IKEY THEN NEW
   ```

3. You can use **NEW** to prevent anyone from listing a program after it has run. For example:

   ```
   00900 IF DONE$="NO" THEN GOTO 00500
   00950 NEW
   ```

NEXT

Statement

Defines the end of a program loop and increments or decrements loop control variables.

AOS/VS	DG/RDOS	UNIX

Format

NEXT *control-variable*

Arguments

control-variable	A nonsubscripted numeric variable that controls the loop by being increased or decreased. It must be the same variable used in the **FOR** statement.

What It Does

The **NEXT** statement must always be the last statement of a **FOR...NEXT** loop. When Business BASIC reaches the **NEXT** statement, it increases or decreases the *control-variable*, according to the **STEP** *expression* in the **FOR** statement. Control then passes back to the beginning of the loop where the *control-variable* is tested.

See **FOR...NEXT** for more information.

NOT

Operator

Evaluates an expression using the Boolean logical NOT.

AOS/VS	DG/RDOS	UNIX

Format

NOT *expression*

Arguments

expression	Numeric expression or variable. If *expression* is 0, then **NOT** *expression* is true; if *expression* is not equal to 0, then **NOT** *expression* is false.

What It Does

NOT gives an expression the opposite Boolean value. **NOT** can be combined with **AND** and **OR**. **NOT** modifies the expression it immediately precedes.

How to Use It

NOT can be used any place a numeric expression is valid.

Before the Boolean Logic operator is executed, the expressions are evaluated and reduced to zero or one. Zero is false. Thus, **NOT** *zero* is true. Any non-zero value is true, so **NOT** *non-zero* is false.

The precedence of all operators is given below.

```
highest          ^ (exponential)
  .              unary +, unary -, NOT
  .              *, /
  .              +, -
  .              <>, >, >=, =, <=, <
  .              AND
 lowest          OR
```

Examples

1. When A is true (not equal to zero) and B is false (equal to zero), begin execution at line 200.

```
00010 IF A AND NOT B GOSUB 00200
```

2. When both B and C are false (equal 0), begin execution on line 100.

```
00010 IF NOT (B OR C) GOTO 00100
```

ON ERR

Statement

Traps an error in your program.

AOS/VS	DG/RDOS	UNIX

Format

ON ERR THEN { *statement* | INT }

Arguments

statement	Any Business BASIC statement except **FOR...NEXT**, **DATA**, **MSG**, **END**, **REM**, and **DEF**.
INT	The word **INT** resets to normal error handling.

What It Does

Normally when an error occurs during program execution, the program halts, Business BASIC prints the error message, and your terminal returns to keyboard mode. If Business BASIC encounters an **ON ERR** statement first, that statement tells Business BASIC what to do if an error occurs. If an error occurs after the **ON ERR** statement, the statement following **THEN** executes.

Use **ON ERR** with the **SYS** error functions and **ERM$**, **AERM$**, and **UERM$**. See **SYS** for more information.

ON ERR THEN INT resets the error trap to normal error handling by Business BASIC.

How to Use It

Use **ON ERR** only as a program statement. Since **ON ERR** traps only errors that occur after it executes, you may want to place an **ON ERR** statement at the beginning of your program. If you place the **ON ERR** statement anywhere else in your program, it traps all errors that occur from that point on.

Examples

1. If an error occurs go to subroutine at 200.

```
00010 ON ERR THEN GOSUB 00200
```

2. If an error occurs, stop the program but don't print the error message.

```
00010 ON ERR THEN STOP
```

ON ERR

continued

3. If an error occurs, go to the routine at 500, which dimensions A$, assigns the error message to A$, and prints the error message.

```
00010 ON ERR THEN GOTO 00500
. . .
00500 DIM A$(64)
00510 LET A$=ERM$(SYS(7))
00520 PRINT A$
```

ON...GOTO and ON...GOSUB

Statement

Conditionally transfers control to other statements.

AOS/VS	DG/RDOS	UNIX

Format

ON *expression* [THEN]$\left[\begin{Bmatrix}\text{GOTO}\\ \text{GOSUB}\end{Bmatrix}\right]$ *line-number1* [,*line-number2* ...]

Arguments

expression	A numeric expression that evaluates to an integer. If *expression* does not evaluate to an integer, it is truncated automatically as if you had used the **INT** (Integer) function.
line-number	A valid line number for a statement or subroutine in the program you want to execute. See Appendix B for the range of line numbers allowed on your operating system.

What It Does

The line to which you transfer control depends on the computed value of *expression* when the statement is executed.

ON...THEN...GOTO and **ON...THEN** do the same thing: *expression* evaluates to an integer and control passes to the first line number in the **ON** statement if the expression evaluates to 1. Control passes to the second line number if the expression evaluates to 2, or the third line number if the expression evaluates to 3, or the tenth line number if the expression evaluates to 10, etc. **ON...GOSUB** works the same way except that the line numbers refer to subroutines with **RETURN** statements that return control to the statement immediately following the **ON...GOSUB** statement.

If the value of *expression* is greater than the number of lines you specify in the **ON** statement or less than or equal to 0, then control passes over the **ON** statement to the next statement.

How to Use It

Use **ON...GOTO** or **ON...GOSUB** to transfer control to one of several lines in a program. You can place these transfer points anywhere in a program to redirect program execution. Use **ON...GOTO** (or **ON...THEN**) to transfer control to another line and use **ON...GOSUB** to transfer control to a subroutine that returns you to the line after the **ON...GOSUB**.

If you do not type either **GOTO** or **GOSUB**, you get the default option—**GOTO**. You can also enter **ON** *expression* **GOTO** or **ON** *expression* **GOSUB** without entering **THEN**; however, Business BASIC inserts any missing **THEN** statements when the program is listed.

ON...GOTO/GOSUB

continued

Examples

1. In this example, if M=l, control passes to the subroutine at 100; if M=2 control passes to 200. If M>4, it falls to the next **ON...GOSUB** statement; if M>8, it falls to the last **ON...GOSUB** statement.

```
00010 INPUT "TYPEMONTH (1-12):",M
00020 ON M THEN GOSUB 00100,00200,00300,00400
00025 ON M-4 THEN GOSUB 00500,00600,00700,00800
00030 ON M-8 THEN GOSUB 00900,01000,01100,01200
00040 REM--ROUTINES RETURN TO HERE
. . .
00100 PRINT "JANUARY"
00110 RETURN
. . .
00200 PRINT "FEBRUARY"
00210 RETURN
. . .
00300 PRINT "MARCH"
00310 RETURN
. . .
```

2. In this example, if X=1, control passes to line 100; if X=2 control passes to 200. If X>5, it falls to the next **ON...THEN...GOTO** statement.

```
00005 RANDOMIZE
00010 LET X=RND(11)
00020 ON X THEN GOTO 00100,00200,00300,00400,00500
00030 ON X-5 THEN GOTO 00600,00700,00800,00900,01000
00040 PRINT "X=0"
00050 STOP
00100 PRINT "X=1"
00120 STOP
00200 PRINT "X=2"
00210 STOP
00300 PRINT "X=3"
00310 STOP
00400 PRINT "X=4"
00410 STOP
. . .
```

3. These two lines function identically:

```
00100 IF A>B THEN GOTO 02000
```

```
00100 ON A>B THEN GOTO 02000
```

When A>B, execution resumes at line 2000; otherwise, the next line after 100 is executed.

ON IKEY

Statement

Traps an interrupt in your program.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{ON IKEY THEN}\left\{\begin{array}{l}\textit{statement}\\ \text{INT}\end{array}\right\}$$

Arguments

statement	Any Business BASIC statement except **FOR...NEXT**, **DATA**, **DO...WHILE/UNTIL**, **END**, **END LOOP**, **REM**, and **DEF**.
INT	The word **INT** resets to normal error handling.

What It Does

The interrupt key is usually Escape, Line Feed, or a key you or your system manager set using **STMA 4,6** or the **TERM** utility program.

Normally, when an interrupt occurs during program execution, the program halts, **SYS(26)** is set to 1, and your terminal returns to keyboard mode. Under normal conditions, an **ON IKEY** statement handles any interrupt that occurs after Business BASIC encounters it. When an interrupt occurs after an **ON IKEY** statement, Business BASIC executes the statement following **THEN**. If the statement is a **GOSUB**, control passes to the subroutine and **RETURN** returns control to the statement immediately following the statement where the interrupt occurred. **ON IKEY THEN INT** resets the error trap to normal error handling by Business BASIC. The values for IKEY are in your Business BASIC user's guide.

How to Use It

Use **ON IKEY** as a program statement. Since **ON IKEY** only traps interrupts that occur after it has been executed, you may want to place an **ON IKEY** statement at the beginning of your program. You can restore normal handling of interrupts with an **ON IKEY THEN INT** statement. If you disable interrupts with an **STMA 6,5**, then an interrupt will not be trapped in an **ON IKEY** nor will it stop the program. You can then test **SYS(26)**, the IKEY indicator, for a value of 1; if **SYS(26)** is 1 you can re-enable interrupts with an **STMA 7,5**, and the interrupt will then be trapped in the **ON IKEY** statement.

Examples

1. If an interrupt occurs, this statement clears working storage.

```
00005 ON IKEY THEN NEW
```

ON IKEY

continued

2. While this program is running, the user presses the interrupt key. Nothing happens while file I/O is executing because line 20 sets the no-IKEY flag. When Business BASIC reaches line 400, it tests **SYS(26)** to see if an interrupt has occurred. Line 400 resets the no-IKEY flag to allow an interrupt to occur and prints the message from line 10.

```
00010 ON IKEY THEN PRINT "IKEY OCCURRED"
00020 STMA 6,5
. . .
(file I/O statements might go here)
. . .
(user presses the interrupt key)
. . .
01000 IF SYS(26)=1 THEN STMA 7,5
. . .
02000 STOP
```

```
* RUN
(press interrupt key)
IKEY OCCURRED


STOP AT 02000
```

ON IKEY

Statement

Traps an interrupt in your program.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{ON IKEY THEN}\left\{\begin{array}{l}\textit{statement}\\ \text{INT}\end{array}\right\}$$

Arguments

statement	Any Business BASIC statement except **FOR...NEXT**, **DATA**, **DO...WHILE/UNTIL**, **END**, **END LOOP**, **REM**, and **DEF**.
INT	The word **INT** resets to normal error handling.

What It Does

The interrupt key is usually Escape, Line Feed, or a key you or your system manager set using **STMA 4,6** or the **TERM** utility program.

Normally, when an interrupt occurs during program execution, the program halts, **SYS(26)** is set to 1, and your terminal returns to keyboard mode. Under normal conditions, an **ON IKEY** statement handles any interrupt that occurs after Business BASIC encounters it. When an interrupt occurs after an **ON IKEY** statement, Business BASIC executes the statement following **THEN**. If the statement is a **GOSUB**, control passes to the subroutine and **RETURN** returns control to the statement immediately following the statement where the interrupt occurred. **ON IKEY THEN INT** resets the error trap to normal error handling by Business BASIC. The values for IKEY are in your Business BASIC user's guide.

How to Use It

Use **ON IKEY** as a program statement. Since **ON IKEY** only traps interrupts that occur after it has been executed, you may want to place an **ON IKEY** statement at the beginning of your program. You can restore normal handling of interrupts with an **ON IKEY THEN INT** statement. If you disable interrupts with an **STMA 6,5**, then an interrupt will not be trapped in an **ON IKEY** nor will it stop the program. You can then test **SYS(26)**, the IKEY indicator, for a value of 1; if **SYS(26)** is 1 you can re-enable interrupts with an **STMA 7,5**, and the interrupt will then be trapped in the **ON IKEY** statement.

Examples

1. If an interrupt occurs, this statement clears working storage.

```
00005 ON IKEY THEN NEW
```

ON IKEY *continued*

2. While this program is running, the user presses the interrupt key. Nothing happens while file I/O is executing because line 20 sets the no-IKEY flag. When Business BASIC reaches line 400, it tests **SYS(26)** to see if an interrupt has occurred. Line 400 resets the no-IKEY flag to allow an interrupt to occur and prints the message from line 10.

```
00010 ON IKEY THEN PRINT "IKEY OCCURRED"
00020 STMA 6,5
. . .
(file I/O statements might go here)
. . .
(user presses the interrupt key)
. . .
01000 IF SYS(26)=1 THEN STMA 7,5
. . .
02000 STOP
```

```
* RUN
(press interrupt key)
IKEY OCCURRED


STOP AT 02000
```

OPEN FILE

Statement and Command

Opens a file or device in an access mode and gives it a channel number.

AOS/VS	DG/RDOS	UNIX

Format

OPEN [*error-code*,] FILE(*channel*[,*mode*]), $\left\{ \begin{array}{l} \text{"}\textit{filename}\text{"} \\ \textit{string-variable} \end{array} \right\}$

Arguments

error-code	An optional variable that receives any error code generated as a result of the execution of the **OPEN FILE** statement. If a Business BASIC error occurs, *error–code* returns a positive number. If *error–code* returns a negative number, then it represents a DG/RDOS error code. Use the **ERM$** function to retrieve the error message associated with the error returned. A 0 in *error–code* means no error occurred. You must assign error–code a value before using it.
channel	A numeric expression for the channel you use to refer to the file in subsequent file I/O. See Appendix B for the range of channel numbers allowed on your operating system. Channel 16 is the console in all cases.
mode	A numeric expression for the access mode you assign to the file in the range 0 through 7. *mode* is an optional argument; it defaults to 0.
filename	A string literal in quotation marks for the pathname or filename of a disk file or device; a device name (or link) must already exist.
string-variable	A string variable that represents *filename*.

What It Does

OPEN FILE assigns the channel and mode of access you specify to *filename* or *string-variable*. You can then use *channel* in a file I/O statement to refer to this file or device.

NOTE: To find the next available channel number on an AOS/VS or UNIX system, use **SYS(34)**.

OPEN FILE

The types of open are:

Exclusive	For AOS/VS and UNIX systems, this mode means exclusive read and write access. For DG/RDOS systems, it means exclusive write access but non-exclusive read access.
Non-exclusive (shared)	Multiple users can access the file. This does not mean users are given shared page I/O.
Shared Page	You are opening a file for shared page (memory resident) access.

The types of access are:

Random	You can use **POSITION FILE** and a byte offset argument.
Sequential	You cannot use **POSITION FILE** and a byte offset argument.

The access modes you can use with each type of open in Business BASIC on AOS/VS and UNIX systems are described in Table 1-1.

The access modes you can use with each type of open in DG/RDOS Business BASIC are described in Table 1-2.

NOTE: For DG/RDOS systems, modes 0, 1, 2, 6, and 7 refer only to write access. Since exclusive access only applies to write access, you can use modes 3 and 4 to read a file that has been opened exclusively by another user.

Use of the *error-code* argument suppresses execution of Business BASIC's default error trap or any **ON ERR** condition which would cause the program to halt and instead returns to *error-code* the same error code as would have been supplied by the **SYS(7)** function. Therefore, you must check the *error-code* value to determine whether an error has occurred in opening a file.

Mode 0

Opens disk files in random mode only. This opens the file exclusively for you (you do not need to lock it) and allows for both input and output. If *filename* is not found in your directory or in a directory in your search path, Business BASIC creates *filename* in your directory as a random file. If *filename* is a link to a file that does not exist, the resolution file is created as a random file under DG/RDOS. Under AOS/VS, if the link file is in the working directory, the error message `File already exists` is returned.

NOTE: Mode 1 and mode 2 are the only modes appropriate for Business BASIC spooler output, i.e., the opening of "?*queuename*". Other modes of open cause an error when the "?" queue indicator is present in the name. Note that the spooler is used only for DG/RDOS systems and that when used to open the spooler, modes 1 and 2 operate identically.

Table 1-1 AOS/VS and UNIX Access Modes

Mode	Type of Open	Input/Output	Type of Access	Rule
0	Exclusive	Input/Output	Random	Create if file does not exist. Links to non-existent files are not resolved on this type of open.
1	Exclusive	Output only	Sequential	Delete file; then create file.
2	Exclusive	Output only	Sequential	Append to existing file or create file.
3	Non-exclusive	Input only	Random	File must exist.
4	Shared page	Input only	Random	File must exist. User must have write access to the file. For AOS/VS, file must have an element size of 4. On UNIX systems, a member of a group can use mode 4 if the group has write access to the file.
5	Shared page	Input/Output	Random	File must exist. User must have write access to the file. For AOS/VS, file must have an element size of 4.
6	Exclusive	Input/Output	Random	File must exist.
7	Exclusive use of magnetic or cassette tape	Input/Output	MTDIO access	Tape drive must exist. Tape file can be created, deleted, or appended to.

Mode 1

Opens either a disk file or an output device for exclusive output. Mode 1 permits only **WRITE FILE**, **PRINT FILE**, and **BLOCK WRITE FILE** statements. If *filename* exists in your directory, Business BASIC deletes it. In any case, Business BASIC creates *filename* in your directory as a sequential file. **OPEN FILE** with mode 1 does not search directories in your search path except when used for the Business BASIC spooler, in which case all files are in $SPL.

OPEN FILE

continued

Table 1-2 DG/RDOS Access Modes

Mode	Type of Open	Input/Output	Type of Access	Rule
0	Exclusive	Input/Output	Random	Create if file does not exist.
1	Exclusive	Output only	Sequential	Delete file; then create file.
2	Exclusive	Output only	Sequential	Append to existing file or create file.
3	Non-exclusive (shared)	Input only	Sequential	File must exist.
4	Non-exclusive (shared)	Input only	Random	File must exist.
5	Non-exclusive (shared)	Input/Output	Random	File must exist.
6	Exclusive	Input/Output	Random	File must exist.
7	Exclusive use of magnetic or cassette tape	Input/Output	MTDIO access	Tape must exist. Tape file can be created, deleted, or appended to.

Mode 2

Opens any output device or file in exclusive append mode. **OPEN FILE** positions the file pointer to the end of the current file so that you can append to it. If *filename* does not exist in your directory, or in a directory in your search path, *filename* is created as a sequential file. Mode 2 allows only sequential output.

NOTE: **BLOCK WRITE FILE** never appends data to the end of a file. Using mode 2 does not give you this feature, because you must specify the block number on the **BLOCK WRITE FILE** in order to write to a block other than block 0.

Mode 3

Opens any input device or file in non-exclusive (shared) mode. *filename* must exist in your directory or in a directory on your search path. With this mode, you can use **READ FILE, INPUT FILE,** or **BLOCK READ FILE** statements. Shared mode means that another user can open *filename* at the same time, so use file locking.

In AOS/VS and UNIX systems, mode 3 also allows random input and use of the **POSITION FILE** statement. This mode does not require that the file have an element size of 4 on AOS/VS or write-access privileges.

NOTE: Programs using mode 3 to read a file that was being updated by another program using mode 5 (shared I/O) cannot be guaranteed to read current record information.

On DG/RDOS systems, mode 3 can access exclusively opened files. **POSITION FILE** cannot be used in mode 3 on DG/RDOS systems.

Mode 4

Opens only disk files for shared random input. Random access allows you to position to a byte or a record within a random file. *filename* must exist in your directory or in a directory in your search path. Only random input is allowed.

On DG/RDOS systems, mode 4 can access exclusively opened files.

On AOS/VS and UNIX systems, mode 4 requires that the file have write-access privileges. On AOS/VS systems, the file must also have an element size of 4. These are operating system requirements for files opened in shared mode. This gives users an input-only mode that can read current record information in a file being accessed and modified by another program in mode 5.

Mode 5

Opens only disk files for shared random access. In Mode 5 you can input data from and output data to a random file. *filename* must exist in your directory or in a directory in your search path.

On AOS/VS and UNIX systems, mode 5 requires that the file have write-access privileges. On AOS/VS systems, the file must also have an element size of 4. These are operating system requirements for files opened in shared mode.

Mode 6

Opens only disk files for exclusive random access. This mode allows you to input data from and output data to a random file, and you have exclusive use of the file. *filename* must exist in your directory or in a directory in your search path.

Mode 7

Opens only tape and cassette devices for direct tape access. The tape device is exclusively opened for use with **MTDIO**.

How to Use It

In multiuser environments, be careful when using exclusive access, especially to devices. Business BASIC allows you to open any file in any mode, but some access modes are useless for certain files. If you open a file in mode 1, 2, 3 (DG/RDOS only), or 7, you cannot use the **POSITION FILE** statement. You can open remote files through XODIAC™ in modes 0, 1, 2, 3, and 6.

OPEN FILE

continued

Examples

1. This statement opens the line printer for output (DG/RDOS).

```
00010 OPEN FILE(0,1), "$LPT"
```

2. This statement opens the spooler queue for output (DG/RDOS).

```
00010 OPEN FILE(15,1), "?LPT"
```

3. This opens file **EMPDATA** on channel 4 for exclusive I/O.

   *** OPEN FILE(4,0),"EMPDATA"**

OR

Function

Performs an inclusive logical OR of two expressions.

AOS/VS	DG/RDOS	UNIX

Format

OR(*expression1,expression2)*

Arguments

expression1 and *expression2*	The numeric expressions or variables you want compared.

What It Does

The inclusive **OR** function is used to set bits in a binary expression. The binary representations of the two arguments are compared bit by bit. If a bit is set to 0 in both expressions, that bit is set to 0 in the result. If a bit is set to 1 in either or both expressions, that bit is set to 1 in the result.

How to Use It

Use the **OR** function to set bit flags or to combine two sets of flags into a single expression. You can use the **OR** function in any numeric expression.

Figure 1–8 shows the result when the bit values of two expressions are compared using **OR**.

OR (192, 127)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Power of 2
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	$expr_1$ = 192
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	$expr_2$ = 127
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	RESULT = 255

Logical OR of Two Numbers

Figure 1–8 Logical OR of Two Numbers

OR

continued

Example

The **OR** function is used to display the value obtained by setting the rightmost seven bits in addition to any other bits set in the initial value of X.

```
00010 INPUT "Initial value of X: ",X
00020 PRINT "Value of OR(X,127): ",OR(X,127)

* RUN
Initial value of X: 192
Value of OR(X,127): 255
```

OR

Operator

Performs a Boolean logical OR of two expressions.

AOS/VS	DG/RDOS	UNIX

Format

expression **OR** *expression*

Arguments

expression	A numeric or relational expression or variable. If *expression* is zero, the condition is false; otherwise the condition is true. Relational expressions consist of two numeric or string expressions separated by a relational operator. The relational operators are: greater than (>), greater than or equal (>=), less than (<), less than or equal (<=), equal to (=), and not equal to (<>). The expressions can also be subscripted variables.

What It Does

OR joins two expressions into a single expression. When executing an **IF** statement, all the expressions united by the **OR** are evaluated and reduced to 0 or 1; if either expression is non-zero (true), the result of the **OR** is 1 (true); otherwise, the result is 0 (false).

Note that the **OR** function which checks bits set to one is also available and is described separately.

How to Use It

OR may be used any place an expression is valid.

Before the **OR** Boolean Logic operator is executed, the expressions are evaluated as true or false, and the operands are reduced to zero or one. Any non-zero evaluation makes the expression true.

The precedence of all operators is given below.

```
highest          ^ (exponentiation)
   .             unary +, unary -, NOT
   .             *, /
   .             +, -
   .             <>, >, >=, =, <=, <
   .             AND
lowest           OR
```

OR

continued

Examples

1. If either variable B or C is not zero, then begin execution at line 100.

   ```
   00030  IF B OR C THEN GOTO 00100
   ```

2. If either expression evaluates to true then execute the subroutine that begins at line 300.

   ```
   00060  IF X>12 OR Y=1 THEN GOSUB 00300
   ```

PACK

Statement and Command

Composes a record string.

AOS/VS	DG/RDOS	UNIX

Format

```
     { format-string-variable }
PACK { format-string-literal  } ,string-variable,variable-list
     { line-number            }
```

Arguments

format-string-variable	A string variable containing the format information that describes the composition of *string-variable*.
format-string-literal	A string literal in quotation marks that specifies format information describing the composition of *string-variable*.
line-number	The statement number of an **RFORM** statement.
string-variable	A string variable or a string array element (UNIX only) to contain the encoded *variable-list*.
variable-list	A list of string and/or numeric expressions to be encoded into the *string-variable* argument.

What It Does

PACK encodes string and numeric information into a single string variable that is treated as a record for I/O purposes.

How to Use It

The *string-variable* must be filled through the last byte to be used with a **PACK** statement. The expression list can contain string literals, numeric literals, numeric expressions, string variables, substrings, numeric variables, and arrays. The format string contains format characters that define how the expression list should be encoded into the string-variable. The *line-number*, which can replace the format string, must be an **RFORM** statement with an appropriate format string. See **RFORM** for the formatting characters and their interpretation.

On AOS/VS and UNIX systems, you can repeat a **PACK** statement by enclosing the format in parentheses and specifying the number of repeats outside the parentheses—for example, **3(A5)**.

PACK

Example

Pack a record string and a compound key.

```
:Compose and write a record string containing record status, name
:address, city, state, zip code, balance, and age.
. . .
00220 RFORM ZJA20A30A12A2DLB
:           |||  |  |  | ||*1-byte unsigned integer for AGE%
:           |||  |  |  | |*4-byte signed integer for BAL
:           |||  |  |  | *3-byte unsigned integer for ZIP
:           |||  |  |  *2-byte truncated string for STTE$
:           |||  |  *12-byte truncated string for CITY$
:           |||  *30-byte truncated string for ADR$
:           ||*20-byte truncated string for NAME$
:           |*2-byte signed integer for STAT%
:           *null fill CUSREC$
. . .
00870 PACK 00220,CUSREC$,STAT%,NAME$,ADR$,CITY$,STTE$,ZIP,BAL,AGE%
00880 LWRITE FILE[4,record-number],CUSREC$
:
:Compose a compound key for an index consisting of customer number,
:order number, type code, and order line item number.
:
01280 PACK "ZLDAB",ORDRKY$,CNUMBR,ONUMBR,OTYPE$,OLNNUM%
:           ||||*1-byte unsigned integer for OLNNUM%
:           |||*1-byte string for OTYPE$
:           ||*3-byte unsigned integer for ONUMBR
:           |*4-byte signed integer for CNUMBR
:           *null fill ORDRKY$ to ensure it is long enough
. . .
01290 KFIND 18,BUF$,ORDRKY$,ORECNO    :Find order key in index
                                      :LOPENed as logical file 18
```

PAGE

Command

Sets the page width for output on a terminal.

AOS/VS	DG/RDOS	UNIX

Format

PAGE=*width*

Arguments

width	An integer expression in the range from 14 to 132 that specifies the number of columns (characters) allowed on a print line.

What It Does

PAGE limits or expands the number of characters permitted on a line. Your width ranges from 14 to 132 characters. If output exceeds page size, a Carriage Return/New Line is inserted, and the remaining output is displayed on the following lines.

How to Use It

PAGE is a keyboard mode command. Set *width* equal to the maximum number of characters you allow on a line at your terminal. To set the page width in a program statement, use **STMA** 4,8.

Example

Set page width at 132 characters.

```
* PAGE=132
```

POS

Function

Determines the position of a substring in a string.

AOS/VS	DG/RDOS	UNIX

Format

POS(*string-expression1,string-expression2,expression*)

Arguments

string-expression1	The primary string, which can be expressed as a string variable, a string array element (UNIX only), a string literal in quotation marks, or a subscripted string variable.
string-expression2	The substring whose position you want to determine, expressed as a string variable, a string array element (UNIX only), a string literal in quotation marks, or a subscripted string variable.
expression	A numeric expression for the position in the *string-expression1* to start the search for *string-expression2*.

What It Does

Starting at position *expression* in a string (*string-expression1*), **POS** searches for the specified substring (*string-expression2*), and returns the first position of the substring in the string as a numeric value. If **POS** cannot find the substring, it returns zero. If your expression is less than 0, an error message occurs. If your expression is greater than the string's current length, **POS** returns zero. If *expression* equals zero, the search begins at the first position of the string.

Example

In this example **POS** starts the search at position 6 ("F") of string A$, thereby missing the first "MNOP" and finding the second "MNOP" starting at position 13.

```
00005  DIM A$(25)
00010  LET A$="AMNOPFGHIJKLMNOPQRS"
00020  LET X=POS(A$, "MNOP", 6)
00030  PRINT X
* RUN
13
```

POSITION FILE

Statement and Command

Positions the file pointer to a byte in a disk file.

AOS/VS	DG/RDOS	UNIX

Format

POSITION FILE(*channel,byte*)

Arguments

channel	The channel number of a file opened in mode 0, 3 (AOS/VS and UNIX), 4, 5, or 6 (for random access); a numeric expression.
byte	A numeric expression or variable for the relative byte within a file to which you want to position the file pointer; for example: *(record-number * record-size)*. In AOS/VS and UNIX, *byte* can be −1. This automatically positions the pointer at the end of the file.

What It Does

POSITION FILE moves the file pointer to a specific byte relative to byte 0 of the file. Your byte expression can contain a record size multiplied by the number of records it takes to get to the specific relative byte where the record starts. (The **POSFL.SL** subroutine uses this form of **POSITION FILE**. **POSITION FILE** also resets the end-of-file flag for the **EOF** function.) File input/output statements move the file pointer whenever they transfer data.

How to Use It

Use **POSITION FILE** to move the file pointer quickly and accurately. You must know either the relative byte you want or the size of the fixed-length records and the number of the record you want. In some environments, relative byte locations are not known. If you are using subfiles, it is easier to use the **POSFL.SL** subroutine with the file characteristics array. You must open your file for random access to use **POSITION FILE**.

Examples

1. Position the file pointer to a place in the file calculated from the record number and the record size.

```
00005  DIM RECORD$[100]
00010  OPEN FILE[0,5], "RANDOM"
00020  INPUT "RECORD NUM TO READ:",RNUM
00030  LET RSIZE=100
00040  LET BYTE=RNUM*RSIZE
00050  POSITION FILE[0,BYTE]
00060  READ FILE[0],RECORD$
```

POSITION FILE

continued

2. First, we move the file pointer to byte 1084 in file **QUICK**. Then we read 50 bytes (bytes 1084–1134) into RECORD$.

```
* OPEN FILE(1,0),"QUICK"
* POSITION FILE(1,1084)
* DIM RECORD$(50)
* READ FILE(1),RECORD$
```

PRINT

Statement and Command

Prints output to a terminal or file.

AOS/VS	DG/RDOS	UNIX

Format

$$\left\{\begin{matrix}\textbf{PRINT}\\ ;\end{matrix}\right\}\left[\textbf{FILE}\ (\textit{channel}),\right]\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\textit{expression}\right]\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\textit{expression}\ \ldots\right]\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\right]$$

Arguments

channel	A numeric expression for the channel number of a file if you are outputting to a file or device; the file must be opened for output.
expression	The item to be printed. Numeric or string expressions (variables, numbers, array elements, string literals, string variables), @ expressions (terminal control and cursor positioning), TAB expressions, numeric functions, but not string functions. You can repeat expression. If you omit *expression* in a **PRINT** or a **PRINT FILE** statement, a blank line is output. TAB and @ expressions are discussed below. If expression is a string, the current length of the string is used to determine the number of bytes to be transferred.
;	A semicolon prints the next expression immediately after the previous output. If you use the semicolon at the end, it applies to the next **PRINT** (unless you use a TAB expression).
,	A comma places the next expression in the next tab zone; if you use a comma at the end, it applies to the next **PRINT**.

What It Does

PRINT outputs numeric and string data to a terminal or file. **PRINT** and **PRINT FILE**(16) perform terminal output; **PRINT FILE**(*channel*) performs file output.

PRINT outputs text with automatically generated delimiters and is meant to be used with **INPUT** on text files and for terminal I/O. To perform binary I/O on data and other types of files, use **READ** and **WRITE**.

Numeric values are displayed with leading space or minus sign and a trailing space. When separated by commas, multiple expressions are displayed in the tab zones (set by the **TAB** command or **STMA 4,9**). When separated by semicolons in the **PRINT** statement, multiple expressions follow each other immediately.

PRINT

continued

A comma or semicolon at the end of the **PRINT** statement causes the next **PRINT** statement to begin displaying data without moving to the next line. When nothing follows the last expression, an end-of-line character (Carriage Return on DG/RDOS systems; New Line on AOS/VS and UNIX systems) is generated automatically. Output reaches the page width (set by the **PAGE** command or **STMA 4,8**) unless the column counter is disabled (by **STMA 6,3**). A single **PRINT** statement can output up to 132 characters unless you used **STMA 6,13** to extend this limit to 255 characters.

Because the **PRINT** statement is data sensitive, it interprets and acts on certain special characters, including null, form feed (Erase Page), and end-of-line. After carrying out the action denoted by the special character, **PRINT** continues its processing. **PRINT FILE**, however, stops its processing when it encounters a null, a form feed, an end-of-line, or another special "terminator" character. To output special characters, use **WRITE**.

PRINT FILE ignores terminal control (@) functions except for @(-30), which generates a form feed. Expressions to be printed should not contain embedded control characters if you want your program to be portable.

How to Use It

When you type a **PRINT FILE** statement or command, you can enter a semicolon (;) in place of **PRINT**, as follows: ; **FILE(0)**, "*expression*".

You select 7- or 8-bit operation mode by setting the hardware switches at the back of the terminal or by choosing the mode from the terminal keyboard using the menus for terminal configuration setup. In addition, on AOS/VS and DG/RDOS systems, if the hardware switch is set for 8-bit mode, you can change the operation mode by using a software command. When you use Business BASIC in 8-bit mode, only characters in the range 200-237 octal (128-159 decimal) have their high bit stripped by Business BASIC. All other characters with the high bit set in the range 240-377 octal (160-255 decimal) are printed exactly as specified. When you use Business BASIC in 7-bit mode, all characters in the range 200-377 octal (128-255 decimal) have their high bit stripped by the terminal at the time of output from a **PRINT** statement and then are displayed on the screen.

On UNIX systems, 8-bit mode is only supported when you are using DG mode. You specify DG mode by including the -D option when you execute Business BASIC.

You can set the tab zones at your terminal using the **TAB** statement or accept the default tab setting (14 columns).

To Print in Tab Zones

Use commas between each expression if you want to print in tab zones. A comma prints the next expression in the first column of the next tab zone. **PRINT** checks to see if expression's output fits on the line; if not, it starts printing in the first column of the next line (the next tab zone). A comma at the end of the list of expressions (at the very end of the **PRINT** statement) will output the next **PRINT** statement to the same line in the next tab zone. If there is no punctuation at the end of the **PRINT** statement, the next **PRINT** statement outputs to a new line.

For a Compact Output

Use semicolons between expressions if you want compact output. A semicolon prints the next expression in the next column. Characters take up only one column position; if you want a space between characters, put the space in a string literal. Numeric output results in a space for a minus sign, the number of digits, and one trailing space. If you separate numeric output with a semicolon, a space still occurs between each number in the output. A semicolon at the end of a **PRINT** statement (after all the expressions) outputs the next **PRINT** statement to the same line in the next column position; whereas, without the semicolon the next **PRINT** statement outputs to a new line.

To Print a Blank Line

You can print a blank line using any of the following methods:

- Use **PRINT** without expressions or punctuation.
- Use terminal control and cursor positioning (@) expressions.
- Use the **TAB** expression.

The TAB Expression

Move the next expression's output to column *n* on the same line by using a **TAB**(*n*) expression, where *n* is an integer. **TAB** numbers columns from 0 to page width–1. If *n* is an integer greater than the page width, *n* is reduced to the remainder after dividing *n* by the page width: **MOD**(*n*,*w*). For example:

```
00030  PRINT TAB(99);"TESTCASE"
```

would cause **TESTCASE** to print starting at position 19.

Terminal Control and Cursor Positioning

Always separate @ expressions from *expression* by following them with a semicolon, because a comma can put the cursor in the wrong position. Also disable the column counter (**STMA 6,3**). Otherwise, the extra characters output by @ expressions can cause the line to appear as if it exceeds the current page width causing an unwanted carriage return/line feed. These extra characters are sent to the terminal by Business BASIC, and they have a special meaning that is understood by the terminal.

PRINT FILE statements ignore all @ expressions except @(–30), which generates a form feed to skip to the next page. You can also use terminal control and cursor positioning expressions in **INPUT USING** statements.

PRINT

Here are formats for terminal control and cursor positioning expressions:

@(*line,column*)	For cursor positioning
@(*item,value*)	For terminal characteristics
@(*item*)	For terminal control characters

Cursor Positioning

Use the format @(*line,column*) where *line* is the line number to which you want to position the cursor on the terminal (from one to 24) and *column* is the column position where you want to start the next output (from one to the page width). For example:

```
PRINT @(3,15)
```

positions the cursor to line 3, column 15. When you only specify a line number with this command, Business BASIC positions the cursor to column 1 of that line number. Thus,

```
PRINT @(14)
```

positions the cursor to line 14, column 1.

Terminal Characteristics

You can temporarily manipulate some terminal characteristics using **STMA** and **PRINT** @ statements. (See **STMA 4**.) We list each item below with the appropriate STMA reference. For terminal characteristics use the format:

@(*item,value*)

where *value* is the ASCII value of the character you want to assign to *item*, and *item* is one of the following (always negative):

-1	Line cancel key, see **STMA 4,1**.
-2	Character delete echo, see **STMA 4,2**.
-3	Character delete key, see **STMA 4,3**.
-4	Primary unpend key, see **STMA 4,4**.
-5	Secondary unpend key, see **STMA 4,5**.
-6	Primary interrupt key, see **STMA 4,6**.
-7	Secondary interrupt key, see **STMA 4,7**.
-8	Page width, see **PAGE** and **STMA 4,8**.
-9	Tab size, see **TAB** and **STMA 4,9**.
-10	Maximum number of characters allowed on input. This allows you to fix the number of characters a typist could type to the next input request. Unpending occurs on the last character if you set the maximum number of characters in *value* to a negative number.
-11	Reserved.

-12	First-echoed character when line cancel occurs, see **STMA 4,12**.
-13	Second-echoed character when line cancel occurs, see **STMA 4,13**.
-14	Pad character, sent after all line feeds, see **STMA 4,14**.
-15	Number of pad characters to send, see **STMA 4,15**.
-16	Reserved.
-17	Reserved.
-18	Set or clear the indicated job status bit (see **STMA 6** and **7**). A positive bit number sets the specified bit while a negative bit number clears the specified bit. The format is **PRINT @(-18,*n*)**.

Terminal Control Characters

NOTE: Not all functions work for a given terminal type. Implementation depends on the capabilities of the terminal type. On AOS/VS and DG/RDOS systems, see the CRT source modules in the **$DOC** directory to determine which functions work with each terminal type. On UNIX systems, see the information on tuning the **terminfo** files in the **DOC** directory that is supplied with the release.

You can output some control characters before and after expressions in a **PRINT** statement to perform special terminal functions. Use the format:

@(*item*)

where *item* is one of the following (always negative):

-19	Reset items -1 to -15 to their default values for your terminal. If you changed your terminal type with **STMA 2,0** and you want the new terminal type's default characteristics, you must immediately follow the **STMA 2,0** with this **PRINT @(-19)** reset.
-20	Move cursor to "home" (top line, first column of CRT screen).
-21	Move cursor right one column.
-22	Move cursor down one line.
-23	Move cursor left one column.
-24	Move cursor up one line.
-25	Sound the bell or audible alarm on the terminal.
-26	Tab to next field on screen.
-27	Return cursor to first column of current line.
-28	Move cursor to beginning of next line (new line).
-29	Reserved.
-30	Clear entire screen (or generate a form feed in a file).
-31	Clear unprotected positions of the screen—high intensity (not available on UNIX systems).
-32	Clear to end of the line.
-33	Clear to end-of-screen.
-34	Lock keyboard (not available on UNIX systems).

PRINT

continued

-35	Unlock keyboard (not available on UNIX systems).
-36	Insert a line.
-37	Delete a line.
-38	Start low-intensity field.
-39	Start high-intensity field.
-40	Start blinking field.
-41	End blinking field.
-42	Roll disable (page mode).
-43	Roll enable (turn off page mode).
-44	Set program mode (not available on UNIX systems).
-45	Clear program mode (not available on UNIX systems).
-46	Set block mode (not available on UNIX systems).
-47	Clear block mode (not available on UNIX systems).
-48	Set flag 1 (not available on UNIX systems).
-49	Clear flag 1 (not available on UNIX systems).
-50	Send line (unprotected fields) (not available on UNIX systems).
-51	Send line (all fields) (not available on UNIX systems).
-52	Start underscored field.
-53	End underscored field.
-54	Turn underscore and bright on.
-55	Turn underscore and bright off.
-56	Restore default display enhancement.
-57	Reverse video on.
-58	Reverse video off.
-59 to -256	Reserved.

Examples

1. Here's an example of how to display a blinking underscored message:

   *** PRINT @(-40);@(-52);"HI! I'M YOUR CRT!";@(-53);@(-41)**

2. This example clears the screen and displays powers of 2 and their values in columns.

```
00010  PRINT @(-30);@(1,10);"POWERS OF 2";@(1,30);"VALUE"
00020  FOR I=0 TO 7
00030    PRINT @(I+2,15);I;@(I+2,30);2^I
00040  NEXT I
* RUN
         POWERS OF 2         VALUE
               0              1
               1              2
               2              4
               3              8
               4              16
               5              32
               6              64
               7              128
```

PRINT USING

Statement

Formats output to a terminal, file, or device.

AOS/VS	DG/RDOS	UNIX

Format

$\left\{\begin{matrix}\text{PRINT} \\ ;\end{matrix}\right\}$ [FILE(*channel*),] USING $\left\{\begin{matrix}\textit{"formats"} \\ \textit{string-variable}\end{matrix}\right\}$ *expression*[*expression* ...]

Arguments

channel	A numeric expression for the channel number of a file or device opened for output when you are outputting to a file.
formats	A string literal in quotation marks that contains any number of formats used to output *expression*; formats are described in "What It Does."
string-variable	A regular or subscripted string variable assigned the value of *formats*.
expression	Numeric and string expressions (variables, numbers, array elements, string literals, and string variables) and numeric functions but not string functions; you can repeat *expression* with any combination of these. This is an optional argument.

What It Does

PRINT USING enhances computer output significantly. This statement is similar to the **FORMAT** statement in FORTRAN and the **PICTURE** statement in COBOL. TAB expressions and semicolons are not permitted with this statement.

When executing a **PRINT USING** statement, Business BASIC uses a 132-character buffer to format the text for display unless you used **STMA 6,13** to specify a 255-character buffer for the statement (AOS/VS and UNIX only). You can use a variety of format arguments to put this text into the buffer. You can use arguments to specify that a string or numeric expression is displayed, that the output starts at a new position in the buffer, that characters are printed only if they match a specified character, or that a certain character is used for padding. When you use the format arguments, remember that spaces serve as delimiters where a number is required. This differs from the rest of Business BASIC, where spaces are ignored in numeric values. The following arguments are used to format the buffer:

'	A single quotation mark places a string literal in the buffer. Use another single quotation mark to terminate this string literal.

PRINT USING

continued

A*n*	Outputs *n* characters from a string expression. *n* has no default value; therefore, if you need only one character, you must specify **A1**. If the string is less than *n* characters, spaces are put to the right of the string. Spaces replace nulls in the string.
C*n* (AOS/VS, UNIX)	Centers a format string in the amount of space specified by *n*.
D*w*.*d*	Outputs a numeric expression using a total of *w* columns with *d* digits to the right of the decimal point. No decimal point is printed if *d* is 0. If the numeric expression is negative, Business BASIC places a minus sign in the leftmost position of the field. Leading characters are suppressed, and the output is right-justified.
E*w*.*d*	The same as D*w*.*d* except the minus sign or space is printed in the rightmost position.
F*w*.*d*	The same as D*w*.*d* except the minus sign is printed immediately to the left of the number and does not fill characters to the left of the sign.
K*w*	Outputs the equivalent octal number using *w* columns for *expression*. The expression must be in the range -2,147,483,648 to 2,147,483,647. Note that this affects only triple and quad precision.
L6*c* or L6_*c*	Sets the floating fill character to *c*; the floating fill character is a space by default. If the number you are printing does not fill the field, BASIC uses the floating fill character. The form L6_*c* is required if *c* is numeric in order to separate *c* from "6".
L7*c* or L7_*c*	Designates *c* as the overwrite character. BASIC replaces any *c* character in the buffer with numeric digits. The overwrite character is a space by default. The form L7_*c* is required if *c* is numeric in order to separate *c* from "7".
L8*c* or L8_*c*	Sets the fill character to *c*. The fill character is a space by default. The fill character replaces leading zeros in D*w*.*d* and E*w*.*d* formats unless the **STMA 6,7** flag is set. If this flag is set, the fill character replaces all characters not used in D*w*.*d* and

	E*w*.*d* formats. The format L8_*c* is required if *c* is numeric in order to separate *c* from L8. For example, the following statement: PRINT USING "('JJJ-JJ-JJJJ',L8_0,L7_J,T0,D11.0)",12 will by default print: 000-00-1234 but if preceded by an **STMA 6,7**, the following will be printed: 00000001234
L9	Used in place of repeat count. BASIC uses the next expression in expressions as the repeat count to repeat use of formats.
L10	Copies the edited contents of the buffer to a string variable in the expression.
L11	Copies the string expression in *expression* into the buffer.
O	Used immediately in front of D*w*.*d*, E*w*.*d* and F*w*.*d* formats to suppress printing if *expression* is 0.
P	Ends the line in the buffer with a form feed instead of a carriage return (skip to top of next page on a DASHER® display terminal).
R*n*	The same as A*n* except any fill characters required are placed on the left, making the output string right justified.
S*n*	The same as A*n* except no filling is done if the string is less than *n*.
T*n*	Sets the column pointer to the *n*th position of the buffer, where 0 <= *n* <= 132 (unless you used AOS/VS' **STMA 6,13** to extend the limit from 132 to 255). If *n* + (length of formatted output) > 132 (or 255), you get `Error 60 - Line too long`. If *n* > 132 (or 255), Business BASIC displays `Error 50 - Invalid operator command`.
*n*X	Moves the column pointer *n* positions relative to its location in the buffer; *n* may be negative.
Z	Ends the line in the buffer with a null instead of a carriage return.

How to Use It

Enclose the list of arguments (strings and formats) following the **PRINT USING** statement in double quotation marks ("). Indicate each string argument by enclosing it in single quotation marks ('). You can mix strings and formats in any order as long as

PRINT USING

you separate them with either commas or spaces. You can also repeat an expression if you precede it with a number indicating how many times you want it to repeat; for example, 3D6.2 repeats three times. To make the expression easier to read, you can enclose it in parentheses; for example, 3(D6.2). If the format string is exhausted before the expression list is, Business BASIC repeats the format string for the remaining expressions.

Examples

1. A*n* R*n*

   Notice that A3 limits the output from the buffer to three characters left justified. The R10 right justifies the five characters from the buffer with five additional spaces.

   ```
   00010 DIM A$(5)
   00020 LET A$="ABCDE"
   00030 PRINT USING "A3",A$
   00040 PRINT USING "R10",A$
   * RUN
   ABC
        ABCDE
   ```

2. S*n* Z

   Here, the Z format is used with both A10 and S10. In the case of A10, additional printing is done after a fill of five spaces (creating a 10-character output). With S10 there is no fill and the second string is printed immediately after the first.

   ```
   00010 DIM A$(5)
   00020 LET A$="ABCDE"
   00030 PRINT USING "A10",A$
   00040 PRINT USING "A10,Z",A$
   00050 PRINT "THIS PHRASE WAS ADDED AFTER 'AN' FORMAT"
   00060 PRINT USING "S10",A$
   00070 PRINT USING "S10,Z",A$
   00080 PRINT "THIS PHRASE WAS ADDED AFTER 'SN' FORMAT"

   * RUN
   ABCDE
   ABCDE     THIS PHRASE WAS ADDED AFTER 'AN' FOR MAT
   ABCDE
   ABCDETHIS PHRASE WAS ADDED AFTER 'SN' FORMAT
   ```

3. D*w.d*

   The format D8.2 implies that the value of N has a total of eight characters (7 + the decimal point) with two of these characters after the decimal point.

   ```
   00010 LET N=1234567
   00020 PRINT USING "D8.2",N
   * RUN
   12345.67
   ```

4. D*w*.*d*
   When the value of N is negative, the minus sign is the first character of the D15.2 field.

```
00010 LET N = -1234567
00020 PRINT USING "D15.2",N
* RUN
-      12345.67
```

5. E*w*.*d*
   When the value of N is negative, the minus sign is the last character of the E15.2 field.

```
00010 LET N = -1234567
00020 PRINT USING "E15.2",N
* RUN

      12345.67-
```

6. F*w*.*d*
   When the value of N is negative, the minus sign is immediately to the left of the value.

```
00010 LET N = -1234567
00020 PRINT USING "F15.2",N
* RUN
      -12345.67
```

7. K*w*
   This format produces the octal equivalent of the value of N.

```
00010 LET N = 123
00020 PRINT USING "K4",N
* RUN
173
```

8. L6
   BB-BBBB is a mask for the value of X. The mask calls for six characters, while the value of X only provides five. L6 provides the fill character, which is given as 0. L7 identifies the character (B) as the overwrite character for the value of X. T0 begins the formatting with the first character in the buffer (zero character). Notice that the mask is contained in single quotation marks (′).

```
00010 LET X = 12345
00020 PRINT USING "′BB-BBBB′,L6_0,L7B,T0,D11.0",X
* RUN
01-2345
```

PRINT USING *continued*

9. L7

   JJJ–JJJJ is a mask for the value of N. Notice the mask is contained in single quotation marks (´). L7 identifies (J) as the overwrite character in the mask. The T0 format identifies the first (zero) character in the buffer as the beginning of output. The D format calls for a total of eight characters (seven from the value of N plus a dash).

```
00010 LET N = 1234567
00020 PRINT USING "´JJJ-JJJJ´/,L7J,TO,D8.0",N
* RUN
 123-4567
```

10. L8

    BBB–BB–BBBB is a mask for the value of X. L7 identifies the overwrite character in the mask. If the value of X does not contain enough digits to fill the mask, the L8 format provides the fill character. T0 identifies the first (zero) character in the buffer as the beginning of output. The D format declares a total of 11 characters to be printed with no implied decimal point.

```
00010 LET X = 12345
00020 PRINT USING "´BBB-BB-BBBB´ ,L8_0,L7B,TO,D11.0",X
* RUN
000-01-2345
```

11. L8 STMA 6,7

    Option 7 of STMA 6 provides leading zeros for the D format. See Example 10 for an explanation of L8_0, T0, and D11.0.

```
00010 STMA 6,7
00020 LET X = 12345
00030 PRINT USING "´BBB-BB-BBBB´,L7B, L8_0, TO, D11.0", X
* RUN
000001-2345
```

12. This technique is used for asterisk protection in check writing. To eliminate the comma embedded in the field of asterisks, use **STMA 6,7**.

```
00010 LET N = 1234567
00020 PRINT USING "´$´,´SSS,SSS,SSS.SS´, L7S, TO, L8*, D15 . 0",N
* RUN
$***,*12,345.67
```

13. Repeat Count

```
00010 LET X=12345 \ Y=67890 \ Z=55555
00020 PRINT USING "D8.2",X
00030 PRINT USING "3(D8.2)", X, Y, Z
* RUN


   123.45
   123.45   678.90   555.55
```

PRINT USING

14. Z *"string"*

```
00010 DIM NAME$(25)
00020 LET NUM=12345
00030 LET NAME$="DATA GENERAL"
00040 PRINT USING "A25,Z",NAME$
00050 PRINT USING "'ACCOUNT NUMBER: ',D5.0",NUM
* RUN
DATA GENERAL        ACCOUNT NUMBER: 12345
```

15. Formats
See Example 10 for an explanation of L7_0, T0, and D8.0.

```
00010 DIM FORMAT$(22)
00020 LET PHONE = 3668911
00030 LET FORMAT$="'000-0000',L7_0, TO, D8.0"
00040 PRINT USING FORMAT$,PHONE
* RUN
366-8911
```

16. T*n*
See Example 10 for an explanation of L7_0, T0, and D8.0.

```
00010 LET PHONE=5551605
00020 PRINT USING "T15,'000-0000',L7_0,TO,D8.0",PHONE
* RUN
             555-1605
```

17. **PRINT FILE**
A20 formats NAME$ and D2.0 formats GRADE.

```
00010 OPEN FILE(0,0), "DATA"
00020 DIM NAME$(20)
00030 READ NAME$,GRADE
00040 IF NAME$ = "DONE" THEN GOTO 00070
00050 PRINT FILE(0), USING "A20,D2.0",NAME$,GRADE
00060 GOTO 00030
00070 CLOSE FILE(0)
00080 PRINT "JOB IS DONE"
00090 DATA "ABBOT",  95
00100 DATA "BROWN",  84
00110 DATA "MORGAN", 72
00120 DATA "DONE",0
* RUN
JOB IS DONE
* !TYPE DATA
ABBOTT                  95
BROWN                   84
MORGAN                  72
```

PRINT USING *continued*

18. L9

    L9 allows a format to be repeated a different number of times each time **PRINT USING** is executed, depending on a variable's contents.

```
00010 LET PSIZE=4
00020 GOSUB 00100
00030 LET PSIZE=8
00040 GOSUB 00100
00050 END
00100 PRINT USING "L9'X'",PSIZE
00110 RETURN
* RUN
XXXX
XXXXXXXX
```

19. L10

    L10 is used here to move an edited value to a string variable that has been dimensioned to 132 bytes, the size of the print buffer.

```
00010 DIM OUT$[132]
00020 LET AMOUNT=312298
00030 PRINT USING "D10.2,L10",AMOUNT,OUT$
* RUN
   3122.98
* PRINT OUT$
   3122.98
```

20. In this second L10 example, data from the NUM variable and the NAME$ string variable are copied into the LINE$ string variable.

```
* LIST
00010 DIM NAME$[25],LINE$[80]
00020 LET NUM=12345
00030 LET NAME$="Data General Corporation"
00040 PRINT USING "A25,1X,D5.0,L10,Z",NAME$,NUM,LINE$
00050 PRINT "LINE$ now contains the data that is in NAME$ and
NUM"
00060 PRINT LINE$

* RUN
Data General Corporation  12345
LINE$ now contains the data that is in NAME$ and NUM
Data General Corporation  12345
```

21. C*n*
    C*n* is used here to center the text "QUARTERLY REPORT" in a 50-space field.

```
00010 DIM STR$[50]
00020 LET STR$="QUARTERLY REPORT"
00030 PRINT USING "C50",STR$
```

```
* RUN
                 QUARTERLY REPORT
```

22. L11
    The L11 format is used here to concatenate L11PT2$ with L11PT1$ so their contents can be printed together.

```
* LIST
00010 DIM L11PT1$[19],L11PT2$[10]
00020 LET L11PT1$="CONCATENATE USING "
00030 LET L11PT2$="L11 WORKED"
00040 PRINT USING "L11,T19,A10",L11PT1$,L11PT2$
```

```
* RUN
CONCATENATE USING L11 WORKED
```

PROGRAM DISPLAY

Command

Displays information about a program in working storage or in a SAVE file.

UNIX

Format

PROGRAM DISPLAY [[*input-file*], [*output-file*],*variable-toggle*]

Arguments

input-file	A string expression or string literal containing the name of the SAVE file program about which you want to display information. This argument is optional.
output-file	A string expression or string literal containing the name of the file to which you want the **PROGRAM DISPLAY** output written. This argument is optional.
variable-toggle	A numeric expression that is evaluated by **PROGRAM DISPLAY** to determine whether the variable count and listing should be displayed. If *variable-toggle* is 0, the variable count and listing are not displayed. The default is 0. If *variable-toggle* is any other value, the variable information is displayed.

What It Does

PROGRAM DISPLAY displays the following information about the program in working storage or a SAVE file (if you specify *program-name*):

- An optional variable count and listing of the variables
- The **GOSUB** stack status
- The line number of the last executed line
- The line number of the next line to execute
- The last error number and the line number where it occurred
- The channel number of the last file accessed
- The line number of the active **ON ERR**, if defined
- The line number of the active **ON IKEY**, if defined
- The name of any user-defined functions, and the line number where each is defined
- The **FOR...NEXT** stack status

- The **DO** stack status
- Program and data size information
- The current DATA line
- The SAVE file format of the program (native or non-native format)

How to Use It

Use **PROGRAM DISPLAY** in keyboard mode.

You can specify **PROGRAM DISPLAY** arguments in any combination. If you specify only one or two of the arguments, you must use a comma in the place of any argument you do not use. However, you do not need to specify ending commas.

When you are executing Business BASIC in DG mode, then **PROGRAM DISPLAY** produces output that is similar to the output of the PD utility.

When you are executing in non-DG mode, **PROGRAM DISPLAY** produces output in a window. If there is more data than can be displayed on one screen, Business BASIC prompts you with the word MORE in the bottom border of the window. At this point, you can press the **q** key to quit the display, New Line to scroll down one line in the display, or the space bar to scroll down one screen.

Examples

1. This statement writes information about the program in working storage to the file **OUTFILE**. In this case, the program information would not include a variable count and listing.

   ```
   * PROGRAM DISPLAY,"OUTFILE"
   ```

2. This statement displays on your screen information about a SAVE file called RPT1. A variable count and listing would be included in the display.

   ```
   * PROGRAM DISPLAY "RPT1",,1
   ```

PROTECT

Command

Protects Business BASIC SAVE files.

UNIX

Format

PROTECT *"filename"*

Arguments

filename	The name of the SAVE file you want to protect.

What it Does

PROTECT modifies a SAVE file so that users cannot use the **LIST** command to display the file. You can use the **LOAD**, **RUN**, **CHAIN**, and **SWAP** commands on a protected file.

PROTECT does not change a program's line numbers.

How to Use It

There is no way to "unprotect" a SAVE file once you protect it. Therefore, keep an unprotected copy of the file in a secure location. You can then modify the unprotected program, if necessary.

Use this command in keyboard mode. Business BASIC displays an error message if you omit the filename.

Example

This example protects **PROG1**.

```
*  PROTECT "PROG1"
*
```

QADD

Statement and Command

Adds using quad precision.

AOS/VS	DG/RDOS	UNIX

Format

QADD *string-variable1,string-variable2,string-variable3*

Arguments

string-variable1	The string variable that stores the sum of *string-variable2* and *string-variable3*.
string-variable2	The string variable that is added to *string-variable3*.
string-variable3	A string variable that is added to *string-variable2*.

What It Does

The QADD statement (and all of the "Q" statements—**QLOAD**, **QSTORE**, **QSUB**, **QMUL**, and **QDIV**) allows for the manipulation of very large numbers in the range +/−9,223,372,036,854,775,807 by operating on eight-byte (64-bit) binary integers stored in string operands. Numbers in this format are often referred to as quad precision since they occupy four words in the processor (see **QLOAD** and **QSTORE**).

The **QLOAD** and **QSTORE** statements allow for conversion of a normal four-byte Business BASIC number into and from the eight-byte quad precision number format.

The first eight bytes of *string-variable1* contain the sum of the first eight bytes of *string-variable2* and *string-variable3*. Consequently, the statement

```
QADD A$,B$,C$
```

implies A$=B$+C$, where each string variable is eight bytes. If any of these strings contain less than eight bytes, it is treated as if it were padded with nulls. No indication of overflow is given.

How to Use It

When the value of a string variable is written to a program's symbol table, all eight bytes are written. For this reason, all string variables used in a quad precision statement must be dimensioned to at least 8 bytes. Violating this rule could cause Business BASIC to write the value into the location of the string variable being written, as well as into the bytes adjacent to that location. This error could result in overwriting another variable in the symbol table and lead to undesirable results.

QADD

continued

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

Example

Refer to the example for QDIV.

QDIV

Statement and Command

Divides using quad precision.

AOS/VS	DG/RDOS	UNIX

Format

QDIV *numeric-variable1,string-variable,numeric-expression*
[*,numeric-variable2,numeric-variable3*]

Arguments

numeric-variable1	The numeric variable that stores the quotient of *string-variable* divided by *numeric-expression*.
string-variable	The string variable that is to be divided by *numeric-expression*.
numeric-expression	The numeric expression or variable that is used as the divisor.
numeric-variable2	The numeric variable that holds the remainder of the **QDIV** operation. *numeric-variable2* is optional. When it is omitted, the remainder is lost, and no error is returned.
numeric-variable3	The numeric variable that returns an error code if an illegal division is attempted. *numeric-variable3* is an optional argument. This argument must be initialized (i.e., set to zero) before it is used.

What it Does

The **QDIV** statement (and all of the "Q" statements—**QLOAD** , **QSTORE**, **QSUB**, **QMUL**, and **QADD**) allows for the manipulation of very large numbers in the range -/+9,223,372,036,854,775,807 by operating on eight-byte (64-bit) binary integers stored in string operands. Numbers in this format are often referred to as quad precision since they occupy four words of space (see **QLOAD** and **QSTORE**).

The **QLOAD** and **QSTORE** statements allow for conversion of a normal four-byte Business BASIC number into and from the eight-byte quad precision number format.

The first eight bytes of *string-variable* are divided by *numeric-expression*, producing *numeric-variable1* as the quotient and *numeric-variable2* as the remainder. The optional *numeric-variable3* returns an error code if an illegal divide is attempted.

Consequently, the statement

```
QDIV A,B$,C,D
```

implies that A=B$/C and D=MOD(B$,C).

QDIV

If the length of *string-variable* is less than eight bytes, it is treated as if it were padded with nulls. If the division results in either *numeric-variable1* or *numeric-variable2* having a value outside the range for four-byte integers, `Error 16 - Arithmetic` (an overflow error) occurs. Rather than requiring an error trap, the optional argument of *numeric-variable3* returns the arithmetic error code.

How to Use It

When the value of a string variable is written to a program's symbol table, all eight bytes are written. For this reason, all string variables used in a quad precision statement must be dimensioned to at least eight bytes. Violating this rule could cause Business BASIC to write the value into the location of the string variable being written, as well as into the bytes adjacent to that location. This error could result in overwriting another variable in the symbol table and lead to undesirable results.

All string variables must be loaded (using **QLOAD**) from numeric variables before the string variable can be used in a quad precision arithmetic statement.

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

The use of the *numeric-variable3* argument suppresses execution of Business BASIC's default error trap or any **ON ERR** condition (which causes the program to halt) and instead returns to *numeric-variable3* the same error code as would be supplied by the appropriate **SYS** error function. Check the *numeric-variable3* value to determine whether an error has occurred.

Example

This example sums and averages four double precision numeric variables. If the average is within the range of a double precision number, it will be computed correctly, even if the sum of the four numbers is larger than the greatest double precision number.

continued **QDIV**

```
00010 DIM A$[8],B$[8],C$[8],D$[8],E$[8],ER$[64]
00020 INPUT "Four numbers to sum: ",A,B,C,D
00030 LET E,F,G,ER=0       :Numeric variables must be initialized
00040 QLOAD A$,A           :Get the numbers into quad variables
00050 QLOAD B$,B
00060 QLOAD C$,C
00070 QLOAD D$,D
00080 ON ERR THEN GOTO 00180
00090 QADD E$,A$,B$                 :Sum them into E$
00100 QADD E$,E$,C$
00110 QADD E$,E$,D$
00120 QDIV E,E$,4,F                 :Divide; remainder goes into F
00130 IF F>=2 THEN LET E=E+1:Round  0.5 up to 1
00140 PRINT "Average is";E
00150 QSTORE G,E$  :We may get an error, even if we didn't on
00160 PRINT "Sum is";G     :QDIV, if the sum but not the average is
00170 END                  :too large
00180 PRINT "Error";SYS(7);"occurred at line";SYS(20)
00190 LET ER$=ERM$(SYS(7))
00200 PRINT "The error message is ";ER$
```

```
* RUN
Four numbers to sum: 10,10,11,11
Average is 11
Sum is 42
```

```
* RUN
Four numbers to sum: 2000000000,2000000000,1000000000,1000000000
Average is 1500000000
Error 16 occurred at line 150
The error message is Arithmetic.
```

Also refer to the example for **QMUL**.

QLOAD

Statement and Command

Loads a numeric expression into a string variable.

AOS/VS	DG/RDOS	UNIX

Format

QLOAD *string-variable,numeric-expression*

Arguments

string-variable	The string dimensioned to at least eight bytes, that receives the four-byte value of *numeric-expression*. The sign bit of *numeric-expression* is extended across the high order four bytes of string.
numeric-expression	The numeric expression to be loaded into string.

What It Does

The four-byte value of *numeric-expression* is loaded into the first eight bytes of *string-variable* with the sign bit of *numeric-expression* being extended across the high order four bytes of string-variable.

The length of *string-variable* following the **QLOAD** is updated the same as for the corresponding **LET** statement. If the maximum length of *string-variable* is less than eight, an overflow results, and no error is indicated.

How to Use It

When the value of *string-variable* is written to a program's symbol table, all eight bytes are written. For this reason, all string variables used in a quad precision statement must be dimensioned to at least eight bytes. Violating this rule could cause Business BASIC to write the value into the location of the string variable being written, as well as into the bytes adjacent to that location. This error could result in overwriting another variable in the symbol table and lead to undesirable results.

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

Example

Refer to the example for **QDIV**.

QMUL

Statement and Command

Multiplies using quad precision.

AOS/VS	DG/RDOS	UNIX

Format

QMUL *string-variable,numeric-expression1,numeric-expression2*

Arguments

string-variable	The string variable that receives the product of *numeric-expression1* * *numeric-expression2*.
numeric-expression1	The numeric expression that is used as the multiplier.
numeric-expression2	The numeric expression that is used as the multiplicand.

What It Does

The QMUL statement (and all of the "Q" statements—**QLOAD**, **QSTORE**, **QSUB**, **QDIV**, and **QADD**) allows for the manipulation of very large numbers in the range -/+9,223,372,036 ,854,775,807 by operating on eight-byte (64-bit) binary integers stored in string operands. Numbers in this format are often referred to as quad precision since they occupy four words of space.

The **QLOAD** and **QSTORE** statements (see **QLOAD** and **QSTORE**) allow for conversion of a normal four-byte Business BASIC number into and from the eight-byte quad precision number format.

The full eight-byte product of the multiplication of *numeric-expression1* and *numeric-expression2* is placed in the first eight or fewer bytes of string. This allows multiplication of normal four-byte numbers to take place without the danger of overflow when using large numbers. If the length of *string-variable* is less than eight bytes, the string is treated as if it were padded with nulls.

How to Use It

When the value of a string variable is written to a program's symbol table, all eight bytes are written. For this reason, all string variables used in a quad precision statement must be dimensioned to at least eight bytes in length. Violating this rule could cause Business BASIC to write the value into the location of the string variable being written, as well as the bytes adjacent to that location. This error could result in overwriting of another variable in the symbol table and lead to undesirable results.

All string variables must be loaded (using **QLOAD**) from numeric variables before the string variable can be used in a quad precision arithmetic statement.

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

QMUL

continued

Examples

The examples below show two subroutines which work with two numbers and a third which is a representation of the percentage that one number is of the other. This percentage can range from 0 to 10,000; a value of 5,000 means that one number is 50% of the other.

```
: MUL - Compute X = Y * PCNT
:
: Calling sequence
: Y     Multiplicand
: PCNT  Multiplier
: GOSUB 7500
: X     Result
: Temporary QTMP$ must be dimensioned to at least 8 bytes
:
07500 REM \ MUL
07510 LET X=0               : In case the user didn't initialize it
07520 QMUL QTMP$,Y,PCNT
07530 QDIV X,QTMP$,10000    : 10000 because that is how 100%
                            : is represented
07540 RETURN
: DIV - Compute PCNT = (X * 10000 / Y without overflow
:
: Calling sequence
: X     Dividend
: Y     Divisor
: GOSUB 7550
: PCNT  Percentage X is of Y
:
: Temporary QTMP$ must be dimensioned to at least 8 bytes
:
07550 REM \ DIV
07560 LET PCNT=0            : In case the user didn't initialize it
07570 QMUL QTMP$,X,10000    : 10000 because that is how 100% is
                            : represented
07580 QDIV PCNT,QTMP$,Y
07590 RETURN
```

QSTORE

Statement and Command

Converts a quad precision string into a double precision variable.

AOS/VS	DG/RDOS	UNIX

Format

QSTORE *numeric-variable,string-variable*

Arguments

numeric-variable	A numeric variable that receives the value of the first eight bytes of *string-variable*. *numeric-variable* must be assigned a value prior to being used in a **QSTORE** statement.
string-variable	A quad precision string that will be converted to double precision in *numeric-variable*.

What It Does

Numeric-variable is set to the value of the first eight bytes of *string-variable*. If this number is outside the range -2,147,483,648 to +2,147,483,647, an arithmetic error results.

If the length of *string-variable* is less than eight bytes, then the string is treated as if it were padded with nulls.

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

Example

Refer to the example for **QDIV**.

QSUB

Statement and Command

Subtracts using quad precision.

AOS/VS	DG/RDOS	UNIX

Format

QSUB *string-variable1,string-variable2,string-variable3*

Arguments

string-variable1	The string variable receiving the result of subtracting *string-variable3* from *string-variable2*.
string-variable2	String variable from which *string-variable3* is subtracted.
string-variable3	A string variable used as the subtrahend.

What It Does

The **QSUB** statement (and all of the "Q" statements—**QLOAD**, **QSTORE**, **QADD**, **QMUL**, and **QDIV**) allows for the manipulation of very large numbers in the range -/+9,223,372,036,854,775,807 by operating on eight-byte (64-bit) binary integers stored in string operands. Numbers in this format are often referred to as quad precision since they occupy four words of space (see **QLOAD** and **QSTORE**).

The **QLOAD** and **QSTORE** statements allow for conversion of a normal four-byte Business BASIC number into and from the eight-byte quad precision number format.

How to Use It

All string variables must be loaded (using **QLOAD**) from numeric variables before the string variable can be used in a quad precision arithmetic statement.

The first eight bytes of the string indicated by *string-variable1* contain the result of subtracting the first eight bytes of *string-variable3* from *string-variable2*. Therefore,

```
QSUB A$,B$,C$
```

implies A$=B$–C$ and each string variable is eight bytes in length.

If any of these strings contain less than eight bytes, the string is treated as if it were padded with nulls.

continued **QSUB**

When the value of a string variable is written to a program's symbol table, all eight bytes are written. For this reason, all string variables used in a quad precision statement must be dimensioned to at least eight bytes. violating this rule can cause Business BASIC to write the value into the location of the string variable being written, as well as into the bytes adjacent to that location. This error could result in overwriting another variable in the symbol table and lead to undesirable results.

NOTE: Do not use any of the arithmetic string functions (such as **ASC**, **ASX**, **CHR$**, **CRM$**, **UCM$**) on a string variable to be used in a quad precision arithmetic statement.

Example

This example is a portion of a general ledger balance sheet allowing large totals. The code shown below illustrates adding of debits and subtracting of credits.

```
. . .
01370 QADD GTOTAL$,GTOTAL$,DEBIT$
01380 QSUB GTOTAL$,GTOTAL$,CREDIT$
. . .
```

RAISE

Statement

Forces an error.

AOS/VS	UNIX

Format

RAISE *err-type,error-number*

Arguments

err-type	A number or variable that represents one of the following values: 0 Business BASIC error 1 DG/RDOS error 2 AOS/VS error 3 UNIX error (UNIX systems only) For information about these errors, see *Using Business BASIC on DG/UX™ and INTERACTIVE UNIX Systems* or *Business BASIC System Managers Guide*.
error-number	A number or variable that represents the code for the error you want to generate. *error-number* must be coded as a positive number.

What It Does

The **RAISE** statement generates an error. If an **ON ERR** trap has been set, it is executed.

RAISE sets **SYS(20)**.

How to Use It

RAISE provides a way of generating errors from within a Business BASIC program. These errors can then be processed by an error routine.

You can use **RAISE** in conjunction with the error code functions **SYS(7)**, **SYS(31)**, **SYS(40)**, **SYS(41)**, **SYS(42)**, and **SYS(43)**.

Example

This example raises the error `FILE DOES NOT EXIST` on UNIX systems.

```
00010 LET UNIX=3
00020 LET FILE_DOES_NOT_EXIST=2
00030 RAISE UNIX,FILE_DOES_NOT_EXIST

* RUN
I/O Error 10 at 30 - File does not exist
```

RANDOMIZE

Statement and Command

Reseeds the random number generator.

AOS/VS	DG/RDOS	UNIX

Format

RANDOMIZE

What It Does

Normally, the **RND** function generates the same sequence of random numbers each time a program runs. **RANDOMIZE** reseeds the random number generator with a different base number according to the time of day, thereby producing different random numbers for each program run.

How to Use It

Start your program with a **RANDOMIZE** statement to initialize the random number generator, and then execute **RANDOMIZE** every time you want a different sequence of random numbers.

Example

Line 20 picks a number from 0 to 10. If the random number is 0, the program will stop.

```
00010  RANDOMIZE
00020  LET X=RND(11)
00030  PRINT X;
00040  IF X=0 THEN STOP
00050  GOTO 00020
* RUN
6  9  5  4  3  0
STOP AT 00040
```

READ

Statement

Assigns DATA statement values to variables.

AOS/VS	DG/RDOS	UNIX

Format

READ *variable*[,*variable*...]

Arguments

variable	A numeric or string variable, depending on the corresponding value in the **DATA** statement. The sequence of variables in **READ** must match the sequence of values in the data list formed by **DATA** statements.

What It Does

DATA statements store values in a data list before the program is executed. The **READ** statement retrieves these values. A data element pointer moves to the next value in the data list after a value is assigned to a **READ** variable. If the number of variables in **READ** exceeds the number of values in the data list, an error occurs. Since **DATA** values form a sequential data list, the first value in the data list is always the first value of the **DATA** statement with the lowest line number. If you want to restart the data list, use **RESTORE**.

How to Use It

Use **READ** only as a program statement. Normally, you place **READ** statements in a program at points where data is to be manipulated, whereas **DATA** statements can go anywhere. You can subscript the numeric and string variables in **READ**. These variables must match (numeric or string) the corresponding values in the **DATA** statement, or an error occurs. **READ** is executable, so if you want to start reading variables at a specific point in your program, place the **READ** statement there.

Example

Assign the values in the **DATA** statement to variables.

```
00010  DIM A$(10),ARA(12,12)
00020  READ X,Y
00030  READ A$,ARA(1,3)
00040  PRINT "X = ";X
00050  PRINT "Y = ";Y
00060  PRINT "A$ is: ";A$
00070  PRINT "ARA(1,3) = ";ARA(1,3)
00080  STOP
00090  DATA 32,46, "T.BONE",1400
* RUN
X = 32
Y = 46
A$ is: T.BONE
ARA(1,3) = 1400
STOP AT 00080
```

READ FILE

Statement and Command

Manages length-sensitive input.

AOS/VS	DG/RDOS	UNIX

Format

READ FILE (*channel*[,*byte-position*]),*variable*[,*variable*...]

Arguments

channel	A numeric expression for the channel number of a file opened for random or sequential access and input.
byte-position	A numeric expression or variable for the relative byte within a file to which you want to position the file pointer.
variable	A numeric or string variable that you can subscript; you can also repeat any combination. The size of *variable* determines the number of bytes input.

What It Does

The size of the variable determines the number of bytes of input. For each variable you supply in a **READ FILE** statement, a specific input occurs. If you have five variables in a **READ FILE**, five separate inputs occur in sequential order. For each input, **READ FILE** reads in the number of bytes needed to fill the variable and move the file pointer to where the next input will start.

How to Use It

You can read a file opened for sequential access from the beginning only. You can read a file opened for random access from any byte position in the file. If your file is a subfile and/or in linked-record format, then you have "records" that are fixed in length. A record is the result of a read or write. It is usually a fixed size in random files, and it must be a fixed size in linked available record files, subfiles, and index files. When records are fixed in length, you can easily determine which byte to position to: multiply the total number of bytes per record by the number of the record you want.

You can read part of a record or an entire record into one or more variables. Since each variable in a **READ FILE** statement is a specific input, you can avoid an interrupt that occurs between inputs of a single **READ FILE** by using one large variable in **READ FILE**. Use the **EOF** function to detect the end of the file.

The *byte-position* argument enables you to use **POSITION FILE** and **READ FILE** in a single program line. This argument is ignored on statements that perform terminal I/O via channel 16. If you specify the argument when the file has been opened in a mode that does not allow **POSITION FILE**, a runtime error is generated.

Examples

1. This program opens the file **JUNK**, and reads in four bytes for each variable. If an end of file occurs, control passes to line 200.

```
00010  OPEN FILE(1,3),"JUNK"
00020  READ FILE(1),A,B,C,D,E,F,G
00030  PRINT A,B,C,D,E,F,G
00040  IF EOF(1) THEN GOTO 00200
```

2. Line 50 reads the first two bytes into STAT%, the next six bytes into X# and the rest into RECORD$. If the status of the record is less than or equal to 0, control passes to line 200.

```
00010  DIM RECORD$(42)
00020  OPEN FILE(2,0),"FILE301"
00030  INPUT "RECORD?",R
00040  POSITION FILE (2,50*R)
00050  READ FILE (2),STAT%,X#,RECORD$
00060  IF STAT% <  = 0 THEN GOTO 00200
.  .  .
```

3. Read 50 bytes of input.

```
00010 DIM RECORD$(50)
00020 OPEN FILE (2,0),"DATA"
00030 INPUT "RECORD YOU WANT TO READ: ",R
00040 POSITION FILE (2,50*R)
00050 READ FILE (2),RECORD$
00060 LET STAT%=ASC(RECORD$(1,2))
00070 LET STAT% =OR(STAT% ,-AND(STAT%,32768))
00080 PRINT STAT%,RECORD$(3,50)...
```

4. Read 50 bytes of input, but do the position in the **READ FILE** statement.

```
00010 DIM RECORD$(50)
00020 OPEN FILE(2,0),"DATA"
00030 INPUT "RECORD YOU WANT TO READ: ",R
00040 READ FILE(2,50*R),RECORD$
00050 LET STAT%=ASC(RECORD$(1,2))
00060 LET STAT%=OR(STAT%,-AND(STAT%,32768))
00070 PRINT STAT%,RECORD$(3,50)
.  .  .
```

REM

Statement

Creates a remark (or comment) statement.

AOS/VS	DG/RDOS	UNIX

Format

REM *text of remark*

Arguments

text of remark	The remark you want to make.

What It Does

The text following **REM** is stored with the program and reproduced exactly when you list the program. Although **REM** statements are not executable, they do require storage space.

When you use a **GOTO** or **GOSUB** statement to refer to a line that is a **REM** statement, the text of the **REM** statement is also put as a comment next to the **GOTO** or **GOSUB** that calls it. Labeling of **GOTO** and **GOSUB** statements happens when you list the program, unless you suppress this feature using **STMA 6,6**. Only the target **REM** statement requires space for the comment.

How to Use It

Use **REM** for comments. You can save space at execution time by keeping another copy of the program as a source file. You can maintain this second file by using the **EDIT** utility's comment mode or by using the : (colon) convention in a source file. These comments are dropped when the source file is entered. They cannot be listed, but they do remain in the source file.

REM is a program statement only. Use it sparingly, and use it as the destination of a **GOSUB** or **GOTO**, if you want the **GOTO** or **GOSUB** statement labeled, as in the example.

Example

GOSUB and **GOTO** are automatically labeled when **REM** is the destination.

```
* LIST
 00010 GOSUB 01000 : *** PERFORM CALCULATIONS
 00020 GOTO 02000  : INSUFFICIENT FUNDS ROUTINE
  . . .
 01000 REM ***PERFORM CALCULATIONS
  . . .
 01050 RETURN
 02000 REM INSUFFICIENT FUNDS ROUTINE
  . . .
```

RENAME

Statement and Command

Changes the directory entry for a file.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{RENAME } [\textit{error-code,}] \left\{ \begin{array}{l} \textit{“oldname”,} \\ \textit{string-variable,} \end{array} \right\} \left\{ \begin{array}{l} \textit{“newname”} \\ \textit{string-variable} \end{array} \right\}$$

Arguments

error-code	An optional variable that receives any error code generated during the execution of the **RENAME** statement. This argument must be initialized (i.e., set to zero) before it is used.
oldname	A string literal within quotation marks naming a file that is in your directory or is a link entry.
newname	A string literal within quotation marks for the new filename you want to give the file.
string-variable	A string variable that represents a filename.

What It Does

RENAME looks for *oldname*, which is either a directory entry for a file in your directory or a link entry for a file in another directory, and changes it to *newname*. **RENAME** does not change the contents of the file.

How to Use It

If you use *string-variable*, you must dimension it and supply a filename's value.

Since a saved file has the program name stored internally, it is better to load a saved file and save it under the new name, rather than to just rename it from the CLI or using RENAME.

NOTE: The use of the *error-code* argument suppresses execution of Business BASIC's default error trap or any **ON ERR** condition (which causes the program to halt) and instead returns the same *error-code* as would be supplied by the appropriate **SYS** error function. You must check the value of *error-code* to determine whether an error has occurred in renaming a file.

Example

```
00010 DIM OLD$(10),IMPROVED$(10)
00020 LET OLD$="NAME"
00030 LET IMPROVED$ = "NAME1"
00040 RENAME OLD$,IMPROVED$
```

RENUMBER

Command

Renumbers lines in the current program.

AOS/VS	DG/RDOS	UNIX

Formats

RENUMBER [*line-number*] [STEP *increment*]

Arguments

line-number	The first statement in the program to be renumbered. The default is 10.
increment	The increment to be added to *line–number* when renumbering program statements. The default is 10.

What It Does

RENUMBER renumbers every statement in your program. It also renumbers the destination line numbers of **IF...THEN**, **GOTO**, **GOSUB**, and **ERASE** statements to coincide with the renumbered lines.

NOTE: If the target line of a **GOTO**, **GOSUB**, or **ERASE** statement does not exist, the **GOTO**, **GOSUB**, or **ERASE** argument is set to 00000 and `Error 53 – Renumbering error(s)` is generated.

How to Use It

If you execute a **RENUMBER** that exceeds the line number limit for your operating system, Business BASIC performs an automatic **RENUMBER 1 STEP 1**. See Appendix B for the maximum number of program lines allowed on your operating system.

You can use **RENUMBER** in the following ways:

RENUMBER
Starting at the first line (which will be assigned the line number 10), renumbers all lines in increments of 10.

RENUMBER *line-number*
Sets first line of your program equal to *line-number*, then renumbers all lines in increments of *line-number*.

RENUMBER STEP *increment*
Starting at the first line, renumbers all lines in increments of *increment*.

RENUMBER

continued

RENUMBER *line-number* **STEP** *increment*
Sets first line of program equal to *line-number*, then renumbers all lines in increments of *increment*.

Examples

1. Renumber using the default numbering system.

   *** RENUMBER**

```
00010  REM--THIS PROGRAM COMPUTES BALANCE
00020  INPUT "TYPE BALANCE:$",BAL
00030  IF BAL=0 THEN STOP
00040  INPUT "AMOUNT OF CHECK:$",CHK
00050  IF CHK=0 THEN STOP
00060  LET BAL=BAL-CHK
00070  PRINT "BALANCE IS:$",BAL
00080  GOTO 00040
```

2. Renumber beginning with 20.

   *** RENUMBER 20**

```
00020  REM--THIS PROGRAM COMPUTES BALANCE
00040  INPUT "TYPE BALANCE:$",BAL
00060  IF BAL=0 THEN STOP
00080  INPUT "AMOUNT OF CHECK:$",CHK
00100  IF CHK=0 THEN STOP
00120  LET BAL=BAL-CHK
00140  PRINT "BALANCE IS:$",BAL
00160  GOTO 00080
```

3. Renumber using increments of 5.

   *** RENUMBER STEP 5**

```
00010  REM--THIS PROGRAM COMPUTES BALANCE
00015  INPUT "TYPE BALANCE:$",BAL
00020  IF BAL=0 THEN STOP
00025  INPUT "AMOUNT OF CHECK:$",CHK
00030  IF CHK=0 THEN STOP
00035  LET BAL=BAL-CHK
00040  PRINT "BALANCE IS:$",BAL
00045  GOTO 00025
```

continued

RENUMBER

4. Renumber beginning at 5 with increments of five.

   *** RENUMBER 5 STEP 5**

```
00005  REM--THIS PROGRAM COMPUTES BALANCE
00010  INPUT "TYPE BALANCE:$",BAL
00015  IF BAL=0 THEN STOP
00020  INPUT "AMOUNT OF CHECK:$",CHK
00025  IF CHK=0 THEN STOP
00030  LET BAL=BAL-CHK
00035  PRINT "BALANCE IS:$",BAL
00040  GOTO 00020
```

REPLACE

Statement and Command

Replaces a saved program with a new program.

AOS/VS	**DG/RDOS**	**UNIX**

Format

AOS/VS and DG/RDOS:

$$\textbf{REPLACE}\left[\left\{\begin{array}{l}\textit{"filename"}\\ \textit{string-variable}\end{array}\right\}\right]$$

UNIX:

$$\textbf{REPLACE}\ [\ \textit{byte-format},\ ]\left[\left\{\begin{array}{l}\textit{"filename"}\\ \textit{string-variable}\end{array}\right\}\right]$$

Arguments

filename	The filename of a program you want to replace. If not specified, Business BASIC uses the filename last loaded or run.
string-variable	A string variable already dimensioned and assigned a filename value.
byte-format	Either 0 or 1 to indicate whether you want to save the file in DG/UX format (0) or INTERACTIVE UNIX format (1). The default for a file that you loaded using **LOAD** is that file's current SAVE file format. The default for a file that you created while in Business BASIC input mode or entered using **ENTER** into Business BASIC is the native format for the operating system.

What It Does

REPLACE performs a SAVE on the program you have in working storage and uses it to replace the program specified by *filename* or the current program name if you do not specify *filename*. If a file exists with the name *filename*, **REPLACE** overwrites that file. If you did not load or run the program, and you do not specify a filename, the new program is saved with the filename **SCRATCH**. The new program does not have to be saved before you replace it.

NOTE: Do not enter **REPLACE** "*filename*" if the file being replaced is currently stored or listed in ASCII source format. This causes the file to reformat in a binary Business BASIC SAVE file format. Instead, you can use **REPLACE** with a new filename to save the file in binary format or **LIST** with a new filename to save a source listing of the new program.

How to Use It

Use **REPLACE** as a keyboard mode command or a program statement. Either way, it saves the current program. If you load or run a program so that it is in working storage, then you can update the program and replace the old program without having to specify *filename*. If you do not load or run an existing program, but create a new one in working storage, you can replace any program with this new program by specifying the old program's name as *filename* in a **REPLACE** statement.

A program's filename is carried internally within the SAVE file. If you rename a saved file, the old name still exists within the SAVE file. Therefore, if you load and then replace a renamed SAVE file without giving a filename, Business BASIC saves the program under its old filename.

The complete pathname is not stored within a SAVE file; so if you execute a **REPLACE** without supplying a pathname, Business BASIC saves the file in your current directory.

On UNIX systems, you can use the optional *byte–format* argument to specify whether the SAVE file format is the DG/UX format or the INTERACTIVE UNIX format. (DG/UX systems and INTERACTIVE UNIX systems differ with respect to the ordering of bytes within a word.) Enter 0 when you want to use the native SAVE file format for DG/UX systems and 1 when you want to use the native SAVE file format for INTERACTIVE UNIX systems.

Examples

1. If you load or run a program, and then modify it, this statement saves the modified one under the old name. If you create a new program in working storage, this saves it under the name **SCRATCH**.

   ```
   * REPLACE
   ```

2. This command replaces old **PROG3** with the current program.

   ```
   * REPLACE "PROG3"
   ```

3. This statement replaces **NEWPROG** with the current program.

   ```
   00900  LET NAME$="NEWPROG"
   00910  REPLACE NAME$
   ```

RESTORE

Statement and Command

Resets the list pointer for a DATA statement.

AOS/VS	DG/RDOS	UNIX

Format

RESTORE [*line-number*]

Arguments

line-number	Any valid line number in your program.

What It Does

RESTORE without *line-number* resets the pointer to the beginning of the data list. The first element in the data list is the first value in the **DATA** statement with the lowest line number. **RESTORE** with *line-number* resets the pointer to the first value in the **DATA** statement specified by *line-number*. If you specify a line number that does not exist in your program or is not a **DATA** statement, **RESTORE** will find the next **DATA** statement following *line-number* and reset the pointer to the first value in it.

How to Use It

When you want a **READ** statement to go back to the beginning of a **DATA** statement to pick up values for its variables, use **RESTORE**. You can use **RESTORE** as a program statement or a keyboard command. If you want to reuse the entire data list, do not specify *line-number*. If you want to reuse the values in a particular **DATA** statement, specify the **DATA** statement by its line number.

If you include *line-number*, **RESTORE** resets the data list pointer to the first value for the **DATA** statement at that line number. See Appendix B for the range of line numbers allowed on your operating system.

Example

This example includes a **RESTORE** to a line number and a **RESTORE** using the default.

```
00005 READ A,B,C
00010 READ D,E,F
00015 RESTORE 00040
00020 READ G,H,I
00025 RESTORE
00030 READ J,K,L
00035 DATA 2,4,6
00040 DATA 8,10,12
```

How to Use It

Use **REPLACE** as a keyboard mode command or a program statement. Either way, it saves the current program. If you load or run a program so that it is in working storage, then you can update the program and replace the old program without having to specify *filename*. If you do not load or run an existing program, but create a new one in working storage, you can replace any program with this new program by specifying the old program's name as *filename* in a **REPLACE** statement.

A program's filename is carried internally within the SAVE file. If you rename a saved file, the old name still exists within the SAVE file. Therefore, if you load and then replace a renamed SAVE file without giving a filename, Business BASIC saves the program under its old filename.

The complete pathname is not stored within a SAVE file; so if you execute a **REPLACE** without supplying a pathname, Business BASIC saves the file in your current directory.

On UNIX systems, you can use the optional *byte-format* argument to specify whether the SAVE file format is the DG/UX format or the INTERACTIVE UNIX format. (DG/UX systems and INTERACTIVE UNIX systems differ with respect to the ordering of bytes within a word.) Enter 0 when you want to use the native SAVE file format for DG/UX systems and 1 when you want to use the native SAVE file format for INTERACTIVE UNIX systems.

Examples

1. If you load or run a program, and then modify it, this statement saves the modified one under the old name. If you create a new program in working storage, this saves it under the name **SCRATCH**.

   ```
   * REPLACE
   ```

2. This command replaces old **PROG3** with the current program.

   ```
   * REPLACE "PROG3"
   ```

3. This statement replaces **NEWPROG** with the current program.

   ```
   00900  LET NAME$="NEWPROG"
   00910  REPLACE NAME$
   ```

RESTORE

Statement and Command

Resets the list pointer for a DATA statement.

AOS/VS	DG/RDOS	UNIX

Format

RESTORE [*line-number*]

Arguments

line-number	Any valid line number in your program.

What It Does

RESTORE without *line-number* resets the pointer to the beginning of the data list. The first element in the data list is the first value in the **DATA** statement with the lowest line number. **RESTORE** with *line-number* resets the pointer to the first value in the **DATA** statement specified by *line-number*. If you specify a line number that does not exist in your program or is not a **DATA** statement, **RESTORE** will find the next **DATA** statement following *line-number* and reset the pointer to the first value in it.

How to Use It

When you want a **READ** statement to go back to the beginning of a **DATA** statement to pick up values for its variables, use **RESTORE**. You can use **RESTORE** as a program statement or a keyboard command. If you want to reuse the entire data list, do not specify *line-number*. If you want to reuse the values in a particular **DATA** statement, specify the **DATA** statement by its line number.

If you include *line-number*, **RESTORE** resets the data list pointer to the first value for the **DATA** statement at that line number. See Appendix B for the range of line numbers allowed on your operating system.

Example

This example includes a **RESTORE** to a line number and a **RESTORE** using the default.

```
00005 READ A,B,C
00010 READ D,E,F
00015 RESTORE 00040
00020 READ G,H,I
00025 RESTORE
00030 READ J,K,L
00035 DATA 2,4,6
00040 DATA 8,10,12
```

In the example, the variables are assigned values as follows:

Variable	Value
A	2
B	4
C	6
D	8
E	10
F	12
G	8
H	10
I	12
J	2
K	4
L	6

RETURN *Statement*

Transfers control from a subroutine back to its calling point.

AOS/VS	DG/RDOS	UNIX

Format

RETURN

What It Does

RETURN transfers control to the line following the last **GOSUB** executed. Since **GOSUB** transfers control to a subroutine, a **RETURN** statement anywhere within that subroutine returns control to the line following that **GOSUB**. You can have more than one **RETURN** in a subroutine. For more information, see **GOSUB...RETURN**.

RFORM

Statement

Defines a record string format.

AOS/VS	DG/RDOS	UNIX

Format

RFORM *format-string*

Arguments

format-string	A string literal without quotation marks that contains format information describing the composition of a string variable.

What It Does

RFORM allows you to define record formatting information outside the **PACK** and **UNPACK** statements. With **RFORM**, you can also define a record's format at a single location and refer to it in multiple **PACK** and **UNPACK** statements.

How to Use It

The format string is composed of the characters listed below. Embedded spaces and commas are allowed.

Format Item	**Field Type Defined**
B,C,D,E,F	Defines a 1-, 2-, 3-, 4-, or 5-byte unsigned integer field respectively. In AOS/VS Business BASIC, the F format is valid in triple precision only. In AOS/VS and DG/RDOS Business BASIC, the E and F formats are valid in triple precision only.
G	Defines a 6-byte unsigned integer field (valid only in quad precision).
H	Defines a 7-byte unsigned integer field (valid only in quad precision).
I,J,K,L,M,N	Defines a 1-, 2-, 3-, 4-, 5-, or 6-byte signed integer field respectively. M and N formats are valid in triple precision only.
O	Defines a 7-byte signed integer field (valid only in quad precision).
P	Defines a 8-byte signed integer field (valid only in quad precision).
S*n*	Defines an *n*-byte string field. Trailing or embedded nulls are significant and are transferred within the field.
A*n*	Defines an *n*-byte string field. (For **UNPACK** statements only, trailing or embedded nulls are used as terminators of the field transfer.)

RFORM

Format Item	Field Type Defined
+*n*	Skips *n* bytes of the record string.
@*n*	Position to byte *n* of the record string before the next action. The bytes of the record string are numbered from 1.
**n*	Repeat the immediately preceding format (B, C, D, I, J, K, or L) *n* times for a numeric array. This transfers successive elements of the array beginning with the element specified in the variable list. The number of elements transferred must be within the dimensioned bounds of the array.
U*n*	Defines a one-bit field in a numeric variable or array element. You must specify a separate **U***n* argument for each bit you want to pack or unpack. The bit indicated by 2^n ($0<=n<=7$) is set by **PACK** when the source value is non-zero or is tested by **UNPACK** placing 0 (cleared) or 1 (set) into the destination field. NOTE: The pointer into the record string is not moved, thus allowing multiple fields to correspond to a single byte.
Z*n*	Fill the record string with nulls and set current length to *n* on **PACK**. This format item is ignored by **UNPACK**. It is only valid as the first descriptor of a format string.

NOTE: The format elements were chosen to keep the string as short as possible to reduce the memory requirements. The following defaults were chosen for *n* on those elements used frequently:

Element	Default Elements
U	0
S,A,+,@,*	1
Z	unlimited (max. length of string)

The format elements must match the expression or variable list in the **PACK** or **UNPACK** statement in data type and number of items. An example using the **RFORM** and **PACK** statements is presented in the **PACK** description.

On AOS/VS and UNIX systems, you can repeat any format argument by putting the argument in parentheses and specifying outside the parentheses the number of times to repeat the argument—for example, **3(A5)** repeats the **A***n* argument three times.

continued **RFORM**

Examples

1. **RFORM** is used here to format a record.

```
:Compose and write a record string containing record  status, name
:address, city, state, zip code, balance, and age.
. . .
00220 RFORM ZJA20A30A12A2DLB
:           |||  |  |  | ||*1-byte unsigned integer for AGE%
:           |||  |  |  | |*4-byte signed integer for BAL
:           |||  |  |  | *3-byte unsigned integer for ZIP
:           |||  |  |  *2-byte truncated string for STTE$
:           |||  |  *12-byte truncated string for CITY$
:           |||  *30-byte truncated string for ADR$
:           ||*20-byte truncated string for NAME$
:           |*2-byte signed integer for STAT%
:           *null fill CUSREC$
. . .
00870 PACK 00220,CUSREC$,STAT%,NAME$,ADR$,CITY$,STTE$,ZIP,BAL,AGE%
00880 LWRITE FILE[4,record-number],CUSREC$
```

2. This example reads and decodes an employee record into fields: employee number, pay rate, overtime rate, deduction array, and tax array.

```
00320 RFORM JLL+8L*10@179L*6
:           |||| |    |  *6 4-byte signed integer elements
:           |||| |    |   into TAXES
:           |||| |    *pick next field starting with byte 179
:           |||| *10 4-byte signed integer elements into DEDNS
:           |||*skip 8 bytes of record string
:           ||*4-byte signed integer for OTRATE
:           |*4-byte signed integer for REGRAT
:           *2-byte signed integer for EMPNO
. . .
01100 LREAD FILE[5,EMPRNO],EMPREC$
01110 UNPACK 00320,EMPREC$,EMPNO,REGRAT,OTRATE,DEDNS[1],TAXES
:
:The array TAXES must have already been dimensioned
```

RFORM

continued

3. This program sets bits 0, 2, 5, and 7 in a **PACK** statement. The result of the **PACK** statement is that the ASCII value of RECORD$ is 165. The **UNPACK** statement is used to see which bits are set.

```
00010 DIM RECORD$[1]
00020 LET RECORD$=FILL$(0)
00030 RFORM U0U2U5U7
00040 RFORM U0U1U2U3U4U5U6U7
00050 PACK 00030,RECORD$,1,1,1,1
00060 UNPACK 00040,RECORD$,BIT0,BIT1,BIT2,BIT3,BIT4,BIT5,BIT6,BIT7
00070 PRINT BIT0;BIT1;BIT2;BIT3;BIT4;BIT5;BIT6;BIT7
```

```
* RUN
 1  0  1  0  0  1  0  1
```

RND

Function

Produces a pseudo-random integer.

AOS/VS	DG/RDOS	UNIX

Format

RND(*expression*)

Arguments

expression	A numeric expression or variable that is one greater than the highest integer you want the **RND** function to generate. *expression* sets the ceiling for randomly generated integers. It must be greater than or equal to 0 and less than or equal to 65535.

What It Does

Each time you use the **RND** function, it produces an integer *n* such that 0 <= *n* < *expression*. If you do not use a **RANDOMIZE** statement at the beginning of your program, you get the same sequence of random numbers each time you run the program. This sequence of numbers is different under AOS/VS and UNIX. **RANDOMIZE** causes the starting point in the set of random numbers to change. The **RND** function produces one set of random numbers on AOS/VS and DG/RDOS systems, a second set on DG/UX systems, and a third set on INTERACTIVE UNIX systems.

How to Use It

RND is a numeric expression and can be used wherever numeric expressions are allowed. If you want to repeat a series of random numbers, do not use **RANDOMIZE**.

Examples

RND is demonstrated with and without **RANDOMIZE**.

```
00010 FOR I = 1 TO 4
00020 PRINT RND(100)
00030 NEXT I
* RUN
4
14
15
85
```

RND *continued*

If you run the same program again, you get the same series of random numbers.

```
* RUN
4
14
15
85
```

Now add a **RANDOMIZE** statement.

```
00005 RANDOMIZE
00010 FOR I = 1 TO 4
00020 PRINT RND(100)
00030 NEXT I
* RUN
43
86
14
23
```

RANDOMIZE results in a different random series for each **RUN** statement.

```
* RUN
45
19
82
37
```

RUN

Command

Executes a program.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{RUN} \left[\left\{ \begin{array}{l} \textit{line-number} \\ \textit{"filename"} \end{array} \right\} \right]$$

Arguments

line-number	A statement line number of the program in working storage where execution begins.
filename	A literal filename within quotation marks for a program file that has been saved or replaced and that you want to load and execute. Do not use *line-number* with *filename*. You can prefix the *filename* argument with a "#" to reference a program from the system library (BASIC.PL). In addition, you can access a program from the user's program library (if it has been established via **STMA 20**) by prefixing the *"filename"* argument with "%".

What It Does

The variations of the **RUN** command are below.

RUN
RUN clears all variables, undimensions all arrays and strings, executes a **RESTORE** statement, initializes the random number generator, and then executes the current program from the first line number. It does not close or unlock files, nor does it move the file pointer.

RUN *line-number*
Retains current values for variables, maintains the positions of file pointers, keeps locks active, and executes the current program beginning at *line-number*; it does not perform a **RESTORE** statement. This form is equivalent to CON if *line-number* is the line following the last point of interruption.

RUN *"filename"*
Searches the current directory for the file specified in *filename*; if not found, it searches the library directory for *filename*. It also clears working storage, brings *filename* into working storage, and then executes *filename* from the first line. You can specify only a saved program in **RUN** *"filename"*, not a listed program. If you try to run a listed file, Business BASIC generates `Error 48 - Not a save file`.

RUN

continued

How to Use It

You must type **RUN** with *"filename"* to run a program; otherwise, *"filename"* by itself performs a **SWAP**. Use the **RUN** *line-number* form to help debug your programs.

Examples

1. This executes the program currently in working storage, from the first line.

   ```
   * RUN
   ```

2. This clears working storage, brings **NEWPROG** into working storage, and then executes **NEWPROG** from the first line.

   ```
   * RUN "NEWPROG"
   ```

3. This executes a program that contains year–end statistics. The program, called **YREND**, has been placed in the system library BASIC.PL.

   ```
   * RUN "#YREND"
   ```

SAVE

Statement and Command

Saves a program and its variables.

AOS/VS	DG/RDOS	UNIX

Format

AOS/VS and DG/RDOS:

$$\text{SAVE}\left\{\begin{array}{l}\textit{"filename"}\\ \textit{string-variable}\end{array}\right\}$$

UNIX:

$$\text{SAVE}\ [\ \textit{byte-format,}\ ]\left\{\begin{array}{l}\textit{"filename"}\\ \textit{string-variable}\end{array}\right\}$$

Arguments

filename	A filename (in quotation marks) for your program file. It should be a new filename; if *filename* already exists, you get an error message.
string-variable	A string variable that you have already dimensioned and assigned a filename.
byte-format	Either 0 or 1 to indicate whether you want to save the file in DG/UX format (0) or INTERACTIVE UNIX format (1). The default for a file that you loaded using **LOAD** is that file's current SAVE file format. The default for a file that you created while in Business BASIC input mode or entered into Business BASIC using **ENTER** is the native format for the operating system.

What It Does

SAVE writes the contents of working storage (statements and values for variables) in internal code to a disk file it creates using the name *filename*. If *filename* already exists in your directory, you get an error message. *filename* is stored internally in the program file, so if you want to rename your program file, save the program under a new name. **SAVE** does not preserve the file status of files in your program.

How to Use It

You must supply *filename* either in quotation marks or as a string variable that holds a value of *filename*. To maintain compatibility across operating systems, limit the filename to a maximum of 13 characters. When you save a program, the current values of all variables are stored with the program as well as the location where the program last stopped.

SAVE

continued

Saved programs can be chained to, swapped to, run, and replaced. You can also save a program that has partially executed, and later resume execution of the program by using a **CON** or **RUN** *line-number* statement. The program retains the values that the variables had when you saved it.

To replace a saved program with a new one (perhaps an edited version of the old), use **REPLACE.** (**RENAME** does not change the internal name of the program file.) Load your program, edit it, then use **REPLACE** (without a filename if you want to keep the same name).

On UNIX systems, you can use the optional *byte-format* argument to specify whether the SAVE file format is the DG/UX format or the INTERACTIVE UNIX format. (DG/UX systems and INTERACTIVE UNIX systems differ with respect to the ordering of bytes within a word.) Enter 0 to use the native SAVE file format for DG/UX systems and 1 to use the native SAVE file format for INTERACTIVE UNIX systems.

Examples

1. Save the following statements and values in a program file called **PROG1.**

```
* NEW
* 10 LET X=1
* 20 LET Y=2
* RUN

* SAVE"PROG1"
* NEW
* LOAD"PROG1"

* PRINT X;Y 1  2
```

2. Lines can be added to the file **PROG1**, and the program can be replaced to retain the same name.

```
* LOAD"PROG1"

* LIST
00010 LET X=1
00020 LET Y=2

* 30 PRINT X;Y
* SAVE"PROG1"
I/O ERROR 9 - File Already Exists
* REPLACE
* NEW
* LOAD "PROG1"
* LIST
00010 LET X=1
00020 LET Y=2
00030 PRINT X;Y


*
```

SCANUNTIL

Statement

Scans a string until characters in a substring are found.

AOS/VS	DG/RDOS	UNIX

Format

SCANUNTIL *position,string-variable1,string-variable2* [*,start*]

Arguments

position	Numeric variable that receives a value representing the string position of the last character in *string-variable1* that was not in the set of characters in *string-variable2*.
string-variable1	String expression to be scanned until a character in it matches a character in *string-variable2*.
string-variable2	String expression containing the set of characters to be searched for in *string-variable1*.
start	Optional numeric expression indicating the location in *string-variable1* to begin scanning. *start* must be assigned a value of at least 1 prior to statement execution.

What It Does

SCANUNTIL lets you locate a specified substring of characters in a string. It returns location in the string of the last character before it finds the substring.

String-variable1 is scanned beginning at relative position *start,* if you include this argument. *Position* is incremented as a relative pointer until a character in *string-variable2* is found. The location of the last character in *string-variable1* that is not in *string-variable2* is returned in *position*.

Start is an optional argument; if you do not specify *start*, the scan starts at the first character. If *start* is greater than the length of *string-variable1*, then *position* is set to 0. If the pointer reaches the end of *string-variable1* before a match is found in *string-variable2*, *position* is set to the length of *string-variable1*. SCANUNTIL checks to see if *string-variable1* contains any one element of *string-variable2*, not the entire substring.

How to Use It

Enter SCANUNTIL as a statement. You must supply values for the arguments *position*, *string–variable1*, and *string–variable2*. If you use the *start* argument, you must supply it with a value of at least 1 prior to executing SCANUNTIL.

SCANUNTIL *continued*

Examples

1. Find the position of the last character in A$ that occurs before a match with B$. Begin scanning A$ at SLOC.

```
00010 DIM A$[100],B$[100]
00020 LET P=0
00025 PRINT "INPUT VALUES FOR A$, B$, AND START LOCATION, PRESS
ESC TO EXIT"
00030 INPUT A$,B$,"      START AT: ",SLOC;
00040 SCANUNTIL P,A$,B$,SLOC
00050 PRINT ," P=";P
00060 GOTO 00030

* RUN
INPUT VALUES FOR A$, B$, AND START LOCATION, PRESS ESC TO EXIT
? ABCDEFGHIJK    ? XYZ    START AT: 1         P= 11
? ABCDE          ? XYZ    START AT: 8         P= 0
? ABCDEFGHI      ? XYFG   START AT: 1         P= 5
? ABCEFGHI       ? ABC    START AT: 1         P= 0
?
IKEY AT 00030
*
```

2. Scan a string and remove characters.

```
* LIST
00010 DIM DATE$[8],DELM$[3],X$[6],Y$[2],NUM$[10]
00020 LET DELM$="/ -" \ NUM$="0123456789"
00030 LET P=0 \ SLOC=1 \ X$=""
00040 INPUT USING "","DATE: ",DATE$
00050 SCANUNTIL P,DATE$,DELM$,SLOC
00060 IF P=0 THEN GOTO 00120
00070 LET Y$[1,2]=DATE$[SLOC,P]
00080 IF SLOC=P THEN LET Y$[1,2]="0",DATE$[SLOC,P]
00090 LET X$=X$,Y$
00100 LET SLOC=P+2
00110 GOTO 00050
00120 PRINT "DATE WITHOUT DELIMITERS = ",X$
00130 LET P=0
00140 SCANWHILE P,X$,NUM$
00150 IF P=6 THEN GOTO 00180
00160 PRINT "INVALID DATE - ENTER AGAIN"
00170 GOTO 00030
00180 END

* RUN
DATE: 12/31/91
DATE WITHOUT DELIMITERS =  123191
```

```
* RUN
DATE: 1-A-91
DATE WITHOUT DELIMITERS =  01A91
INVALID DATE - ENTER AGAIN
DATE: 1 1 91
DATE WITHOUT DELIMITERS = 010191


*
```

SCANWHILE

Statement

Scans a string while characters match those in a substring.

AOS/VS	DG/RDOS	UNIX

Format

SCANWHILE *position,string-variable1,string-variable2*[*,start*]

Arguments

position	Numeric variable that receives a value representing the string position of the last character in *string-variable1* that is also in the set of characters in *string-variable2*.
string-variable1	String expression to be scanned while its characters match characters in *string-variable2*.
string-variable2	String expression containing the set of characters to be passed over in *string-variable1*.
start	Optional numeric expression indicating the location in *string-variable1* to begin scanning. *start* must be assigned a value of at least 1 prior to statement execution.

What It Does

SCANWHILE scans the string you supply as long as the characters in it match the characters in the substring you supply. It returns location in the last character in the string that is also in the substring.

String-variable1 is scanned beginning at relative position *start,* if you include this argument. *Position* is incremented as a relative pointer. The character pointed to in *string-variable1* is in the set indicated by *string-variable2*. Scanning stops when the first non-matching character is found, regardless of whether later characters in *string-variable1* match those of *string-variable2*. The location of the last character in *string-variable1* that is also in *string-variable2* is returned in *position.*

Start is an optional argument; if you do not specify *start,* the scan starts at the first character. If *start* is greater than the length of *string-variable1,* then *position* is set to 0. If the pointer reaches the end of *string-variable1* before a non–matching character is found in *string-variable2, position* is set to the length of *string-variable1*.

How to Use It

Enter **SCANWHILE** as a statement. You must supply values for the arguments *position*, *string–variable1*, and *string–variable2*. If you use the *start* argument, you must supply it with a value of at least 1 prior to executing **SCANWHILE**.

Examples

1. Find the position of the last occurrence of any character in A$ that occurs in B$. Begin scanning A$ at SLOC.

```
00010 DIM A$[100],B$[100]
00020 LET P=0
00025 PRINT "INPUT VALUES FOR A$, B$, AND START LOCATION, PRESS
          ESC TO EXIT"
00030 INPUT A$,B$,"       START AT: ",SLOC;
00040 SCANWHILE P,A$,B$,SLOC
00050 PRINT ," P=";P
00060 GOTO 00030

* RUN
INPUT VALUES FOR A$, B$, AND START LOCATION, PRESS ESC TO EXIT
? ABBCCCDDDDEEEEE        ? ABCE   START AT: 1        P= 6
? 12345.65          ? 0123456789   START AT: 1        P= 5
? ABCDE             ? ABCDEF       START AT: 8        P= 0
? ABCDE             ? ABCDEF       START AT: 1        P= 5
?
IKEY AT 00030
*
```

2. Allow only selected characters in a date string.

```
* LIST
00010 DIM DATE$[8],NUM$[13]
00020 LET NUM$="/- 0123456789"
00030 LET P=0 \ SLOC=1
00040 INPUT USING "","DATE: ",DATE$
00050 SCANWHILE P,DATE$,NUM$,SLOC
00060 IF P=LEN(DATE$) THEN GOTO 00090
00070 PRINT "INVALID DATE - ENTER AGAIN"
00080 GOTO 00030
00090 PRINT "THIS DATE DOES NOT CONTAIN INVALID CHARACTERS"
00100 END

* RUN
DATE: 1/2/91
THIS DATE DOES NOT CONTAIN INVALID CHARACTERS

* RUN
DATE: 1-A-91
INVALID DATE - ENTER AGAIN
DATE: 1 22 91
THIS DATE DOES NOT CONTAIN INVALID CHARACTERS


*
```

SGN

Function

Determines the sign (positive or negative) of an expression.

AOS/VS	DG/RDOS	UNIX

Format

SGN(*expression*)

Arguments

expression	A numeric expression or variable for the number whose sign you want to know.

What It Does

SGN returns a value of 1 if *expression* is positive, −1 if *expression* is negative, and 0 if *expression* is 0.

How to Use It

Use SGN as a numeric expression wherever numeric expressions are allowed.

Example

Determine the sign of X.

```
* LIST
00010 INPUT X
00020 IF SGN(X)=-1 THEN PRINT "NEGATIVE"
00030 IF SGN(X)=1 THEN PRINT "POSITIVE"
00040 IF SGN(X)=0 THEN PRINT "EQUALS ZERO"

* RUN
?-20
NEGATIVE

* RUN
?0
EQUALS ZERO
```

SHELL

Statement and Command

Calls the UNIX shell.

UNIX

Format

SHELL [*error-variable,command* [, *buffer*]]

Arguments

error-variable	A variable that receives an error code if an error occurs during creation of the shell process.
command	A string variable or string literal containing the command you want to execute from the UNIX shell and any options you want to specify for the command.
buffer	A string variable that receives the output (both standard output and standard error) of the **SHELL** command.

What It Does

The **SHELL** statement calls either the Bourne shell or the C shell, depending on the value in the optional environment variable BBSHELL. To specify the BOURNE shell, BBSHELL should contain `/bin/sh`. If BBSHELL is not set, **SHELL** calls your initial shell program. If you specify a command, that command is executed and you return to the Business BASIC prompt. Otherwise, you remain in the shell. You cannot pass information from the UNIX shell back to Business BASIC.

If a buffer string variable is provided, the command's output is written to the buffer; otherwise, the output is written to the screen.

How to Use It

SHELL assumes that the directory **/bin** is on the search path defined by your PATH variable. Also, you must have started Business BASIC with the **–c** option; otherwise, **SHELL** returns without executing any commands.

Error–variable must be a numeric variable and must be initialized. This argument receives an exceptional status code if the UNIX **fork** or **exec** call fails, a zero if the **-c** option was not specified, or a -1 if the command was successful. The exceptional status code is the value the shell returns. For a list of UNIX error codes and messages, see *Using Business BASIC on DG/UX™ and INTERACTIVE UNIX Systems.*

The command line you specify using the **SHELL** statement must be in shell format instead of Business BASIC CLI format. You must terminate the string variable or literal with a null character. If you omit the null character, Business BASIC displays `Error 34 - Function argument`.

SHELL

When you include the *buffer* variable, make sure it is large enough to hold all the output from the command. If *buffer* is too small, the command's output is truncated to the dimensioned length of *buffer* and the value returned in the error variable is unpredictable.

If you omit the arguments, **SHELL** creates a subordinate shell, from which you can execute any number of shell commands. To return to the Business BASIC prompt, log out of the shell as you normally would.

Examples

1. This example executes one command at the shell and then returns you to the Business BASIC prompt.

```
00010  DIM A$(20)
00020  ER=0
00030  A$="pwd<0>"
00040  SHELL ER,A$

* RUN
/usr/bbusers/debbie/programs
*
```

2. This example creates a subordinate shell process and starts a **vi** editing session.

```
* SHELL
% vi
 .
 .
 .
% logout
*
```

3. This example executes two commands at the shell and then returns you to the Business BASIC prompt. The output of the commands is written to **BUF$**.

```
00010  DIM BUF$(1024)
00020  ER=0
00030  SHELL ER, "cd;pwd<0>",BUF$
```

SHFT

Function

Performs a shift left or right with zero fill.

AOS/VS	DG/RDOS	UNIX

Format

SHFT(*expression,bits*)

Arguments

expression	A numeric expression or variable; the number you want to shift.
bits	The number of bits you want to shift. A negative (*–bits*) shifts *bits* number to the right (less), and a positive (*+bits*) shifts *bits* number to the left (greater).

What It Does

SHFT moves all of the bits of *expression*, whether it is single, double, triple, or quadruple precision. **SHFT** with a negative number in *bits* moves the expression *bits* number of bits to the right. **SHFT** with a positive number in *bits* shifts the expression *bits* number of bits to the left. **SHFT** with a zero in *bits* does nothing.

How to Use It

Use **SHFT** as a numeric expression, wherever numeric expressions are allowed.

Figure 1–9 shows the bit positions that result when an expression is shifted using SHFT.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Power of 2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X = 32768, or 2^{15}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SHFT(X, –1) = 2^{14}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SHFT(X, +2) = 2^{16}

Figure 1–9 The SHFT Function

SHFT

continued

Example

This program displays the system status.

```
00010 DIM D$[40]
00020 LET STAT=SYS(30)
00030 FOR I=15 TO 0 STEP -1
00040   READ D$
00050   IF AND(STAT,1) THEN PRINT "Bit";I;"means ";D$
00060   LET STAT=SHFT(STAT,-1)
00070 NEXT I
00080 DATA "Double Precision","Triple Precision","Quad Precision"
00090 DATA "UNIX","N/A","RDOS","AOS","N/A"
00100 DATA "AOS/VS","Single-user system","N/A","Virtual console"
00110 DATA "N/A","N/A","N/A","N/A"
```

```
* RUN
Bit 15 means Double Precision
Bit 9 means AOS

*
```

SIZE

Command

Determines the size of the program in working storage.

AOS/VS	DG/RDOS	UNIX

Format

SIZE

What It Does

SIZE tells you how many bytes of working storage the current program occupies and how many bytes are still available. Both values are decimal.

For UNIX systems executing in DG mode, **SIZE** displays output similar to the output it displays for AOS/VS. For UNIX systems executing in non-DG mode, **SIZE** displays its output in a window on the right side of the screen. The window occupies rows 2-13 and columns 45-80. The program name, the sizes of the program and data segments, the total space used, and the space remaining are shown.

How to Use It

SIZE is used in keyboard mode.

Example

1. In this DG/RDOS example, the current program and values held by variables occupy 6700 bytes of working storage; 8077 bytes remain.

```
* SIZE
USED: 6700 BYTES
LEFT: 8077 BYTES
```

2. In this AOS/VS example, the current program and data occupy 2536 bytes; 259234 bytes remain.

```
* SIZE
Used:    2400 bytes (Program)
          136 bytes (Data)
         2536 bytes (Total)

Left:    128534 bytes (Program)
         130700 bytes (Data)
         259234 bytes (Total)
```

SIZE

continued

3. In this example on a UNIX system executing in non–DG mode, **PROG1** is loaded into working storage and the **SIZE** command is issued before the program is executed.

```
* load "PROG1
* size

                      ──── Program/Data Size Information ────
                        Program Name   :  PROG1
                        Maximum size   :  524288 bytes
                        Used
                          Program size :  380 bytes
                          Data size    :  1720 bytes
                          Total size   :  2100 bytes
                        Remaining      :  522188 bytes

                        ■



                      ──────── Press any key to continue ────
```

SQR

Function

Square root of an expression (truncated to an integer).

AOS/VS	DG/RDOS	UNIX

Format

SQR(*expression*)

Arguments

expression	The numeric expression or variable whose square root you want to find.

What It Does

SQR finds the square root of *expression*. It truncates any fraction, making the value an integer.

How to Use It

Use **SQR** as a numeric expression, wherever numeric expressions are allowed. If you want more precision, multiply the expression by a suitable power of 10 (see example).

Example

Print the square root of two and the square root of two to the third decimal place.

```
* LIST
00010 LET X=2
00020 PRINT SQR(X)
00030 PRINT USING "D5.3",SQR(X*(10^3)^2)

* RUN
1
1.414

*
```

STEP

Command

Executes the next statement of the program currently in memory.

AOS/VS	UNIX

Format

STEP [*number*]

Arguments

number	The number of statements to execute with this **STEP** command.

What It Does

The **STEP** command is a debugging tool. When you issue this command, Business BASIC executes the next statement of the program currently in memory. Using the optional *number* argument, you can request that more than one statement be executed. The default is one statement. As **STEP** executes a line, it displays the line number in brackets.

If Business BASIC reaches the end of the program before finishing the execution of a **STEP** command, it displays the remaining lines and then displays the message END OF PROGRAM. A **STEP** command with an argument of 0 produces no output but causes the line counter to be reset to the start of the program currently in memory.

How to Use It

Issue the **STEP** command from keyboard mode.

STEP is always available on UNIX systems. On AOS/VS systems, the **STEP** command is available only when you include the debugging features in your Business BASIC interpreter during system generation. For information about generating a Business BASIC system, see your Business BASIC user's guide.

continued **STEP**

Example

This program illustrates the use of the **STEP** command.

```
* LIST
00010 REM * STEP example
00030 LET X=0
00040 PRINT "X= ";X
00045 LET X=X+1
00050 IF X<2 THEN GOTO 00040

* STEP 0

* STEP

[10]
* STEP 3

[30]
[40]X = 0
[45]
* STEP 1

[50]
* STEP 3

[40]X = 1
[45]
[50]
End of program
```

STMA

Statement and Command

Performs system calls that examine or modify aspects of a job, a terminal, or the system.

AOS/VS	DG/RDOS	UNIX

Format

STMA *type,argument(,argument...)*

Arguments

type	A numeric expression or number designating the type of parameter that is being examined or modified. Each type is discussed below.
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each STMA requires one or more arguments. For an explanation of the required arguments for each system call, see the explanation for that type of STMA in Table 1–3.

What It Does

STMA statements and commands are system calls that examine and modify a wide range of parameters related to the system, a specific job, or a terminal. The "How To Use It" section gives you a general idea of how each STMA is used.

NOTE: Table 1–3 lists the syntax formats for each STMA. In this table, each string, string expression, or substring argument ends with a dollar sign ($); all other arguments are numeric. In addition, a string variable or numeric variable (*numeric–variable*) is a value that the STMA returns. A string expression or numeric expression (*numeric–expression*) is a value that you assign before executing the STMA; both the string expression and the numeric expression can be either an expression or a variable. Since the arguments are not all fully explained in the "How To Use It" section, check the statement's syntax in Table 1–3 if you have a question about whether an argument requires a numeric or string argument.

continued

STMA

Table 1-3 Summary of STMA Syntax Formats

STMA Parameter	Data Type of Its Arguments
STMA 1,	*item, numeric-variable*
STMA 2,	*item, numeric–expression*
STMA 3,	*item, numeric-variable*
STMA 4,	*item, numeric–expression*
STMA 5,	*examine-flag, numeric-variable*
STMA 6,	*turn-flag-on*
STMA 7,	*turn-flag-off*
STMA 8,	*item*
STMA 9,	*item, string-variable$*
STMA 10,	*item, string-expression$*
STMA 11,	*Julian-date, numeric-variable1, numeric-variable2, numeric-variable3*
STMA 12,	*numeric-variable4, month, day, year*
STMA 13,	*string-variable$, table-string$*
STMA 14,	*string-variable$, type*
STMA 15,	*string1$, string2$, numeric-variable*
STMA 16,	*numeric–expression*
STMA 17,	*port, error*
STMA 18,	*filename$, numeric-variable*
STMA 19,	*error-number*
STMA 20	[*,channel*]
STMA 21,	*string$,numeric-expression, numeric-variable1, numeric-variable2* [*,numeric-variable3* (*,numeric-variable4*) (*,numeric-variable5*)]
STMA 22,	*item,* [*length, width, start-row, start-column, border, title$*]

STMA

continued

How to Use It

This section includes a description of each **STMA**, its required arguments, and any special considerations for performing a particular system call.

NOTE: Before you specify a numeric variable in an **STMA** statement, you must assign a value to the variable. All numeric values used in **STMA** statements are 16-bit values.

STMA 1 and 2

STMA 1 and **STMA 2** are related; **STMA 2** sets the value of *item* and **STMA 1** receives the value of *item*. These items include terminal type, error code, passed variables, and security code. Before you execute an **STMA 1**, your variable must be initialized. After the **STMA** execution, your variable will hold the value you're requesting. The variables in **STMA 1** and **STMA 2** pass 16-bit values, or 1-word allocations.

STMA 1, *item,variable*
STMA 2, *item, value*

variable	A simple numeric variable that receives a value when you execute the call.
value	A numeric expression for the value you want to set the item to.
item	A number indicating one of the available items:

0 Terminal type.

1 Error code passed between programs that are using **SWAP** and/or **CHAIN** statements.

2 Available for you to pass or retrieve information between programs (1-word allocations).

3 Same as 2. You may need more than one place to pass or retrieve information.

4 Reserved.

For example:

STMA 1,0,X gives you the terminal type in X.
STMA 1,1,ER gives you the error code passed to your program from another program that used **STMA 2,1** and a **SWAP** or **CHAIN** to your program. Error code will be in ER.

STMA 1,2,VARIABLE or **STMA 1,3,VARIABLE** gives you the data in VARIABLE that is passed to your program from another program that used **STMA 2,2** or **STMA 2,3** and a **SWAP** or **CHAIN** to your program.

continued **STMA**

STMA 2 can be used to set your terminal type. Note that if you set a new terminal type, and you want to set the default terminal characteristics for the new type, you must do a **PRINT @ (-19)**.

The **STMA 2** example of the program called **HOME.BB** passes a SYS(7) error code and data using **STMA** statements to the program **REMOTE.BB**.

In **HOME.BB**, if an error occurred on input, the error code would be in **SYS(7)** and an error message would appear at the terminal. To pass this error to the other program, pass only the value of **SYS(7)**. The input value is passed using VARIABLE and retrieved in D. Each transfer is a two-byte transfer.

STMA 1 and 2 Examples

1. Use **STMA 2** to pass an error code and to pass data.

```
00010 REM THIS IS HOME.BB
00020 ON ERR THEN STMA 2,1,SYS(7)  :If error, pass the error code.
00030 INPUT USING "",VAR          :Incorrect input will cause error.
00040 STMA 2,2,VAR                :Pass data in VAR.
00050 SWAP "REMOTE.BB"            :Execute REMOTE.BB and return.
```

2. Use **STMA 1** to pass an error code and to pass data.

```
00010 REM THIS IS REMOTE.BB
00020 DIM ERR$(30)            :Initialize error string.
00030 LET D = 0               :Initialize variables.
00040 LET ER = 0
00050 STMA 1,1,ER             :Get error code, if any.
00060 IF ER <> 0 THEN GOSUB 00100:Routine to print error, if any.
00070 STMA 1,2,D              :Receive data passed from HOME.BB.
00080 PRINT D                 :Print data.
00090 STOP
00100 REM ROUTINE TO PRINT ERROR
00105 PRINT   "THERE IS AN ERROR HERE."
00110 LET ERR$=ERM$(ER)       :Use the ERM$ function to get error.
00120 PRINT ERR$              :Print error message.
00130 RETURN
```

STMA 3 and 4

STMA 3 and STMA 4 are related; STMA 4 sets the value of *item* and STMA 3 examines it. Among the items accessed by STMA 3 and STMA 4 are the detach key (RDOS only), line cancel key, unpend keys, and the interrupt keys.

STMA 3, *item, variable*
STMA 4, *item, value*

variable	A numeric variable that receives the value of *item* when item is retrieved.

STMA *continued*

value	A numeric expression for the value you want to set the item to (usually an ASCII value).
item	A number indicating one of the available options.

0 (RDOS only) Detach key. Striking this key while executing a job causes the job to detach from the terminal and leaves the terminal free for other uses. The job will continue to run in a detached state. The default key is Ctrl-D.

1 Line cancel key. Strike this key to cancel an erroneous line you just typed. The default key for terminal type 6 is Ctrl-X. The default key for terminal type 8 is Ctrl-U. You can reset this key only if you are using terminal type 6.

2 Character delete echo. This character is sent to the terminal whenever you delete a character. If you set it to a value greater than 127, then the sequence (value-128), space, (value-128) will be echoed on the terminal. This is how it appears to erase a character: the cursor backs up, prints a space, and backs up again. In this way, you would set your value to the ASCII decimal value of a backspace plus 128. You can reset this value only if you are using terminal type 6.

3 Character delete key. Default is DELETE. Useful only as a statement. You can reset this key only if you are using terminal type 6.

4 Primary unpend key. Carriage Return or New Line. Useful only as a statement. See explanation of the secondary unpend key.

5 Secondary unpend key. Useful only as a statement. When a program stops on an **INPUT** statement, it is waiting for an answer. An unpend key terminates your input. For example, when you enter a statement or command line, you usually end the line with a carriage return.

In many DG/RDOS systems, the carriage return (ASCII decimal value <13>) is the default primary unpend character. However, if you are using a D200, D400, D450, or G300 terminal, the New Line is the default primary unpend character on all but the master console, which will use carriage return. In AOS/VS and UNIX systems, the default primary unpend character is New Line (ASCII decimal value <10>). For more information about the UNIX unpend characters, see the on-line file **TERMINALS.DOC** in the **DOC** directory.

When a program stops running, the primary unpend character (no matter what it's currently set at) and the secondary unpend character are both returned to their default values. The secondary unpend character is typically set to a New Line if the primary unpend character is Carriage Return or Carriage Return if the primary unpend character is New Line.

This is automatic in case you forget which key to press when you want to unpend. When the program stops, you can set the unpend keys and unpending will occur if you strike one of those keys. The unpend key is converted to a carriage return (or New Line) before it is placed in the I/O buffer.

If an **INPUT** statement is terminated by a secondary unpend key, **SYS(10)** will equal 1 (see the **SYS** function).

If you set both unpend keys to the ASCII value <255> (nonenterable character), then unpending occurs on the next character you input and the character is put in the I/O buffer. This is useful for allowing only one-character inputs.

When you set both unpend keys to <255>, control characters and lowercase characters are recognized, and the line cancel and character delete keys are not processed. Unpend characters are never echoed on the terminal.

NOTE: Function keys on a Data General DASHER® terminal actually send a two-character sequence where the first character on DG/RDOS systems is <30> for all function keys and the second character is a lowercase character from q to z. You can use a function key as a secondary unpend key by setting the secondary unpend key to <30>. On AOS/VS and UNIX systems, use **SYS(50)** instead of <30>.

6 Primary interrupt key. This is usually the Escape key. You can change its function.

7 Secondary interrupt key. This also can be the function of any key you choose. Interrupts stop a program's execution and return to keyboard mode unless you handle the interrupts in your program with **ON IKEY** statements. It is also possible to test whether an interrupt occurred without enabling interrupts—see **STMA 5**, **6**, and **7** and **SYS(26)**. **SYS(10)** indicates what kind of interrupt occurred.

A zero value is a primary interrupt, and a one value is a secondary interrupt.

NOTE: If you are using AOS/VS, the key you set is the second in a sequence beginning with Ctrl–C. The second key must also be a Ctrl–key combination.

8 Page width or number of characters allowed on a line at your terminal. See the **PAGE** command for more information.

9 Tab size. This sets tabs of equal length on a line for spacing your output. A tab should not be longer than the current PAGE width setting. See the **TAB** statement for more information.

10 Maximum number of characters allowed on input. Useful only as a statement. This allows you to fix the number of characters a terminal operator can type as input. Any additional characters are neither echoed nor stored. The maximum number of characters is always reset to 255 whenever the program stops. If you set it to a negative number, the absolute value of the number will be the number of characters allowed on input and unpending will occur on the last character.

11 Used only in the **HELLO** program to set the push level to 0.

12 First-echoed character when a line cancel occurs. This is usually set to a backslash (\), which is an ASCII <92>.When you press the line-cancel key, you usually get two characters echoed: the first character is the backslash (\), and the second character is usually the default primary unpend key (see discussion above). This character is used to return the cursor to the beginning of a new line.

13 Second-echoed character when a line cancel occurs. This is usually the same as the primary unpend key. See explanation for first-echoed character (**STMA 4,12**).

14 Pad character. This character is sent after all line feeds to prevent you from typing anything before the cursor returns to the home position. This is useful on terminals with a slow line feed action.

15 Number of pad characters to send. Normally you don't need any pad characters for fast terminals; e.g., DASHER display terminals.

16 Returns a 0 value.

17 Returns a 0 value.

18 The default primary unpend key. The primary unpend key will be set to the specified value (for the issuing user), the next time you either return to the Business BASIC prompt or return from a program to which you swapped. This item is *not* destroyed by **PRINT @(−19)**, which resets terminal characteristics to system default values.

19 The default secondary unpend key.

Items 18 and 19 are copied into items 4 and 5 (described above) when **PRINT @(−19)** is executed or if the system returns to the Business BASIC prompt, unless both 18 and 19 contain the ASCII value <255>. If both 18 and 19 contain <255>, a **PRINT @(−19)** causes the defaults for the terminal type to be copied into items 4 and 5 respectively.

STMA 5, 6, and 7

STMA 5, 6, and **7** are related; the items involved in these system calls are flags. They are set to one by **STMA 6**, set to zero by **STMA 7** and may be examined by using **STMA 5**. The status flags accessed by these calls include character code, lowercase or uppercase, and column counter.

STMA 5, *examine-flag, numeric-variable*
STMA 6, *turn-flag-on*
STMA 7, *turn-flag-off*

numeric-variable	A numeric variable that receives the value of *flag* and must be initialized before use.
flag	is one of the following available status flags:

0 No echo. Also called half-duplex. Input characters are not sent back (echoed) to the terminal.

1 Allow lowercase characters to be input along with uppercase characters. Remember some data must be typed in uppercase when lowercase is enabled. For example, Business BASIC programs that test strings and only test for an uppercase value will fail if the string contains lowercase. In addition, reserved

continued **STMA**

filenames must be uppercase. (**STMA 14** can be used to convert lowercase to uppercase.)

2 Allow control characters to be input. Normally, control characters (except line and character deletes) are ignored on input. In AOS/VS (terminal type 6), you cannot input certain characters unless you precede them with a Ctrl–P. These characters are ^C, ^O, ^P, ^Q, ^S, ^U, ^T.

3 Disable the column counter. Normally, when enough characters have been output to equal the page width, Business BASIC inserts a carriage return to put the cursor at the beginning of the next line. Setting this flag inhibits that action.

4 (DG/RDOS only) No messages allowed. **STMD 1** and certain **OPCLI** commands may override this flag; **STMD 0** and **MSG** will not.

5 No IKEY. This flag does not allow an interrupt to stop your program or to catch it in an **ON IKEY** trap. It only allows the IKEY indicator, **SYS(26)**, to be set to 1 to indicate that an interrupt occurred. If you cleared this flag and **SYS(26)** equaled 1, then an interrupt would occur immediately. You can reset **SYS(26)**, without enabling interrupts, by using **STMA 8,3**.

6 Suppress listing of labels on **GOTO** and **GOSUB**. If the destination of a **GOTO** or **GOSUB** is a **REM** comment, then the comment appears after the **GOTO** or **GOSUB** when the program is listed. This always occurs when you use **GOSUB** to branch to one of the Business BASIC subroutines because each subroutine begins with a **REM** statement. To prevent it from occurring, set this flag.

7 When set, this flag indicates that fill characters in **PRINT USING** statements are to overwrite all characters not used in D*w.d* and E*w.d* formats. When cleared (default), fill characters will overwrite only the leading zeros in D*w.d* and E*w.d* formats.

8 No echo. Also called half-duplex. Input characters are not sent back (echoed) to the terminal.

9 Allow listing of eight-bit characters. Normally, characters with ASCII values greater than 127 decimal will be listed as the ASCII value enclosed in angle brackets. If this flag is set, they will be output literally. Eight-bit characters that can output literally to seven-bit terminals will often erroneously appear as the seven-bit character made by stripping the eighth bit. For example, this occurs when a listing file is created and then displayed to a seven-bit device using **TYPE** or **PRINT**.

10 Allow users who select operating system multiplexor support to disable or enable operating system interrupts for the multiplexor line. If an interrupt character must be passed to Business BASIC on a read, this statement should be used. When set, this flag indicates that Ctrl–C Ctrl–*x* interrupts for the line are disabled. When cleared (this is the default), Ctrl–C Ctrl–*x* interrupts for the line are enabled. This item is available only in a DG/RDOS Business

STMA

continued

BASIC system that has been generated with operating system multiplexor support.

11 (DG/RDOS only) Allow users who select operating system multiplexor support to disable the operating system Ctrl-S (suspend processing) and Ctrl-Q (resume processing) functions for the multiplexor line.

12 (UNIX only) Perform pathname conversion. When set, pathnames in either UNIX or AOS/VS format are accepted. The AOS/VS pathnames' colons (:) are converted to slashes (/), and all characters are converted to upper case. When cleared, AOS/VS-style pathnames are rejected.

13 (AOS/VS and UNIX only) Allow 255 characters on **INPUT** and **PRINT**. When this value is not set, only 132 characters are allowed.

Example

Check the message flag; if it is set, clear it. Check the IKEY flag; if it is clear, set it.

```
00010 LET VARIABLE = 0                  :Initialize variable.
00020 STMA 5,4,VARIABLE                 :Is no message flag set?
00030 IF VARIABLE = 1 THEN STMA 7,4     :If set, clear it.
00040 STMA 5,5, VARIABLE                :Is no-ikey flag set?
00050 IF VARIABLE = 0 THEN STMA 6,5     :If not, set it.
. . .
00100 IF SYS(26) = 1 THEN STMA 7,5    :If an interrupt occurred, then
                             :re-enable interrupts to process IKEY.
. . .
```

STMA 8

STMA 8 is used to reset the **FOR...NEXT** stack, **GOSUB...RETURN** stack, IKEY indicator and clear the input buffer. If you nest **FOR...NEXT** loops or **GOSUB...RETURN** routines beyond their legal nesting limits, you get `Error 18 - GOSUB nesting` or `Error 20 - FOR nesting`.

STMA 8, *item*

item is one of the following values:

0 Resets the **FOR...NEXT** and **GOSUB...RETURN** stacks. On AOS/VS and UNIX systems, this also resets the **DO...WHILE** stack.

1 Resets the **FOR...NEXT** stack.

2 Resets the **GOSUB...RETURN** stack.

3 This resets the **SYS(26)** IKEY indicator. Use it with **STMA 6,5** and the **SYS** function described in this chapter. Normally, you use **STMA 8,3** to test whether an interrupt occurred without enabling interrupts.

4 This clears, or "flushes," the input buffer. It is useful in heavy key-entry environments where operators may key ahead rapidly. If an operator makes an error, the input buffer should be flushed before having the operator rekey the information; otherwise there is "garbage" in your input buffer. This causes everything up until the next **INPUT** statement to be flushed.

5 This clears, or "flushes," the output buffer during heavy output. (This applies only to AOS/VS and UNIX; it is ignored by DG/RDOS.) For example, if you executed the following lines of code:

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

Processing these lines would take less than one second; but since Business BASIC tries to store, under certain conditions, up to 512 bytes of information in its internal output buffer, output can be sporadic and might not be synchronized with the execution of the statements. Insert an **STMA 8,5** at line 25 of the program to overcome this problem.

6 (AOS/VS and UNIX) Resets the **DO...WHILE** stack.

STMA 9

This statement examines the following information: account or username, name of the current directory, name of the current program, current default output device name, name of the system library directory, name of the system directory, and complete account name.

STMA 9, *item, string-variable$*

string-variable$	A string variable (already dimensioned) that receives the string value of *item*.
item	One of the following options:

0 Account, login name, or username; i.e., the name you used to log on. This name is truncated to five bytes and the terminal type resides in the sixth byte. If the account name is less than five bytes, it fills the remaining bytes with spaces.

1 Name of the current directory.

2 Name of the current program, set by a **SAVE** command.

3 Current default output device name.

4 Name of the system library directory.

5 Name of system directory.

6 (UNIX only) Complete account name up to 32 characters. This does not have the terminal type appended to it.

STMA *continued*

STMA 10

This sets the special characters allowed in crammed strings (i.e., **CRM$/UCM$** conversions) or sets the default output file.

STMA 10, *item, string-expression$*

string-expression$	A string literal in quotation marks or a string variable or string array element (UNIX only) already dimensioned and assigned a string value.
item	One of the following options:

0 The **CRM$** function crams three bytes of a string into two bytes of the resulting string. You may cram and uncram a string accurately if the string has any of these characters: uppercase A–Z, 0–9, and four special characters: space, comma (,), dash (–), and period or decimal point (.). These are the default special characters, but you can specify your own **CRM$/UCM$** characters with this **STMA**. Your string, as either a string literal in quotation marks or a string variable, must contain the special characters you want to specify. See the **CRM$** and **UCM$** functions.

NOTE: If you use **STMA 10,0** to change the special characters, the first special character in your list (for example the $ in a list of "$+–,") is the one that will be used in place of an invalid character. If you use **STMA 10,0** to restore the default special characters, you should use a space as the first character in the list, so that a space replaces an invalid character.

1 The value of *string-expression* becomes the default output file as returned by **STMA 9,3,***string-variable$*. This is the same function that is performed by the Business BASIC CLI **SQUE** command. **STMA 10,1,** *string-expression* resets standard output.

Examples

Both these examples define the special characters as a plus (+), minus (–), dollar sign ($) and equal sign (=). Now these characters can be used instead of the default characters in **CRM$/UCM$** conversions.

```
00010 DIM A$(4)
00020 LET A$="+-$="
00030 STMA 10,0,A$
```

or

```
00010 STMA 10,0,"+-$="
```

STMA 11 and 12

STMA 11 converts a Julian date to a month/day/year format, and **STMA 12** reverses this process.

STMA 11, *Julian-date, variable1, variable2, variable3*
STMA 12, *variable4, month, day, year*

variable1	Receives the value for month.
variable2	Receives the value for day.
variable3	Receives the value for year.
Julian-date	A numeric expression or variable for the number of days since 1/0/68 (January 0, 1968).
variable4	Receives the value for the Julian date (days since January 0, 1968).
month, day, year	Numeric expressions or variables for the date you specify.

Initialize all variables before using them.

A Julian date is the number of days from a base year; we use Data General's birth year as the base year: January 0, 1968. Any input to **STMA 12** that attempts to set a date prior to January 0, 1968, returns invalid results. **STMA 12** converts a date in the form month/day/year to a Julian date. Valid months are 1 through 13, where 13 returns a date 12 months from the date entered. If you enter a number outside this range, the results are unpredictable. You can refer to the previous month by supplying a value of 0 or a negative value for the *day* argument. A value of 0 for *day* returns the last day of the previous month, a value of -1 returns the next to last day of the previous month, and so on. You can determine a date 60 days from a date by adding 60 to the *day* argument. Valid years begin at 68.

By using the **CHR$** function, a Julian date can be stored in two bytes of a string variable thus saving four bytes in a record being written to a disk file. The Julian date is returned in *variable*. You can get the day of the week by using the **MOD** function with *variable*; e.g., **MOD** (*variable*,7) would return a 0 for Sunday, 1 for Monday, and so forth.

Two-digit year designations are only appropriate for the twentieth century. For the year 2000 and beyond, the year variable is 100 and beyond. For the year 2100 and beyond, the year variable is 200 and beyond.

STMA

continued

Examples

1. This program will print the result 8/9/83, the date translated from the Julian date 5700.

```
00010 LET MONTH=0
00020 LET DAY=0
00030 LET YEAR=0
00040 LET JULIAN=5700
00050 STMA 11,JULIAN,MONTH,DAY,YEAR
00060 PRINT MONTH;"/";DAY;"/";YEAR
```

2. Determine the day of the week of a date by converting the date to Julian. MOD is then used to determine the remainder after dividing by seven.

```
00010 DIM D$[10],M$[10]
00020 LET JULIAN=0
00030 DATA "SUNDAY","MONDAY","TUESDAY","WEDNESDAY","THURSDAY",
"FRIDAY" ,"SATURDAY"
00040 INPUT "MONTH: ",MONTH
00050 INPUT "DAY: ",DAY
00060 INPUT "YEAR: ",YEAR
00070 RESTORE 00030
00080 STMA 12,JULIAN,MONTH,DAY,YEAR
00090 LET D=MOD(JULIAN,7)
00100 FOR I=0 TO D
00110   READ D$
00120 NEXT I
00130 PRINT "THIS IS A ";D$
00140 END
```

Sample use of this program is shown below. Note that the year 2005, below, is input as 105.

```
* RUN
MONTH: 6
DAY: 11
YEAR: 84
THIS IS A MONDAY

* RUN
MONTH: 4
DAY: 12
YEAR: 105
THIS IS A TUESDAY

* RUN
MONTH: 10
DAY: 5
YEAR: 83
THIS IS A WEDNESDAY

*
```

3. By using 0 as the day with valid month and year numbers, **STMA 12** and **STMA 11** return the last day of the previous month.

```
00010 LET MO=2 \ DAY=0 \ YR=88 \ JULIAN=0
00020 STMA 12,JULIAN,MO,DAY,YR
00030 STMA 11,JULIAN,MO,DAY,YR
00040 PRINT MO;"/";DAY;"/";YR                    :displays 1/31/88
```

To find the last day of the year, use 13 as the month number. In addition, you can determine a date 60 days from another date by adding 60 to the DAY variable.

STMA 13 and 14

STMA 13 and **14** are related; **STMA 13** is used to translate a string of characters using a table which is set up by the user. **STMA 14** is a special case of **STMA 13**; it translates ASCII to EBCDIC, EBCDIC to ASCII, lowercase to uppercase and uppercase to lowercase.

STMA 13, *string-variable$, table-string$*
STMA 14, *string-variable$, type*

string-variable$	A string variable that contains the string to be translated and receives the translation. Any character in the string variable whose code is greater than the length of the *table-string* translates to the last character in the *table-string*.
table-string$	Another string variable or string literal in quotation marks containing the table with a maximum of 256 characters.
type	A number indicating one of the following available translation types:

0 Upper and lowercase alphabetic ASCII characters to uppercase alphabetic ASCII characters.

1 ASCII code to EBCDIC code.

2 EBCDIC to ASCII.

3 Uppercase ASCII characters to lowercase.

4 Characters with eighth bit to space.

5 Characters without eighth bit to space.

6 Eighth bit is cleared.

7 Eighth bit is set.

8 Characters without eighth bit to space and eighth bit is cleared on those where set.

STMA

continued

You must dimension string variables before using them.

In a computer, a character exists as a coded number. Each character (including uppercase and lowercase letters, control characters, symbols, and even unprintable characters) corresponds to one of 256 codes. Different computer systems may use different code systems. **STMA 13** and **14** translate a string variable from one code system to another.

Use **STMA 14** to translate between ASCII and EBCDIC, and between uppercase and lowercase. Use **STMA 13** to translate from a table you create.

When you use **STMA 13** to translate an ASCII character string, each ASCII character's numeric code represents an offset in the translation table (string) you defined. The base (first) character is at offset 0. If the code exceeds 255, the last character in the translation table is used.

For example, if the first character in a string variable has a code number 5, **STMA 13** counts five positions (from 0) into the table string and arrives at the sixth character position. The character in this position becomes the first character in the translated string. Therefore, you count "code+1" positions into the table string to find a character.

Examples

1. In this example, a character with a code of 2 translates to a 3. A character with a code of 5 translates to a 6, and a character with a code of 8 translates to a 9.

```
00010 DIM TRANS$(50),TABLE$(256)
00020 LET TABLE$="1234567890"
00030 LET TRANS$="<2><4><8>"
00040 STMA 13,TRANS$,TABLE$
00050 PRINT TRANS$

* RUN
369
```

2. In this example, the program allows any ASCII character input to translate to the same ASCII character except the ampersand (&) sign (ASCII 38), which translates to a comma (ASCII 44). Line 30 fills the table with ASCII values. Line 50 changes the ampersand to a comma in the table. Line 60 allows for the input of data, which is translated on line 70.

```
00010 DIM TRANS$(256),TABLE$(256)
00020 FOR J=0 TO 255
00030   LET TABLE$(J+1)=CHR$(J)
00040 NEXT J
00050 LET TABLE$(39,39)=CHR$(44)
00060 INPUT USING "",TRANS$
00070 STMA 13,TRANS$,TABLE$
00080 PRINT TRANS$
```

3. This program reads data from tape in EBCDIC format and converts EBCDIC to ASCII.

```
00010 OPEN FILE(0,7), "MTO:0"
00020 DIM TRANS$(80)
00030 MTDIO 0,1,TRANS$,E%
00040 STMA 14,TRANS$,2
00050 OPEN FILE(1,2), "OUTPUT"
00060 PRINT FILE(1),TRANS$
```

STMA 15

This compares two strings and retrieves a match or no match variable.

STMA 15, *string1$, string2$, variable*

string1$* and *string2$	String variables, string literals, string expressions, or string array elements (UNIX only).
variable	A simple numeric variable that receives either a 1 (match) or 0 (no-match) value.

The characters in *string1* are compared with those in *string2* returning a value of 1 in variable if they match, or 0 if they don't match. The strings must be equal in length and must match character-by-character for a match to occur, unless you use dash (–), asterisk (*) and plus (+) conventions, (only allowed in *string1*).

Here is a summary of the dash, asterisk, and plus conventions:

A dash (–) matches any number of characters, including none, but does not match a decimal point (used also for filename extensions). Here are examples:

string1	string2	
"-AB-"	"CLASSAB"	Match!
"-AB-"	"FABRST"	Match!
"-AB-"	"CABR.ST"	No match. (Forgot extension.)
"-AB-.-"	"CABRST.UVWXYZ "	Match!

Plus (+) matches any number of characters including a decimal point. Here are examples:

string1	string2	
"+AB+"	"CLASSAB"	Match!
"+AB+"	"CLSS.AB"	Match!
"-AB+"	"CL.AB"	No match.
"+AB-"	"CL.ABSS"	Match!

STMA

continued

Asterisk (*) matches any character which is in the same place as the asterisk. Only one character matches per asterisk. Here are examples:

string1	string2	
"*AB**"	"CABRS"	Match!
"*L*B***"	"AL BOVE"	Match!
"*CD**"	"RSTCDEF"	No match.
"*CD-"	"RCDEFGH I J"	Match!
"-CD-.*"	"UVWCDRST.A"	Match!
"-CD-.*"	"UVWCDRS.TA"	No match.
"*****"	"1234"	No match. (string1 greater)

Example

```
00005  LET NUM=0
00010  DIM A$(20),B$(20)
00020  OPEN FILE(0,0), "MYFILE"
00030  READ FILE(0),A$
00040  OPEN FILE(1,0), "YOURFILE"
00050  READ FILE(1),B$
00060  STMA 15,A$,B$,NUM
00070  IF NUM THEN PRINT "MATCH!" ELSE PRINT "NO MATCH."
```

STMA 16 and 17

(DG/RDOS only)

STMA 16 and **17** are related; **STMA 16** detaches a job while **STMA 17** attaches a job or process to a specified port.

STMA 16, *value*
STMA 17, *port, error*

value A number, numeric expression or variable that has a value of 0 or 1 as follows:

0 When the current job detaches, the terminal remains inactive until you log on again; i.e., the **HELLO** program is not called.

1 When the current job detaches, it calls the **HELLO** program, which automatically starts the logon procedure at your terminal.

port A number, numeric expression, or variable indicating a terminal number (use **STAT** or **PORTS** to find the ones in use).

error A numeric variable, already initialized, that receives the error code if an error occurs.

Use **STMA 16** in your program to detach your terminal from your job. When you make *value* a 0, you leave the terminal inactive until someone else logs on by pressing the escape key. If you want the **HELLO** program to start the logon procedure automatically, make *value* a 1.

STMA 17 attaches the current job to the terminal indicated by *port*, leaving the current terminal inactive. The current terminal may be logged on again. If the port already has a job attached to it or if the port is invalid, STMA 17 returns an error code in *error*. If operating system multiplexor support has been generated, `Error 24 – No available channels` is returned if no channel is available to open the multiplexor line for the target terminal. This is useful for starting a print job on a receive-only terminal that you cannot log on to otherwise.

Use **STMA 16** and **STMA 17** in your program to detach the program and attach to another port. A practical application is to have a large print program detach itself from your terminal and attach itself to a receive-only port that has no keyboard and cannot be logged on.

Example

In this example, line 20 detaches the job and calls **HELLO** to log on automatically. Line 30 attaches the job to port 5. If there is an error, call the **STAT** program.

```
00010  LET ERROR=0
00020  STMA 16,1
00030  STMA 17,5,ERROR
00040  IF ERROR THEN SWAP "STAT"
  . . .
```

STMA 18

(DG/RDOS only)

This examines, assigns, or frees a reserved file or device.

STMA 18, *filename$, variable*

filename$	The name of a reserved file or device (reserved files are set up during BASIC system generation).
variable	A numeric variable that returns a specific value when you assign a specific value to it. The available values are:

0 To examine a device or reserved file's condition. It returns a 0 in *variable* if *filename* is free; otherwise, it returns a number greater than 512 that is the base address in the user status table of the job that has the file reserved.

1 To free *filename* from the job executing the **STMA** so that other jobs may use the file or device. If the call is successful, it returns a 1 in *variable*; otherwise it returns a value greater than 512—the base address in the user status table of the job that has the file reserved.

2 To assign *filename* to the job executing the **STMA** so that no one else may access that device or file. If the call is successful, it returns a 2 in *variable*; otherwise it returns a value greater than 512—the base address in the user status table of the job that has the file reserved.

STMA *continued*

If *filename* is not a valid reserved file or device, **STMA 18** returns a value of -1 in *variable*.

Use **STMA 18** to control access to reserved files and devices. To determine the reserved filenames in your system, examine the *systemname*.SG audit file created during BASIC system generation.

Example

```
00010 LET VARIABLE = 0                 :Initialize variable.
00020 STMA 18,"$LPT", VARIABLE         :Examine the printer.
00030 IF VARIABLE > 512 THEN GOTO 00010      :If not free, try again.
00040 PRINT "DEVICE READY"
00050 LET VARIABLE = 2
00060 STMA 18,"$LPT",VARIABLE          :Assign printer to my job.
00070 IF VARIABLE = 2 THEN GOTO 00090 :If successful, continue.
. . .                                  :Error code goes here.

00090 REM -- PRINT ROUTINE
. . .
00300 REM -- END OF PRINT ROUTINE
00310 LET VARIABLE = 1
00320 STMA 18,"$LPT",VARIABLE                :Free the line printer.
00330 IF VARIABLE = 1 THEN GOTO 00400 :If successful, GOTO
                                       :end of program.
00340 PRINT "RESERVED FILE NOT FREE; WILL TRY AGAIN"
00350 GOTO 00310
. . .
```

STMA 19

STMA 19 generates system errors.

STMA 19, *error-number*

error-number	A positive *error-number* indicates a Business BASIC error. The value is placed in **SYS(31)** and **SYS (7)**. If the error number is a negative DG/RDOS error, the value is placed in **SYS(7)** only. To generate an AOS/VS error, subtract 65536 from the negative AOS/VS error number and use the result as the *error-number* argument. This action causes **SYS(7)** and **SYS(31)** to be set.

STMA **19** sets **SYS(20)**.

STMA **19** results in an error interrupt or the execution of an **ON ERR** trap. This provides a convenient way of generating errors from within a Business BASIC program that can be handled by a generalized error processing routine.

On AOS/VS and UNIX systems, the **RAISE** statement can be used for the same purposes as **STMA 19**.

This program uses **STME 4** to return the complete pathname of a file. The error results because the file does not exist.

```
00010 ON ERR THEN GOTO 00090
00020 DIM FILENM$[32],PATH$[132],ER7$[132],ER31$[132]
00030 LET FILENM$="TEST" \ PATH$=FILL$(0) \ ERROR=0
00040 STME 4,ERROR,FILENM$,PATH$
00050 PRINT "The error returned from STME 4 is: ";ERROR
00060 IF ERROR <>-1 THEN STMA 19,-ERROR-65536
00070 PRINT "The pathname of the file is: ";PATH$
00080 END
00090 REM ** Error Routine
00100 LET ER7$=ERM$(SYS(7)) \  ER31$=AERM$(SYS(31))
00110 PRINT SYS(7);ER7$,SYS(31);ER31$
00120 END
```

```
* RUN
The error returned from STME 4 is:  21
-10 File does not exist        -21 File does not exist
```

STMA 20

STMA 20 passes a user library file to BASIC by opening the file and passing its channel number.

STMA 20 [,*channel*]

channel	An optional numeric expression for the channel number of a file opened for random or sequential access and input.

STMA 20 passes to Business BASIC the channel number of a user program library file that was opened for reading. The specified BASIC channel then becomes a BASIC reserved channel and appears to be closed, so that it can be used and opened again for another file. If the user already had a library channel assigned to his job, it is closed before making the specified channel the new library file channel for the job. Calling **STMA 20** without the channel argument causes the user's current library file channel to be closed.

Example

```
00010 OPEN FILE[1,4],"MYLIB.PL"        :Open the user library
00020 STMA 20,1               :Indicate that it is a user library
00030 SWAP "%ADVENTURE"
00040 END
```

STMA *continued*

STMA 21

STMA 21 is a helper for the **SFORM.SL** subroutine, which is a part of the CSM utility. It is used in displaying a CSM screen file.

STMA 21, *string$,numeric-expression, numeric-variable1, numeric-variable2*
[*,numeric-variable3* (*,numeric-variable4*) (*,numeric-variable5*)]

STMA 21 corresponds to the variables used in **SFORM.SL** as follows:

string	A string that is expected to be in the format of the variable SCRN$.
numeric-expression	A numeric expression that is treated as the variable F where F<0 is interpreted as relative field number (1–*n*). When F>100, it is interpreted as a number representing row and field in the format RRFF. When F<100 and F>0, it is interpreted as the field number only.
numeric-variable1	A numeric variable that corresponds to the variable XROW. *Numeric-variable1* is optionally considered to be the input variable R when F>0 and F<100, otherwise *numeric-variable1* is ignored as input.
numeric-variable2	A numeric variable that is returned as the position of field F in a row corresponding to XPOS. *Numeric-variable2* is returned as –1 when the specified field does not exist.
numeric-variable3	A numeric variable that corresponds to the variable XCOL.
numeric-variable4	A numeric variable that corresponds to the variable XWID.
numeric-variable5	A numeric variable that corresponds to the variable XFLGS.

STMB

Statement and Command

Performs system calls that examine or modify portions of memory.

AOS/VS	DG/RDOS	UNIX

Format

STMB *type,argument*(*,argument...*)

Arguments

type	A numeric expression or number designating the system call to be performed. Each type is discussed below.
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each **STMB** requires one or more arguments. For an explanation of the required arguments for each system call, see the explanation for that type of **STMB** in Table 1–4.

What It Does

The **STMB** calls allow you to access and modify portions of memory. When you alter memory contents, you must be extremely careful to ensure that you are actually modifying the locations you want.

NOTE: Table 1–4 lists the syntax formats for each **STMB**. In this table, each string, string expression, or substring argument ends with a dollar sign ($); all other arguments are numeric. In addition, a string variable or numeric variable (*numeric–variable*) is a value that the **STMB** returns. A string expression or numeric expression (*numeric–expression*) is a value that you assign before executing the **STMB**; both the string expression and the numeric expression can be either an expression or a variable. Since the arguments are not all fully explained in the "How to Use It" section, check the statement's syntax in Table 1–4 if you have a question about whether an argument requires a numeric or string argument.

How to Use It

This section includes a description of each **STMB**, its required arguments, and any special considerations for performing a particular system call.

STMB

continued

Table 1-4 Summary of STMB Syntax Formats

STMB Parameter	Data Type of Its Arguments	
STMB 0,	*item, address*	
STMB 1,	*wordcount, address, destination*	
STMB 2,	*wordcount, address, source*	
STMB 3,	*byteaddress, destination$*	*(DG/RDOS)*
STMB 3,	*address, destination$*	*(AOS/VS and UNIX)*
STMB 4,	*byteaddress, source$*	*(DG/RDOS)*
STMB 4,	*address, source$*	*(AOS/VS and UNIX)*
STMB 5,	*wordsize, address, destination*	
STMB 6,	*job-number, error*	*(DG/RDOS)*
STMB 6,	*wordsize, address, source*	*(AOS/VS and UNIX)*
STMB 7,	*job-number, error*	
STMB 8,	*job-number, error*	
STMB 10,	*function, address, mask*	
STMB 11,	*string$, numeric–variable1, numeric–variable2, numeric–variable3*	
STMB 12,	*job-number*	
STMB 13,	*job-number, error, flag, string$*	
STMB 14,	*job-number, error, flag*	
STMB 15,	*port-number, characteristics*	
STMB 16,	*flag*	
STMB 18,	*address, numeric–expression*	
STMB 19,	*port-number, modem-status*	
STMB 22,	*byte-address, string-variable$*	
STMB 23,	*address, numeric–variable*	
STMB 24,	*error, accumulator$* [*,flag*]	
STMB 25,	*user-channel, system-channel*	

STMB 0, *item, address*

Retrieves a word address. Items that are undefined return a value of 65535.

DG/RDOS systems, the word address is a single word; on AOS/VS and UNIX systems, it is a double word.

address	A variable that receives the word address of *item*.
item	A number indicating one of the available items:

0 Returns the memory address of a 21-element array. Each element of this array is the address returned by the corresponding **STMB 0,** *item, address* call. On UNIX systems, this call is not supported and always returns 65535.

1 (DG/RDOS only) Returns the memory address of the line table address. The line table is an array where the first line is port 0. Each element in the table represents the word address of the start of the User Status Table for a given line. Note that detached jobs are not reflected in this array.

2 (DG/RDOS only) Returns the memory address of the job table address. The job table is an array. Each element in the table represents the word address of the start of the User Status Table for a given job number. Note that the number of entries corresponds with the number of jobs specified when Business BASIC was executed; i.e., an address is reserved in the User Status Table even if the job is not running.

3 (DG/RDOS only) Returns the memory address of a word indicating the highest multiplexor line number available to Business BASIC. This number is two more than the number of multiplexor lines generated into Business BASIC.

4 Reserved.

5 Reserved.

6 For DG/RDOS:

Returns the memory address of the channel table. The channel table is an array of *n* elements, where *n* is the number of channels specified in your BSG. Each value of *n* corresponds to an RDOS channel number. The value contained in an array element is either:

- The User Status Table address of the user who has a file opened on that channel.
- The Business BASIC system address, if Business BASIC has opened a file on that channel.

STMB

For AOS/VS:

Returns the memory address of a word containing the address of the user channel table. The channel table contains four words for each of the 150 user channels that can be opened. The first four words apply to user channel 0; the next four words apply to user channel 1; etc. For each channel number, the meanings of the words are:

- System channel number
- Mode of open
- current file position—high
- current file position—low

This information corresponds to the first two columns of information returned by the utility program **SCHANS**.

For UNIX:

On UNIX systems, this feature is not supported and always returns 65535.

7 (DG/RDOS only) Returns the memory address of the reserved file table. This table contains the reserved filenames generated by the options chosen in the **BSG** program (for example, **$LPT**) and any additional reserved filenames specified in the BSG questions.

8 Returns the memory address of a word containing the maximum amount of memory available for the current job. On UNIX systems, this call is not supported and always returns 65535.

9 Returns the memory address of a two-element array that represents the global switches on Business BASIC. On UNIX systems, this call is not supported and always returns 65535.

10 (DG/RDOS only) Returns the memory address of a word containing the number of jobs specified when Business BASIC was executed.

11 Reserved.

12 (DG/RDOS only) Returns the memory address of the lock buffer area, which is an array of 11 by *n* elements, where *n* is the number of locks specified in the BSG. Words in each lock buffer have the following meanings:

1	Reserved
2	Reserved
3	User job number
4	Lock number assigned by user
5	Filename (10 characters)
6	Filename (10 characters)

7 Filename (10 characters)

8 Filename (10 characters)

9 Filename (10 characters)

10 Starting sector number

11 Ending sector number

13 Reserved.

14 (DG/RDOS only) Returns the memory address of a word that indicates the length in blocks of one segment of the push file (**BASIC.PS**). This number is determined by the following formula:

segment length = (maximum–program–size + 1) * (number–of–jobs)

where the maximum program size is determined upon execution with the /M switch and is allocated in 512-byte blocks, and the number of jobs is specified at execution with the /J switch.

15 Returns the memory address of a double-word that is used by **SYS(29)**. On UNIX systems, this feature is not supported and always returns 65535.

16 Reserved.

17 Reserved.

18 Reserved.

19 Returns the memory address of a word containing the address of the current user's User Status Table. On UNIX systems, this feature is not supported and always returns 65535.

20 Reserved.

21 Returns the address of the physical channel number for the push file (**BASIC.PS**).

STMB 1, *wordcount, address, destination*

Copies the contents of memory starting at *address* (word address) for the number of words specified in *wordcount* to the variable or array *destination.*

STMB 2, *wordcount, address, source*

Copies from *source* into memory the number of words specified in *wordcount.* The words are stored in memory starting at *address* (word address).

STMB 3, *byteaddress, destination$* *(DG/RDOS only)*

Copies bytes from memory into the string variable or substring *destination$* until *destination$* is full. The copy starts at the *byteaddress* (word address * 2) position in memory.

STMB

continued

STMB 3, *address, destination$* (*AOS/VS and UNIX*)

Copies bytes from memory beginning at *address* into the string or substring variable *destination$* until *destination$* is full. In AOS, a byte address (word address * 2) is supplied for the *address* argument. In AOS/VS, a word address is supplied and converted to a byte address by Business BASIC.

STMB 4, *byteaddress, source$* (*DG/RDOS only*)

Copies the entire contents of the string variable or substring *source$* into memory starting at *byteaddress* (word address * 2).

STMB 4, *address, source$* (*AOS/VS and UNIX*)

Copies the entire contents of a string or substring variable *source$* into memory starting at the address specified. In AOS, a byte address (word address * 2) is supplied for the *address* argument. In AOS/VS, a word address is supplied and converted to a byte address by Business BASIC.

STMB 5, *wordsize, address, destination* (*AOS/VS and UNIX*)

Copies the contents of memory into the *destination* variable, starting at *address* (word address) for the number of words specified in *wordsize*. The *wordsize* argument must be 1 (for a narrow word) or 2 (for a double word). If any other value is used, `Error 37 – User routine` occurs. Use **STMB 5** when you want to store a double word value in the destination variable.

STMB 6, *job-number, error* (*DG/RDOS only*)

Attaches the specified detached job to the current terminal, returning any error in the variable *error*. Note also that the job which was attached to the current terminal (and which issued the **STMB 6**) becomes detached. Errors you can receive are:

51	Job not logged on
70	Invalid job number
71	Job already attached

STMB 6, *wordsize, address, source* (*AOS/VS and UNIX*)

Copies from the variable *source* into memory the number of words specified in *wordsize*, starting at the word address specified. The *wordsize* argument must be 1 (for a narrow word) or 2 (for a double word). If any other value is used, `Error 37 – User routine` results. Use **STMB 6** to retrieve a double word value from a source variable.

STMB 7, *job-number, error* (DG/RDOS only)

Forces the specified job to execute a **BYE** command, returning any error in the variable *error*. Errors you can receive are:

51 Job not logged on

70 Invalid job number

78 MSG in progress

STMB 8, *job-number, error* (DG/RDOS only)

Resets the **ON IKEY** condition for the specified job, clears the "ignore IKEY" flag, and then simulates an interrupt, stopping the program. Errors you can receive are:

51 Job not logged on

70 Invalid job number

STMB 9

Reserved.

STMB 10, *function, address, mask*

If *function* = 1, **STMB 10** sets the bits in the word at the specified word address according to the bits set in the variable *mask*. If *function* = 0, **STMB 10** clears the bits in the word at the specified address according to the bits set in the variable *mask*.

STMB 11, *string$, variable1, variable2, variable3*

Scans the contents of *string$*, puts the number of 1 bits in *variable1*, and puts the largest consecutive number of 0 bits in *variable2*. *Variable3* is required and is set to 0. *string* may be a substring, but it must be word-aligned and of even byte length—that is, the subscripts of *string$*[*x,y*] must be such that *x* is odd and *y* is even.

STMB 12, *job-number* (DG/RDOS only)

Scans the job table for the first available job and executes **HELLO** for that job. Business BASIC returns the number of the job in the variable *job-number*. If no available job exists, it returns -1 in *job-number*. Note that when this call finds an available job, it starts the job as a detached job. The **HELLO** program waits 15 seconds for input and if none is received, it does a **BYE** for that job.

STMB 13, *job-number, error, flag, string$* (DG/RDOS only)

Places the contents of *string$* into the input buffer for the specified job. Business BASIC sets the alternate IKEY or unpend flag value for *job-number* to the value of *flag* (0 or 1), and unpends the job. Errors you can receive are:

STMB *continued*

51 Job not logged on

70 Invalid job number

72 Job is not waiting on input

If you specify any job that is running (other than your own job), you get error = 0 (successful), unless input for that job is currently being processed, in which case you get `Error 72 - Job is not waiting on input.` If the job to which you send a string is waiting on an **INPUT** statement, but you send it an inappropriate string, it displays a prompt for more input (\?). If the job to which you send a string is at the asterisk prompt, but you send it an inappropriate command, you receive the error message `Error 2 - Statement of command syntax is invalid.`

If operating system multiplexor support is chosen, using **STMB 13** in a tight loop to send commands to a job can cause a system hang, due to timing problems associated with the **.OPEN** and **.CLOSE** system calls. To avoid this problem, you can add a **DELAY 0** statement after each **STMB 13.**

You can use **STMB 13** with **STMB 12** to start up a job, if one is available. Use **STMB 13** to supply **HELLO** with answers to the account name, password, and directory/program questions.

STMB 14, *job-number, error, flag* *(DG/RDOS only)*

Sets the alternate IKEY or unpend flag value for *job-number* to the value of *flag* (0 or 1), and the job acts as if an interrupt occurred. Errors you can receive are:

51 Job not logged on

70 Invalid job number

If you specify any job that is running (other than your own job), regardless of whether it is running a program or sitting at the Business BASIC prompt, you get error = 0 (successful).

STMB 15, *port-number, characteristics* *(DG/RDOS only)*

Sets the multiplexor line to the specified line characteristics. Port number 0 corresponds to multiplexor line 2. If *characteristics* is set to -1, then Business BASIC returns the value to which the line characteristics was last set. For invalid lines, *characteristics* is set to -1. The line characteristics word contains the following:

```
Bits      0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
          0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

Content - x x x x x C C C C S S D D P P x (ALM, ULM)

Content - x x x x C C C C S S P P D D x x (ASLM, USAM)
```

continued **STMB**

where:

CCCC = Clock number

Baud rates corresponding to clock numbers are:

ALM: depend on how the board is jumpered

ULM:

0	0	4	134.5	8	9600	12	2400
1	19200	5	200	9	4800	13	300
2	50	6	600	10	1800	14	150
3	75	7	2400	11	1200	15	110

ASLM, USAM:

0	50	4	150	8	1800	12	4800
1	75	5	300	9	2000	13	7200
2	110	6	600	10	2400	14	9600
3	134.5	7	1200	11	3600	15	19200

SS= Number of stop bits

ALM, ULM:
00 = 1 stop bit
01 = 2 stop bits

ASLM, USAM:
01 = 1 stop bit
10 = 1 stop bit
11 = 2 stop bits

DD= Number of data bits

00 = 5 data bits
01 = 6 data bits
10 = 7 data bits
11 = 8 data bits

PP= Parity type

ALM,ULM:00 = none
01 = odd
10 = even
11 = mark

STMB *continued*

ASLM, USAM:
00 = odd, disabled
01 = odd, enabled
10 = even, disabled
11 = even, enabled

x = Meaningless. The interpreter will disregard these bits and will always return them to the user set to 0.

STMB 16, *flag*

Sets (*flag* = 1) or clears (*flag* = 0) the run-only flag in the current program. If set, this flag prevents a file from being listed or modified by non-AA accounts. For information about run-only programs, see your Business BASIC user's guide.

STMB 17

On AOS/VS and UNIX systems, performs an immediate **BYE** to log off the current job, completes system housekeeping, and returns the user to his previous level.

On DG/RDOS systems, executes the **.RTN** system call, which shuts down Business BASIC. If you are using Business BASIC multiplexor support, **STMB 17** also shuts down the multiplexor.

STMB 18, *address, value* *(DG/RDOS only)*

Returns in *value* the word stored at address in the operating system's address space.

STMB 19, *port-number, modem-status* *(DG/RDOS only)*

Sets the multiplexor line to the modem status. Port number 2 corresponds to multiplexor line 0. If status is set to −1, then Business BASIC returns the value to which the modem status was last set. For invalid lines, status is set to −1.

Note that the interpreter manipulates Request To Send and Data Terminal Ready when input and output are performed. Thus, you cannot write a Business BASIC program to drive a half-duplex device. **STMB 19** is used to reinitialize modem lines after the abnormal termination of a job logged on the line.

The modem status word contains the following:

```
Bits        0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
            0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

Content – x x x x x x x x x x x x x x R D (ALM, ULM)

Content – x x x x x x x x x x R x x x D x (ASLM, USAM)

where:

R = Request To Send

D = Data Terminal Ready

x = Meaningless. The interpreter will disregard these bits and will always return them to the user set to 0.

STMB 20

Performs an immediate **BYE** to log off the current job, completes system housekeeping, and returns to the log-on banner.

STMB 21

Reserved.

STMB 22, *byte-address, string-variable$* (*AOS/VS and UNIX*)

Returns the byte address of the string variable.

STMB 23, *address, variable* (*AOS/VS and UNIX*)

Returns the word address of the numeric variable in *address*.

STMB 24, *error*, *accumulator$* [*flag*] (*AOS/VS only*)

Performs a system call using the information in the accumulator string. The optional *flag* argument consists of bit flags modifying the other arguments. The value returned in *error* is a system error code or zero.

STMB *continued*

NOTE: Do not use **STMB 24** to make system calls that can be made with an existing Business BASIC statement, utility, or command. Particularly avoid the use of **STMB 24** system calls to manipulate files that are already being manipulated through Business BASIC statements. Mixing **STMB 24** with Business BASIC statements can cause unpredictable, and probably undesirable, results.

For example, if you use the **OPEN FILE** statement to open a file, do not close the file by using **STMB 24** to issue a **?CLOSE** call. You should instead use the **CLOSE FILE** statement. Similarly, use the **POSITION FILE** statement, not **STMB 24**, to issue a **?SPOS** call to position a file pointer. In addition, when using terminal type 8, do not use **STMB 24** to make a **?SDLM** call to change the delimiter table. Terminal type 8 sets the delimiter table for its own purposes.

On AOS systems, the accumulator (AC) string contents are:

Bytes	**Contents**
1–2	Contents of AC0 for system call.
3–4	Contents of AC1 for system call.
5–6	Contents of AC2 for system call.
7–8	System call number (system dependent).
9–10	Channel number for I/O calls (this may duplicate the contents of an accumulator).

On AOS/VS systems, the accumulator (AC) string contents are:

Bytes	**Contents**
1–4	Contents of AC0 for system call.
5–8	Contents of AC1 for system call.
9–12	Contents of AC2 for system call.
13–14	System call number.
15–16	Channel number for I/O calls.

You can use only Business BASIC channel numbers, not system channel numbers. If you need an address for an accumulator, you can use **STMB 22** or **STMB 23** to obtain it. When I/O is being performed, the *flag* argument specifies the Business BASIC channel. Depending on the flag, Business BASIC maps this number into the appropriate system channel and places the value into the proper accumulator for the call. The bit values for *flag* are as follows:

Bit	Mask	Meaning
15	1	I/O call (convert channel and put in AC2).
14	2	OPEN call returning channel number that must be equated to a Business BASIC channel.
13	4	Reserved; must be 0.
12	8	Convert channel into AC0.
11	16	Convert channel into AC1.

On AOS/VS systems, when you are using **STMB 24** with **STMB 22** and **STMB 23**, remember that addresses are 4 bytes long in a 32-bit environment. To alter an address, use the **SHFT** function. Suppose, for example, you have used **STMB 22** to retrieve the byte address of a string, but the system call requires a word address. You should use the following command to shift the address to the right: **SHFT(*byte-address*,−1)**. To change a word address to a byte address, use this command: **SHFT(*word-address*,1)**.

STMB 25, *user-channel, system-channel*

Maps the user channel to the system channel, making the system channel accessible to the user. On AOS/VS and DG/RDOS systems, this is used to access the push file, **BASIC.PS**, in utilities like **RNAM**, **VAR**, and **PD**. A **CLOSE** on a user channel opened with **STMB 25** does not close the physical channel but frees the user channel.

STMC

Statement and Command

Performs operating system calls.

AOS/VS	DG/RDOS	UNIX

Format

STMC *type,error,argument*[*,argument...*]

Arguments

type	A numeric expression or number designating the system call to be performed. Each type is discussed below.
error	A variable that receives the error returned by the operating system. A positive code is returned, so you must negate it to find it in the **BASIC.ER** file. A -1 is returned if the call is successful. On UNIX systems, if -60 is returned in the error variable, use SYS(42) to retrieve the AOS/VS system error code. If SYS(42) is equal to -276, use SYS(43) to determine the error that occurred during the STMC. Note that the SYS functions will be set only if an undefined error occurs (i.e.. the error variable is set to -60).
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each **STMC** requires one or more arguments. For an explanation of the required arguments for each system call, see the explanation for that type of STMC in Table 1-5.

What It Does

The STMC calls allow you to perform operating system calls from within Business BASIC. Refer to your operating system reference manual for descriptions of the operating system calls, the arguments needed, and the errors returned.

Business BASIC checks for compatibility of variable types; it does not check to see if the variables you use to receive values are large enough. Therefore, if a 36-byte string is required and you supply only a 20-byte string, the 16 bytes following the string will overwrite part of your program.

When you use a string literal as an argument, the literal must include a terminating null. Without the null, the operating system will read beyond the end of the literal and produce erroneous results. In order to pass the literal "ABC," you must write the literal as "ABC<0>".

When you use a string variable as an argument to receive a value, fill the variable with nulls before using the STMC.

continued **STMC**

Table 1-5 Summary of STMC Syntax Formats *(Continues)*

STMC Parameter	Data Type of Its Arguments	
STMC 0,	*error, filename$, sectors*	
STMC 1,	*error, directory-name$*	
STMC 2,	*error, attributes, channel*	
STMC 3,	*error, attributes, channel*	
STMC 4,	*error, numeric–array, channel*	
STMC 5,	*error, partition-name$, sectors*	
STMC 6,	*error, filename$*	
STMC 7,	*error, filename$*	
STMC 8,	*error, filename$*	
STMC 9,	*error, directory-name$*	
STMC 10,	*error, oldname$, newname$*	
STMC 11,	*error, code*	
STMC 12,	*error, program-name$, AC1value, AC2value*	
STMC 13,	*error, program-name$, AC1value, AC2value*	
STMC 14,	*error, flag*	
STMC 15,	*error, input-console$*	
STMC 16,	*error, output-console$*	
STMC 17,	*error, directory–name$*	
STMC 18,	*error, attributes, channel*	
STMC 19,	*error, system$*	
STMC 20,	*error, devicename$, flag*	
STMC 21,	*error, linkname$, resolution-name$*	
STMC 22,	*error, string$*	*(AOS/VS and UNIX)*
STMC 22,	*error, master-directory$*	*(DG/RDOS)*
STMC 23,	*error*	
STMC 24,	*error*	
STMC 25,	*error, oldname$, newname$*	
STMC 26	*error*	
STMC 27,	*error, devicename$*	

STMC

continued

Table 1-5 Summary of STMC Syntax Formats *(Concluded)*

STMC Parameter	Data Type of Its Arguments
STMC 28,	*error*
STMC 29,	*error, day, month, year-1968*
STMC 30,	*error, devicename$*
STMC 31,	*error, devicename$*
STMC 32,	*error, devicename$*
STMC 33,	*error, filename$, buffer*
STMC 34,	*error, seconds, minutes, hours*
STMC 35,	*error, linkname$*
STMC 36,	*error, channel*
STMC 37,	*error, frequency*
STMC 38,	*error, filename$, mask, channel*
STMC 39,	*error, filename$, mask, channel*
STMC 40,	*error, filename$, mask, channel*
STMC 41,	*error, filename$, mask, channel*
STMC 42,	*error, filename$, mask, channel*
STMC 43,	*error*
STMC 44,	*error, filename$, mask, channel*
STMC 45,	*error, filename$, sectors, date*
STMC 46,	*error, filename$, date*
STMC 47,	*error, filename$, date*
STMC 48,	*error, AC0*
STMC 49	Not used
STMC 50,	*error, string$, channel*
STMC 51,	*error, filename$, buffer*
STMC 52,	*error, filename$, sectors*
STMC 53,	*error, channel*
STMC 54,	*error, filename$, month, day, year*

Arguments that refer to channel numbers may refer to absolute operating system channels or to program-relative Business BASIC channels. A negative channel number refers to the operating system channel; e.g., –6 refers to operating system channel 6. A positive channel number refers to a file opened in the current Business BASIC program on this relative channel; e.g., 2 refers to the file opened in the current program on channel 2. Channel numbers for **STMC** calls range from –255 to 15. Operating system channel 0 is a special case that is passed as –0 (32768). You should use absolute channel numbers only to refer to Business BASIC channel numbers.

Business BASIC leaves a user and/or system channel allocated when an **STMC** attempts to open a file and fails. Therefore, if an error occurs, you should close the channel by using **STMC 53** if the channel is negative (i.e., a system channel) or by using **CLOSE FILE**(*channel*) if the channel is positive (i.e., a Business BASIC user channel).

DG/RDOS Systems

When you access a file through the standard Business BASIC statements (**OPEN**, **READ**, **PRINT**, **CLOSE**, etc.), the interpreter inserts directory specifiers before the user-specified filename to indicate the user's current directory, which is stored in the User Status Table, or the library directory (e.g., **$LIB**). This is not necessarily the same as the system directory, which is where DG/RDOS looks for files that are being created or opened with **STMC** calls when no directory is specified. Thus, a directory should always be specified explicitly when **STMC** statements are used.

AOS/VS and UNIX Systems

When you access a file through the standard Business BASIC statements, Business BASIC uses the operating system's search path mechanism.

You can explicitly supply directory specifiers only if a special bit in the User Status Table is set; otherwise, any filename containing a colon is rejected unless it is in the Reserved Filename table. **STMC** statements and reserved filenames are not handled in this manner; if you specify no directory, no directory is specified by the interpreter. Thus, the operating system will look for the file in the current system directory, unless it is a file that is treated specially by the operating system. Since it is possible to change the current system directory at runtime (using **!SDIR**, **DIR**, or **STMC 9**), links to reserved files and other special files which are accessed using **STMC** calls must be handled very carefully.

NOTE: Table 1–5 lists the syntax formats for each **STMC**. In this table, each string, string expression, or substring argument ends with a dollar sign ($); all other arguments are numeric. In addition, a string variable or numeric variable is a value that the **STMC** returns. A string expression or numeric expression is a value that you assign before executing the **STMC**; they can be either an expression or a variable. Not all the arguments are fully explained in the "How To Use It" section, so check the statement's syntax in Table 1–5 if you have a question about an argument.

How to Use It

This section includes a description of each **STMC**, its required arguments, and any special considerations for performing a particular system call.

STMC *continued*

STMC 0, *error, filename$, sectors*

For AOS/VS systems, **?CREATE** creates a contiguous file with all locations initialized to 0. The maximum value of *sectors* is 32767.

For DG/RDOS systems, **.CCONT** creates a contiguous file with all locations initialized to 0.

For UNIX systems, Business BASIC creates a file and initializes locations to make the file the desired length.

STMC 1, *error, directory-name$*

For AOS/VS systems, **?CREATE** creates a directory name

For DG/RDOS systems, **.CDIR** creates a directory name.

For UNIX systems, Business BASIC creates a subdirectory.

STMC 2, *error, attributes, channel* *(DG/RDOS only)*

.CHATR changes the attributes for the file opened on *channel.*

STMC 3, *error, attributes, channel* *(DG/RDOS only)*

.CHLAT changes the link attributes of the link file opened on *channel.*

STMC 4, *error, numeric–array, channel* *(DG/RDOS only)*

.CHSTS returns the 36-byte status table for the file opened on *channel.*

STMC 5, *error, partition-name$, sectors*

For AOS/VS, **?CREATE** creates a control point directory.

For DG/RDOS, **.CPART** creates a subpartition.

For UNIX, Business BASIC creates a subdirectory and ignores the *sectors* argument.

STMC 6, *error, filename$*

For AOS/VS systems, **?CREATE** creates a UDF-type file with the user's default ACL, an element size of four, and the default number of index levels.

For DG/RDOS systems, **.CRAND** creates a random file.

For UNIX systems, Business BASIC creates a data file with the user's access privileges.

continued **STMC**

STMC 7, *error*, *filename$*

For AOS/VS systems, **?CREATE** creates a UDF-type file with the user's default ACL, an element size of four, and the default number of index levels.

For DG/RDOS systems, **.CREAT** creates a sequential file.

For UNIX systems, Business BASIC creates a data file with the user's access privileges.

STMC 8, *error*, *filename$*

For AOS/VS systems, **?DELETE** deletes a file. If the file is a link entry, the resolution file is deleted.

For DG/RDOS systems, **.DELET** deletes a file. If the file is a link entry, the resolution file is deleted.

For UNIX systems, Business BASIC deletes a file. If the file is a Business BASIC link entry, the resolution file is deleted.

STMC 9, *error*, *directory-name$*

For AOS/VS systems, **?DIR** changes the working directory to *directory-name*. Read and execute access to *directory-name* are required.

For DG/RDOS systems, **.DIR** changes the current system directory. This does not change the default user directory, which is stored in the User Status Table.

For UNIX systems, Business BASIC changes the working directory. Search permission is required for all components of the destination directory's pathname.

STMC 10, *error*, *oldname$*, *newname$* *(DG/RDOS only)*

.EQIV renames a device.

STMC 11, *error*, *code*

For AOS/VS systems, **?RETURN** terminates Business BASIC, passing the error code to the previous level.

For DG/RDOS systems, **.ERTN** performs the same function.

For UNIX systems, Business BASIC returns control to the calling shell and passes the error code to the previous level.

STMC

continued

STMC 12, *error*, *program-name$*, *AC1value*, *AC2value*

For AOS/VS systems, **?PROCESS** creates a son process and blocks the current Business BASIC process until the son terminates. Values for AC1 and AC2 must be given but they are ignored. (Zero is the recommended value for AC1 and AC2.) To use this call, you need the ability to create a son process.

For DG/RDOS systems, **.EXBG** puts a checkpoint in the background and executes the program you specify. Business BASIC itself cannot have checkpoints.

For UNIX systems, Business BASIC creates a child process and blocks the Business BASIC process. *AC1value* and *AC2value* are not required and are ignored.

STMC 13, *error*, *program-name$*, *AC1value*, *AC2value*

For AOS/VS systems, **?PROCESS** creates a son process without blocking the current Business BASIC process until the son terminates. Values for AC1 and AC2 must be given but they are ignored. To use this call, you need the ability to create an unblocked son process.

For DG/RDOS systems, **.EXFG** executes *program-name* in the foreground partition. An AC2 value must be given but is ignored.

For UNIX systems, Business BASIC creates a child process without blocking the Business BASIC process. *AC1value* and *AC2value* are not required and are ignored.

STMC 14, *error*, *flag*

For AOS/VS systems, **?PSTAT** determines whether a process has sons. A 1 is returned in *flag* if a son process is found; otherwise, 0 is returned.

For DG/RDOS systems, **.FGND** determines whether a foreground program is already running. A 1 is returned in *flag* if a foreground program is found; otherwise, 0 is returned.

For UNIX systems, Business BASIC always returns 0 because a parent process doesn't know whether it has children or not.

STMC 15, *error*, *input-console$*

For AOS/VS and UNIX systems, this returns "@INPUT."

For DG/RDOS systems, **.GCIN** returns the name of the console input device for this ground.

STMC 16, *error*, *output-console$*

For AOS/VS and UNIX systems, this returns "@OUTPUT."

For DG/RDOS systems, **.GCOUT** returns the name of the console output device for this ground.

STMC 17, *error*, *directory-name$*

For AOS/VS systems, **?GNAME** returns the pathname of the current directory.

For DG/RDOS systems, **.GDIR** returns the name of the current Business BASIC system directory.

For UNIX systems, Business BASIC returns the pathname of the current working directory.

STMC 18, *error*, *attributes*, *channel* *(DG/RDOS only)*

.GTATR returns the attributes of the file opened on *channel*.

STMC 19, *error*, *system$* *(DG/RDOS only)*

.GSYS returns the current operating system's name.

STMC 20, *error*, *devicename$*, *flag* *(DG/RDOS only)*

.INIT initializes a device into Business BASIC. *flag* = −1 specifies full initialization; *flag* <> −1 specifies partial initialization.

STMC 21, *error*, *linkname$*, *resolution-name$*

For AOS/VS systems, **?CREATE** creates a link entry.

For DG/RDOS systems, **.LINK** creates a link entry.

For UNIX systems, Business BASIC creates a link entry. This link file is known only to Business BASIC as a link; it does not appear to be a link file to your operating system. For more information on Business BASIC link files, see *Using Business BASIC on DG/UX and INTERACTIVE UNIX Systems*.

STMC 22, *error*, *string$* *(AOS/VS and UNIX)*

Causes *string$* to have a length of zero.

STMC 22, *error*, *master-directory$* *(DG/RDOS only)*

.MDIR returns the name of the current master directory.

STMC 23, *error* *(DG/RDOS only)*

.ODIS disables console keyboard interrupts.

STMC *continued*

STMC 24, *error* *(DG/RDOS only)*

.OEBL enables console keyboard interrupts.

STMC 25, *error*, *oldname$*, *newname$*

For AOS/VS systems, **?RENAME** renames a file.

For DG/RDOS systems, **.RENAM** renames a file.

For UNIX systems, Business BASIC renames a file.

STMC 26, *error* *(DG/RDOS only)*

Reserved.

STMC 27, *error*, *devicename$* *(DG/RDOS only)*

.RLSE releases a previously initialized device.

STMC 28, *error*

For AOS/VS systems, **?RETURN** returns control to the program at the previous push level. No error code or message is returned to the previous level.

For DG/RDOS systems, **.RTN** returns control to the program at the previous push level.

For UNIX systems, Business BASIC returns control to the previous level.

STMC 29, *error*, *day*, *month*, *year-1968* *(DG/RDOS only)*

.SDAY sets the current system date.

STMC 30, *error*, *devicename$* *(DG/RDOS only)*

.SPDA disables spooling for a device.

STMC 31, *error*, *devicename$* *(DG/RDOS only)*

.SPEA enables spooling for a device.

STMC 32, *error*, *devicename$* *(DG/RDOS only)*

.SPKL kills spooling for a device.

STMC 33, *error*, *filename$*, *buffer* *(DG/RDOS only)*

.STAT returns the 36-byte status of the file you specify.

STMC 34, *error*, *seconds*, *minutes*, *hours* *(DG/RDOS only)*

.STOD sets the time of day.

STMC 35, *error*, *linkname$*

For AOS/VS systems, **?GLINK** removes *linkname$* if it is a link entry.

For DG/RDOS systems, **.ULNK** removes a link entry.

For UNIX systems, Business BASIC determines if the file is a Business BASIC link file and, if it is, deletes the link file, not the resolution file. It also deletes the associated shadow file. For more information on Business BASIC link files and shadow files, see *Using Business BASIC on DG/UX and INTERACTIVE UNIX Systems*.

STMC 36, *error*, *channel*

For AOS/VS systems, **?ESFF** updates the disk-resident copy of the file open on *channel* by flushing any modifed shared pages of the file to disk.

For DG/RDOS systems, **.UPDAT** updates the disk-resident copy of the file open on *channel*.

For UNIX systems, Business BASIC flushes any modified shared pages to disk.

STMC 37, *error*, *frequency*

For AOS/VS systems, **?GHRZ** returns the frequency of the system's real-time clock.

For DG/RDOS systems, **.GHRZ** returns the frequency of the system's real-time clock.

For UNIX systems, returns the system clock frequency.

STMC 38, *error*, *filename$*, *mask*, *channel* *(AOS/VS and DG/RDOS)*

For AOS/VS systems, **?GOPEN** opens a file for direct magnetic tape I/O. You must include the mask in the **STMC 38** format; however, the system ignores it.

For DG/RDOS systems, **.MTOPD** opens a file for direct magnetic tape I/O.

STMC *continued*

STMC 39, *error, filename$, mask, channel*

For AOS/VS systems, **?OPEN** opens a file exclusively on *channel* for appending and for input and output. The file must exist.

For DG/RDOS systems, **.APPEND** opens a file for appending.

For UNIX systems, Business BASIC opens a file exclusively.

On UNIX and AOS/VS systems, you must include the mask in the **STMC 39** format; however, the system ignores it.

STMC 40, *error, filename$, mask, channel*

For AOS/VS systems, **?OPEN** opens a file exclusively on *channel* for input and output. The file must exist.

For DG/RDOS systems, **.EOPEN** opens a file for exclusive use.

For UNIX systems, Business BASIC opens a file exclusively.

On UNIX and AOS/VS systems, you must include the mask in the **STMC 40** format; however, the system ignores it.

STMC 41, *error, filename$, mask, channel*

For AOS/VS systems, **?OPEN** opens a file for input only. The file must exist.

For DG/RDOS systems, **.ROPEN** opens a file for read-only access.

For UNIX systems, Business BASIC opens a file in read-only mode.

On UNIX and AOS/VS systems, you must include the mask in the **STMC 41** format; however, the system ignores it.

STMC 42, *error, filename$, mask, channel*

For AOS/VS systems, **?OPEN** opens a file for input and output. The file must exist.

For DG/RDOS systems, **.OPEN** opens a file for shared access.

For UNIX systems, Business BASIC opens a file for input and output.

On UNIX and AOS/VS systems, you must include the mask in the **STMC 42** format; however, the system ignores it.

STMC 43, *error*

For AOS/VS systems, **?BRKFL** terminates the Business BASIC process and creates a break file consisting of the memory image of the terminating process. The break file is named *?pid#.time*.**BRK**, where *pid#* is the PID of the terminated process and *time* is in the form *hh_mm_ss*.

For DG/RDOS systems, **.BREAK** terminates Business BASIC and saves an image of the Business BASIC interpreter in **BREAK.SV**.

For UNIX systems, Business BASIC terminates the runtime system and creates a break file.

STMC 44, *error, filename$, mask, channel*

For AOS/VS systems, **?OPEN** opens a file for input and output. The file must exist.

For DG/RDOS systems, **.TOPEN** transparently (without changing the date last written or accessed) opens the file you specify for exclusive access.

For UNIX systems, Business BASIC opens a file for input and output.

STMC 45, *error, filename$, sectors, date* *(DG/RDOS only)*

.TCCONT creates a contiguous file transparently (without changing the date last written or accessed). The time/date status area is set to the 6-byte *date*.

The format of *date* for STMCs **45**, **46**, and **47** is:

- Bytes 1-2 Julian value, date last accessed
- Bytes 3-4 Julian value, creation date
- Bytes 5-6 Julian value, creation time, one-half of seconds past midnight

STMC 46, *error, filename$, date* *(DG/RDOS only)*

.TCRND creates a random file transparently (without changing the date last written or accessed). The time/date status area is set to the 6-byte *date*.

STMC 47, *error, filename$, date* *(DG/RDOS only)*

.TCRET creates a sequential file transparently (without changing the date last written or accessed). The time/date status area is set to the 6-byte *date$*.

STMC 48, *error, AC0* *(DG/RDOS only)*

.GMEM gets the current memory allocation for the current ground.

STMC *continued*

STMC 49

Reserved.

STMC 50, *error, string$, channel*

For AOS/VS systems, **?WRITE** writes *string$* to the file open on *channel*. The write is data-sensitive, and the maximum length of *string$* is 134.

For DG/RDOS systems, **.WRL** writes a line to the file opened on *channel*.

For UNIX systems, Business BASIC performs a data-sensitive write.

STMC 51, *error, filename$, buffer* *(DG/RDOS only)*

.RSTAT returns the status of the resolution file when you specify a link entry.

STMC 52, *error, filename$, sectors*

For AOS/VS systems, **?CREATE** creates *filename$* with an element size of *sectors* and zero index levels, and writes one byte at the end of the file to cause AOS/VS to allocate space.

For DG/RDOS systems, **.CONN** creates a contiguous file without initializing the file to nulls.

For UNIX systems, Business BASIC creates a file and initializes locations to make the file the desired length.

STMC 53, *error, channel* *(DG/RDOS only)*

.CLOSE closes the file opened on *channel*. *channel* must be negative; use the **CLOSE FILE** statement for positive channels.

STMC 54, *error, filename$, month, day, year* *(AOS/VS and UNIX)*

This statement returns the date on which *filename$* was last modified. The *year* variable receives a two-digit version of the year; i.e., 91, not 1991. If no error occurs, *error* is set to -1.

STMD

Statement and Command

Sends messages to and receives responses from user terminals.

DG/RDOS

Format

STMD *type,port-number,argument*[*,argument...*]

Arguments

type	The number of the system call to be performed. Only types 0 and 1 are valid. Both types are discussed below.
port-number	A number, numeric expression, or variable that defines the port number of the user terminal you want to contact.
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each **STMD** requires one or more arguments. For an explanation of the required arguments for a particular system call, see the description for that type of **STMD**.

What It Does

STMD statements and commands allow the system manager (an AA account) to send messages to and receive responses from any user's terminal.

The **STMD** statements and commands send the message string literally, so if you need carriage returns or control characters, you have to enclose their ASCII decimal values in angle brackets. For example, <13> causes a carriage return or line feed to occur at the position in the message string where it appears. Output of the message string stops on a null or end of string, whichever comes first.

The **STMD** statements and commands are:

STMD 0, *port-number*, 0, *message$*

STMD 0, *port-number*, *wait*, *message$*, *reply$*

STMD 1, *port-number*, 0, *message$*

STMD 1, *port-number*, *wait*, *message$*, *reply$*

STMD 1 overrides any no-message flag set by the terminal; **STMD 0** does not. You must supply the port number and your message.

STMD

continued

How to Use It

You can supply either a 0 for no wait time and no reply, or a numeric expression for wait and a string variable *reply$* to receive a reply. If you specify wait and supply a string variable *reply$*, Business BASIC waits for a reply according to the rules of **TINPUT USING**. Each character received into *reply$* updates the current string length. If you do not get a reply in the specified amount of time, Business BASIC executes the next statement in your program (or returns you to keyboard mode) without changing the contents of *reply$*. You can then check **SYS(22)** to see if a timeout occurred.

You cannot request a reply from a terminal that is not logged on to Business BASIC if you are using operating system multiplexor support. However, if the terminal is logged on, you can request **STMD** with reply.

During the execution of an **STMD**, it is as if the job that issued the call had attached to the destination terminal. Interrupts are disabled for the job issuing the call. When no reply is requested, any characters struck at the destination terminal while the message is being displayed will be echoed later at the requesting job's terminal, and any input at the requesting terminal will be discarded.

The example below shows an **STMD** that forces a message to terminal number 1. The bell character (<7>) is included, along with a carriage return (<13>). The **STMD** waits 30 seconds for a reply; if a timeout occurs, line 110 directs control back to line 100. The *wait* flag is in tenths of seconds.

```
0100 STMD 1,1,300,"<13>Tell me your name in 30 seconds!<7>",REPLY$
0110 IF SYS(22)=0 THEN GOTO 0100
0120 PRINT REPLY$
```

STME

Statement and Command

Performs operating system calls.

AOS/VS	UNIX

Format

STME *type,error,argument(,argument...)*

Arguments

type	A numeric expression or number designating the system call to be performed. Each type is discussed below.
error	A variable that receives the error returned by the operating system. A positive code is returned, so you must negate it to find it in the **BASIC.ER** file. A -1 is returned if the call is successful. On UNIX systems, if -276 is returned in the error variable, use SYS(43) to retrieve the UNIX system error code that occurred during the STME. The SYS functions are set only if an undefined error occurs (i.e., the error variable is set to -276).
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each **STME** requires one or more arguments. For an explanation of the required arguments for each system call, see the explanation for that type of **STME** in Table 1-6.

What It Does

The **STME** statement allows you to perform operating system calls from within Business BASIC. Refer to your operating system reference manual for descriptions of the operating system calls, the arguments needed, and the errors returned.

Contiguous files in AOS/VS are created by specifying an element size large enough for the entire file and zero index levels.

String variables that receive a value must be filled with nulls before the **STME** is performed. You must also use the **DIM** (dimension) statement or command to specify the exact size stated in your operating system manual in which system calls are described. Business BASIC does not check to see if the variables you use to receive values are large enough. For this reason, if a 36-byte string is required and you supply only a 20-byte string, the 16 bytes following the string may be appended to the string and overwrite part of your program. To avoid this problem, include a terminating null when you assign string arguments.

STME

continued

NOTE: Table 1-6 lists the syntax formats for each **STME**. In this table, each string, string expression, or substring argument ends with a dollar sign ($); all other arguments are numeric. In addition, a string variable or numeric variable is a value that the **STME** returns. A string expression or numeric expression is a value that you assign before executing the **STME**; both the string expression and the numeric expression can be either an expression or a variable. Since the arguments are not all fully explained in the "How To Use It" section, check the statement's syntax in Table 1-6 if you have a question about whether an argument requires a numeric or string argument.

Before using any **STME** calls you should be thoroughly familiar with system calls.

Table 1-6 Summary of STME Syntax Formats

STME Parameter	Data Type of Its Arguments
STME 0,	*error, channel, pointer, filename$, template$*
STME 1,	*error, username$*
STME 2,	*error, PID*
STME 3,	*error, message$*
STME 4,	*error, filename$, pathname$*
STME 5,	*error, buffer$, channel*
STME 6,	*error, filename$, buffer$*
STME 7,	*error, filename$, buffer$*
STME 8,	*error, filename$, sectors, date$*
STME 9,	*error, filename$, date$*
STME 10,	*error, pathname$, channel*
STME 11,	*error, linkname$, resolution-name$*
STME 12,	*error, buffer$*
STME 13,	*error, buffer$*
STME 14,	*error, buffer$*
STME 15,	*error, PID, buffer$*
STME 16,	*error, consolename$, buffer$*
STME 17,	*error, buffer$*
STME 18,	*error, buffer$*
STME 19,	*error, program-name$, buffer$*
STME 20,	*error, control$, send$*
STME 21,	*error, control$, receive$*
STME 22,	*error, control$, send$, receive$*
STME 23,	*error, portname$, local-port-num*
STME 24,	*error, global-port-num, PID, local-port-num*
STME 25,	*error, portname$, global-port-num*
STME 26,	*error, local-port-num, global-port-num*
STME 27,	*error, string-variable$, flag*

How to Use It

This section includes a description of each **STME**, its required arguments, and any special considerations for performing a particular system call.

STME 0, *error, channel, pointer, filename$, template$* *(AOS/VS only)*

For AOS/VS systems, **?GNFN**, starting from the position *pointer*, returns the next filename matching *template$* in the directory open on *channel*.

STME 1, *error, username$*

For AOS/VS systems, **?GUNM** returns the process's username.

For UNIX systems, **getuid(2)** returns the process's username.

STME 2, *error, PID*

For AOS/VS systems, **?PNAME** returns the process's PID.

For UNIX systems, **getpid(2)** returns the process's PID.

STME 3, *error, message$* *(AOS/VS only)*

?GTMES returns a CLI message.

STMU 4 performs a similar function on UNIX systems.

STME 4, *error, filename$, pathname$*

For AOS/VS systems, **?GNAME** returns the complete pathname of *filename$*.

For UNIX systems, returns the complete pathname of *filename$*.

STME 5, *error, buffer$, channel* *(AOS/VS only)*

?FSTAT returns the status packet for *channel* in the string variable *buffer$*.

STMU 0 performs a similar function on UNIX systems.

STME 6, *error, filename$, buffer$* *(AOS/VS only)*

?FSTAT returns the status of a file to ***buffer$***.

STMU 1 performs a similar function on UNIX systems.

STME 7, *error, filename$, buffer$* *(AOS/VS only)*

?FSTAT returns the status of the resolution file to ***buffer$*** when you specify a link entry.

STMU 2 performs a similar function on UNIX systems.

STME 8, *error, filename$, sectors, date$* *(AOS/VS only)*

?CREATE creates a contiguous file of the size specified in ***sectors***. The time/date status area is set to the 12-byte ***date$***.

STMU 3 performs a similar function on UNIX systems.

This example illustrates the creation of a file named **TESTFILE** where the last accessed date and last modified date are the same as the date created.

```
00010 DIM DATE$[12]
00020 LET E=0
00030 STMA 12,E,SYS(2),SYS(1),SYS(3)-1900 :Convert the system date
                                          :to Julian.
00040 LET DATE$[1,2]=CHR$(E,2)            :Date the file was created.
00050 LET DATE$[3,4]=CHR$(SYS(0)/2,2)     :Time file was created in
                                          : bi-seconds.
00060 LET DATE$[5,8]=DATE$[1,4]           :Date & time last accesed.
00070 LET DATE$[9,12]=DATE$[1,4]          :Date & time last modified.
00080 STME 8,E,"TESTFILE<0>",4,DATE$
00090 END
```

STME 9, *error, filename$, date$* *(AOS/VS only)*

?CREATE creates a file. The time/date status area is set to the 12-byte ***date$***.

STMU 3 performs a similar function on UNIX systems.

STME 10, *error, pathname$, channel*

For AOS/VS systems, **?GNAM** returns the complete pathname of the file open on *channel*.

STME

continued

For UNIX systems, returns the complete pathname of the file open on *channel*.

STME 11, *error, linkname$, resolution-name$*

For AOS/VS systems, **?GLINK** returns the resolution name of a link.

For UNIX systems, returns the resolution name of a link.

STME 12, *error, buffer$* *(AOS/VS only)*

?GCHR returns the device characteristics of @INPUT to *buffer$* (6 bytes).

STME 13, *error, buffer$* *(AOS/VS only)*

?GCHR returns the device characteristics of @OUTPUT to *buffer$* (6 bytes).

STME 14, *error, buffer$* *(AOS/VS only)*

?SCHR sets the device characteristics of @INPUT to *buffer$* (6 bytes).

STME 15, *error, PID, buffer$* *(AOS/VS only)*

?SEND sends the message in *buffer$* to the process identified by *PID*.

STME 16, *error, consolename$, buffer$* *(AOS/VS only)*

?SEND sends the message in *buffer$* to the process controlling the console.

STME 17, *error, buffer$*

For AOS/VS systems, Business BASIC puts the 256-bit (32-byte) delimiter table for @INPUT into *buffer$*. At initialization, this table is all zero bits. This table displays the delimiters you have set with an **STME 18** statement.

For UNIX systems, Business BASIC places the delimiter table for standard input in a buffer.

STME 18, *error, buffer$*

For AOS/VS systems, Business BASIC sets the 256-bit (32-byte) delimiter table for @INPUT.

For UNIX systems, Business BASIC sets the delimiter table for standard input.

STME 19, *error, program-name$, buffer$*

For AOS/VS systems, **?PROC** executes *program-name$*, passing the contents of *buffer$* to the new process as the initial IPC. The Business BASIC process is blocked until *program-name$* finishes.

buffer$* must end with a <0> if data is stored in the variable. If *buffer$ is zero length, and *program-name* is **:CLI.PR**, the AOS/VS CLI is executed much like the current implementation using **STMC 12**. If *buffer$* contains an AOS/VS CLI command, and **:CLI.PR** is in *program-name*, the AOS/VS CLI command is executed, and control will be passed immediately back to the Business BASIC program after the command completes. *buffer$* may contain several AOS/VS CLI commands separated by semicolons.

program-name$* and *buffer$ must be string variables; substrings and literals are not supported.

For UNIX systems, *program–name$* is executed and receives the contents of *buffer$* as its arguments. Separate each argument from *buffer$* by one or more spaces. The Business BASIC process is blocked until *program–name$* finishes. The argument *program–name$* must be an executable that will process the argument list if supplied in *buffer$*.

For example, if *program–name$* is **CLI.PR<0>**, there must be an executable called **CLI.PR** that will read the AOS/VS CLI command from its argument list and perform the UNIX emulation of the CLI command.

Proper usage of **STME 19** is illustrated below:

```
00030 DIM PROG$[100],BUFF$[100]
00040 LET ER=0
00050 LET PROG$=":CLI.PR<0>"
00060 LET BUFF$=""
00070 INPUT "COMMAND? ",BUFF$
00080 IF LEN(BUFF$)<>0 THEN LET BUFF$[0]="<0>"
00090 STME 19,ER,PROG$,BUFF$
```

Note that BUFF$ is null if you are creating a CLI process to issue multiple commands; otherwise, BUFF$ contains the CLI command followed by a null.

STME 20, *error, control$, send$*

The contents of *send$* are sent via interprocess communications. **?ISEND** is called directly on AOS/VS while **?ISEND** is emulated on UNIX.

STME *continued*

control$* is a 24-character string. On AOS Business BASIC, *control$ is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IDPH	?IDPL
9	?IOPN	?ILTH
13		?IPTR
17		
21		

On AOS/VS and UNIX Business BASIC, *control$* is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IDPH	
9	?IOPN	?ILTH
13	?IPTR	
17	0	?IRLT
21	?IRPT	

You must set the following flags:

?ISFL	System Flags
?IUFL	User Flags
?IDPH	Destination Port, High
?IDPL	Destination Port, Low
?IOPN	Origin Port

You can set the following flag:

?IPTR	Secondary user flags if the message length is zero.

Business BASIC calculates the following:

?ILTH	Message length in words. Note that the message will be rounded up to an even number of bytes.
?IPTR	If the message length is nonzero, this points to the message. If the message length is zero, the user supplied value (?IPTR) is sent.

STME 21, *error, control$, receive$*

?IREC receives the contents of *receive$* via interprocess communications. ?IREC is called directly on AOS/VS while ?IREC is emulated on UNIX.

control$* is a 24-character string. On AOS Business BASIC, *control$ is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IOPH	?IOPL
9	?IDPN	?ILTH
13		?IPTR
17		
21		

STME

continued

On AOS/VS and UNIX Business BASIC, *control$* is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IOPH	
9	?IDPN	?ILTH
13	?IPTR	
17	0	?IRLT
21	?IRPT	

You must set the following flags:

?ISFL	System Flags
?IUFL	User Flags
?IOPH	Origin Port, High
?IOPL	Origin Port, Low
?IDPN	Destination Port
?ILTH	The maximum message length will be set to the current word length of *receive$*, after *receive$* has been truncated to an even number of bytes. This means that receive$ should be initialized.
?IPTR	The message pointer will be set to point to *receive$*. Note, however, that if a zero length message is received, Business BASIC will not update *receive$* (you can retrieve the contents of ?IPTR).

control$ is updated even if an error occurs.

STME 22, *error, control$, send$, receive$*

?IS.R sends the contents of *send$* via interprocess communications and then receives an interprocess communications message in *receive$*. **?IS.R** is called directly on AOS/VS while **?IS.R** is emulated on UNIX.

continued **STME**

control$* is a 24-character string. On AOS Business BASIC, *control$ is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IDPH/?IOPH	?IDPL/?IOPL
9	?IOPN/?IDPN	?ILTH
13		?IPTR
17		?IRLT
21		?IRPT

On AOS/VS and UNIX Business BASIC, *control$* is formatted as follows:

Byte #		
1	?ISFL	?IUFL
5	?IOPH	
9	?IDPN	?ILTH
13	?IPTR	
17	0	?IRLT
21	?IRPT	

You must set the following flags:

?ISFL	System Flags
?IUFL	User Flags
?IDPH	Destination Port, High
?IOPH	Origin Port, High
?IDPL	Destination Port, Low
?IOPL	Origin Port, Low

STME *continued*

?IOPN	Origin Port
?IDPN	Destination Port

Business BASIC performs the following functions:

?ILTH	Send message length in words. Note that the message will be rounded up to an even number of bytes.
?IPTR	If the message length is non-zero, this points to the message. If the message length is zero, the user-supplied value is sent.
?IRLT	This maximum message length will be set to the current word length of *receive$*, truncated to the nearest word boundary. This means that *receive$* should be initialized.
?IRPT	This message pointer will be set to point to *receive$*.

control$ is updated even if an error occurs.

STME 23, *error, portname$, local-port-num*

?CREATE creates an IPC file entry with the appropriate name and local port. **?CREATE** is called directly on AOS/VS while **?CREATE** is emulated on UNIX. Note that AOS/VS requires you to be in the directory in which Business BASIC was initially invoked in order to execute this call successfully.

STME 24, *error, global-port-num, PID, local-port-num*

?GPORT finds the owner of a global port. *global-port-num* is set before the call. *PID* and *local-port* are returned. **?GPORT** is called directly on AOS/VS while **?GPORT** is emulated on UNIX.

STME 25, *error, portname$, global-port-num*

?ILKUP looks up a port number. *portname$* is set before the call. *global-port* is returned. **?ILKUP** is called directly on AOS/VS while **?ILKUP** is emulated on UNIX.

STME 26, *error, local-port-num, global-port-num*

?TPORT translates a port number. *local-port-num* is set before the call. *global-port-num* is returned. **?TPORT** is called directly on AOS/VS while **?TPORT** is emulated on UNIX.

On UNIX, since a PID is 4 bytes, the global port number contains a unique job number for each obit client instead of the PID number.

Remember a global port number in AOS/VS is slightly different from one under AOS. The 32-bit number is constructed as follows:

AOS/VS Bits	
0-15	PID #
16-19	Ring #
20-31	Local port #

In AOS, the ring number does not apply; those bits are always 0.

Since the global port number is different on each operating system, you should not depend on this format.

STME 27, *error, string-variable$, flag*

The contents of *string-variable* are moved to an I/O buffer for display by the **INPUT** statement.

When you move a string with **STME 27**, you can use the screen edit keys to modify the contents of the string when you display it using an **INPUT** statement. This editing feature is available only for terminal types 8 and 9. *string-variable* is the string to be displayed; it must not be a literal. *flag* is a number or numeric variable that controls the display of *string-variable*. When *flag* is set to 1, *string-variable* is displayed. When flag is set to 0, *string-variable* is not displayed unless you enter Ctrl–A.

NOTE: On AOS/VS systems and UNIX Business BASIC systems executing in DG mode, if you press the Carriage Return key while the cursor is positioned within the string, the string is truncated at the cursor position. (You receive this same result by pressing Carriage Return while in the AOS/VS CLI.)

STMU

Statement and Command

Performs operating system calls.

UNIX

Format

STMU *type,error,argument*(*,argument...*)

Arguments

type	A numeric expression or number designating the system call to be performed. Each type is discussed below.
error	A variable that receives the error returned by the operating system. You must use the **UERM$** function to obtain the UNIX error message. A –1 is returned if the call is successful.
argument	A number, numeric expression, or variable that further defines the system call. The meaning of *argument* depends on the *type* you specify. Each **STMU** requires one or more arguments. For an explanation of the required arguments for a particular system call, see the description for that type of **STMU**.

What It Does

The **STMU** statement allows you to perform operating system calls from within Business BASIC. Refer to your operating system reference manual for descriptions of the operating system calls, the arguments needed, and the errors returned.

How to Use It

This section includes a description of each **STMU** and its required arguments. Any special considerations for performing a particular system call are noted in the explanation of that type of **STMU**.

STMU 0, *error, buffer$, channel*

fstat(2) returns to *buffer$* the **fstat** status packet for the file opened on *channel.*

STME 5 performs a similar function on AOS/VS systems.

STMU 1, *error, filename$, buffer$*

stat(2) returns the status of *filename$* to *buffer$.*

STME 6 performs a similar function on AOS/VS systems.

STMU 2, *error, filename$, buffer$*

stat(2) returns to *buffer$* the status of the resolution file for the link file specified in *filename$*.

STME 7 performs a similar function on AOS/VS systems.

STMU 3, *error, filename$, date$*

access(2), **creat(2)**, **close(2)**, and **utime(2)** create *filename$* and set the date/time stamp, using the values you specify.

STME 8 and **STME 9** perform similar functions on AOS/VS systems.

STMU 4, *error, message$*

getopt(3C) returns the options (switches) used during invocation of Business BASIC.

STME 3 performs a similar function on AOS/VS systems.

STMU 5, *error, envvar$, contents$*

getenv(3C) returns the contents of the environment variable named in *envvar$* to the *contents$* string. Make sure the string *contents$* is dimensioned large enough to receive the contents of the environment variable requested or an error 34 (`Function argument error`) is returned.

STOP

Statements

Stops program execution.

AOS/VS	DG/RDOS	UNIX

Format

STOP

What It Does

STOP halts the execution of a program. Usually a program ends when it runs out of statements or when it encounters an **END** statement. **STOP** is useful for debugging programs because it prints out the line number where the stop occurred.

When BASIC encounters **STOP**, the program stops and a message appears at your terminal:

```
STOP AT xxxxx
```

where xxxxx is the line number of the **STOP** statement. The terminal is then put in keyboard mode to wait for your next command.

How to Use It

Use **STOP** as a program statement only. It can be its own statement, or either it can be a statement after **IF...THEN**, or **ON ERR...THEN**, or **ON IKEY...THEN**. Use **STOP** statements anywhere in a program.

Example

Stop on an error, on an IKEY, if the input equals 0, or at the end of the program.

```
00010 ON ERR THEN STOP
00020 ON IKEY THEN STOP
00030 INPUT X
00040 IF X=0 THEN STOP
00050 PRINT X
00060 STOP
```

STRPOS

Statement and Command

Finds the starting position of a substring within a string.

AOS/VS	DG/RDOS	UNIX

Format

STRPOS *position,string-expression1,string-expression2*[*,start,scan-increment*]

Arguments

position	The variable that receives the relative position in *string-expression1* of the first occurrence of *string-expression2*.
string-expression	String variables, string literals in quotation marks, string expressions, or string array elements (UNIX only).
start	An optional argument which indicates where in *string-expression1* to begin the **STRPOS** operation.
scan-increment	An optional argument indicating the increment value for the scan pointer after each comparison of *string-expression1* to *string-expression2*. The pointer will continue to be incremented until a match is found or the end of *string-expression1* is reached.

What It Does

String-expression1 is scanned for the first occurrence of *string-expression2* as a substring. When the optional arguments *start* and *scan-increment* are present, the scan of *string-expression1* begins with the relative character at the place indicated by *start*. The pointer for the scan is incremented by *scan-increment* after each comparison until a match is found or the end of *string-expression1* is reached. If the entire *string-expression2* is found within *string-expression1*, *position* returns the relative position of the first character. If *string-expression2* is not found within *string-expression1*, *position* returns zero. If both *string-expression1* and *string-expression2* are null, the position returned is 1. If *string-expression1* is null and *string-expression2* is *not* null, the position returned is 0. If *string-expression1* is *not* null and *string-expression2* is null, the position returned is 1.

STRPOS

continued

Examples

1. Find the first occurrence of string B$ in string A$. Record that position in P.

```
00010 DIM A$[100],B$[100]
00020 LET P=0                :Must be defined before STRPOS
00030 INPUT A$,B$;           :Target string, search string
00040 STRPOS P,A$,B$         :Find B$ in A$
00050 PRINT ," P=";P         :Where was it found?
00060 GOTO 00030

* RUN                         results comments

? ABCDEFGHIJKLM ? KLM        P= 11   found it starting in col 11
? 9876543210 ? 54            P= 5
? NOW IS THE ? TH            P= 8
? THE QUICK ? BROWN          P= 0    did not find it
? ABCDEFGHIJKL ? XYZ         P= 0    did not find it
? ABBCCCDDDDEEEEE ? DDE      P= 9
```

2. This example uses **STRPOS** to find all occurrences of a string.

```
00010 DIM A$[100],B$[100]
00020 LET P=0                 :Define variable before STRPOS.
00030 LET S=1                 :Scan from first character.
00040 INPUT A$,B$," INC=",I   :Target string, search
                              :string, increment.
00050 STRPOS P,A$,B$,S,I      :Find B$ in A$ starting at S in
                              :steps of I.
00060 PRINT " P=";P;                  :Print  matching locations.
00070 LET S=P+I               :Continue from this location.
00080 IF P THEN GOTO 00050    :Scan until P=0 (end of A$).
00090 PRINT
00100 GOTO 00030

* RUN
? ABCABCABCABCABCABCABC ? AB INC=1
P= 1 P= 4 P= 7 P= 10 P= 13 P= 16 P= 19   P= 0
? ABCABCABCABCABCABCABC ? AB INC=2
P= 1 P= 7 P= 13 P= 19 P= 0
? ABCDEFGHIJKL ? XYZ INC=1
P= 0
?
IKEY AT 00040
*
```

SWAP

Statement and Command

Executes a utility or another program and then returns to the current program.

AOS/VS	DG/RDOS	UNIX

Format

$$\text{SWAP} \left\{ \begin{array}{l} \textit{"filename"} \\ \textit{string-variable} \end{array} \right\} \left[\text{THEN} \left\{ \begin{array}{l} \text{GOTO } \textit{line-number} \\ \text{CON} \end{array} \right\} \right]$$

Arguments

filename	A literal filename (in quotation marks) for a program file, but not for an ASCII listing or source file. The file must contain a Business BASIC program or Business BASIC utility program.
string-variable	A string variable already dimensioned and assigned a filename value.
line-number	A valid line number in the program to which you want to swap. Execution begins at *line-number* rather than the beginning of the program, and the program retains the values it had when it was saved.

What It Does

Like **CHAIN**, **SWAP** executes another program from the program that is running. However, **SWAP** does not clear the current program from working storage. After the **SWAP**, you return to the current program. Using **SWAP THEN CON** starts the new program at the point where it last stopped before it was saved and retains the variables' values as if a **CON** had occurred. Using **SWAP THEN GOTO** *line-number* starts the new program at *line-number* and retains the values the variables had when the program was saved, as if a **RUN** *line-number* had occurred.

SWAP searches for *filename* in your directory; if not found, it searches the library directory (in AOS/VS and UNIX systems, it follows your search path). If **SWAP** still does not find *filename*, it gives you an error message. **SWAP** does not change the status of files.

NOTE: During the execution of a **SWAP** statement, keyboard interrupts are ignored. This means that you may not be able to interrupt a series of short programs executed using **SWAP** or **CHAIN** statements.

SWAP

continued

How to Use It

Use **SWAP** as a program statement or a keyboard mode command. As a keyboard mode command, the keyword **SWAP** is optional. Enter the filename in quotation marks. If used as a keyboard mode command, **SWAP** executes the new program and returns you to keyboard mode, preserving the contents of working storage. If used as a program statement, **SWAP** executes the new program and, when the new program stops, returns you to the original program that had the **SWAP** statement.

Examples

1. Start executing **PROG102** at line 100 and then return to line 910.

```
00900  SWAP "PROG102" THEN GOTO 00100
00910  PRINT "PROG102 FINISHED"
```

2. This example executes **PROG3** at the point where it last stopped before it was saved.

   ***SWAP "PROG3" THEN CON**

3. This example swaps to the INDEXBLD utility program.

   ***"INDEXBLD"**

4. This example swaps to **PROG3** and starts at line 200.

   ***"PROG3" THEN GOTO 00200**

SYS

Function

Returns system information.

AOS/VS	DG/RDOS	UNIX

Format

SYS(*item*)

Arguments

item	A numeric expression or variable for one of the items listed in "What It Does."

What It Does

SYS returns the information indicated by *item*, where *item* is one of the following:

0 Time of day since midnight (seconds past 00:00).

1 Day of the month (1–31).

2 Month of the year (1–12).

3 Year in four digits (e.g., 1984).

4 Terminal port number (-1 if detached or under AOS/VS BATCH or UNIX background). On DG/RDOS systems:

Background console = 0

Foreground console = 1

Multiplexor consoles = 2 to *n*

AOS/VS systems return the console number. You may need to check bit 4 of **SYS(30)** to differentiate between physical and virtual consoles. For AOS/VS systems, any console not a son of EXEC returns 0 for **SYS(4)**. You may need to use **SYS(33)** to differentiate between devices.

UNIX systems return the console number specified in the **ttymap** file. In order for a Business BASIC process to be considered a UNIX background process, both standard input and standard output must be redirected from and to a file, respectively. If the device is not a tty, **SYS(4)** returns -1. You may need to use **SYS(33)** to differentiate between devices.

5 CPU time used, in tenths of seconds.

6 I/O usage. (In DG/RDOS systems, the number of system calls made; in AOS/VS systems, the number of blocks read or written; in UNIX systems, always 0.)

SYS *continued*

7	Error code of the last or current error; used with the **ERM$** function and the error messages in the **BASIC.ER** file. If **SYS(7)** = –60, see **SYS(31)**. On AOS/VS and UNIX systems, use **SYS(40)** and **SYS(41)** instead of SYS(7), and **SYS(42)** instead of **SYS(31)**.
8	Channel number of the file most recently referenced in a file I/O statement. The value of **SYS(7)** is invalid if no runtime error occurred, and the value of **SYS(8)** is invalid if no file was opened.
9	In DG/RDOS systems, job number; in AOS/VS and UNIX systems, PID (process ID).
10	Alternate IKEY or unpend flag; set to 1 if you use the alternate IKEY or unpend key.
11	Global switches used when the Business BASIC system was brought up.
12	Time of day (seconds past the last minute).
13	Time of day (minutes past the last hour).
14	Time of day (hours since midnight).
15	Status word 1 (from User Status Table).
16	Status word 2 (from User Status Table).
17	Program length in bytes.
18	Data length in bytes.
19	Maximum program and data length.
20	Statement number of last error or IKEY.
21	Current CPU switch settings (always 0 in AOS/VS and UNIX systems).
22	Time remaining after last **INPUT** or **TINPUT** (in seconds).
23	Log-on time (minutes past 00:00).
24	Current push level.
25	Processor number (0=single, 1=secondary) (always 0 in AOS/VS and UNIX systems).
26	Status of IKEY flag (1 if IKEY occurred). Reset by **STMA 8,3**.
27	Time of day as *hhmmss* (*hh* is hours, *mm* is minutes, *ss* is seconds).
28	Date as *mmddyy* (*mm* is month, *dd* is day, *yy* is year).
29	In DG/RDOS systems, **SYS(29)** returns a unique number that is determined by the number of times the **SYS(29)** function has been accessed by all jobs. In AOS/VS and UNIX systems, **SYS(29)** returns the process's PID number plus a multiple of 1000. That multiple is determined by the number of times the **SYS(29)** function has been accessed by that process.

30 System status. Business BASIC returns system status information in bit flag format as follows:

Bits 0–3	Reserved
Bit 4 (2048)	Virtual console
Bit 5 (1024)	Reserved
Bit 6 (512)	Single user system
Bit 7 (256)	AOS/VS operating system
Bit 8 (128)	Reserved
Bit 9 (64)	AOS operating system
Bit 10 (32)	RDOS operating system
Bit 11 (16)	Reserved
Bit 12 (8)	UNIX operating system
Bit 13 (4)	Quadruple precision
Bit 14 (2)	Triple precision
Bit 15 (1)	Double precision

A 257, for example, would signify a double precision system running under AOS/VS, because its binary representation would be:

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1

See the **AND** statement for more information about bit positions.

31 Used with **AERM$** and the error messages in the **BASIC.ER** file. If **SYS(7)** is −60, **SYS(31)** returns the AOS/VS or UNIX system error code of the last error that occurred. On AOS/VS and UNIX systems, you should also check **SYS(40)**, **SYS(41)**, **SYS(42)**, and **SYS(43)**. If **SYS(7)** is any value other than −60, **SYS(31)** is undefined. If **SYS(31)** = −276, see **SYS(43)**.

32 Status word 3 (from User Status Table).

33 On UNIX systems, returns the device number. If the device is not a tty device, **SYS(33)** returns −1. On AOS/VS systems, returns a unique device code for each console type under EXEC. If the console is not under EXEC, or your process is under BATCH, SYS(33) returns −1.

34 On AOS/VS and UNIX systems, returns the next available channel number, beginning with the lowest number available. If all channels are in use, **SYS(34)** returns −1.

35–39 Reserved.

40	On AOS/VS and UNIX systems, the error code of the last or current error; used with the **ERM$** function and the error messages in the **BASIC.ER** file. This function replaces **SYS(7)** on UNIX systems. If **SYS(40)** has a negative value, use **SYS(41)** instead.
41	On AOS/VS and UNIX systems, the error code of the last or current error; used with the **ERM$** function and the error messages in the **BASIC.ER** file. This function replaces **SYS(7)** on UNIX systems. If **SYS(40)** has a negative value, use **SYS(41)** instead.
42	On AOS/VS and UNIX systems, used with **AERM$** and the error messages in the **BASIC.ER** file. This function replaces **SYS(31)** on UNIX systems. If **SYS(40)** is negative and **SYS(41)** is –60, **SYS(42)** returns the AOS/VS or UNIX system error code of the last error that occurred. If **SYS(41)** is any value other than –60, **SYS(42)** is undefined. If **SYS(42)** = –276, see **SYS(43)**. On UNIX systems, if **SYS(42)** = –276, see **SYS(43)**.
43	On AOS/VS and UNIX systems, used with **UERM$** and the error messages in the **BASIC.ER** file. This function returns UNIX system errors for which there is no match in AOS/VS, DG/RDOS, or Business BASIC. If **SYS(40)** is negative, **SYS(41)** is –60, and **SYS(42)** is –276, **SYS(43)** returns the UNIX system error code of the last error that occurred. If **SYS(41)** is any value other than –60, **SYS(42)** and **SYS(43)** are undefined. If **SYS(41)** is –60 and **SYS(42)** is any value other than –276, **SYS(43)** is undefined.
44–49	Reserved.
50	On AOS/VS and UNIX systems, returns the function key header. This provides you with a more generic way of handling function keys. Thus, you can enter STMA 4,6,SYS(50) instead of using `STMA 4,6,30` to hardcode the Data General function key header.
51	On UNIX systems, returns the numeric keypad representation of the last function key pressed. The function keys 1-15 return the values 1-15; shift–function keys 1-15 return 101-115; control–function keys 1-15 return 201-215; and control–shift–function keys 1-15 return 301-315. (**Note**: Not all terminals support these function keys.)
52–256	Reserved.

How to Use It

Use **SYS** as a numeric expression wherever numeric expressions are allowed. If you don't understand the explanation given or the value received, look up each item or consult your system manager.

Examples

1. Print the time of day.

```
*PRINT SYS(27)     234804
```

2. Print the date as *mmddyy*.

```
*PRINT SYS(28)      51491
```

3. Print the date without delimiters.

```
*LIST
00010 DIM X$[8]
00020 LET X$=SYS(2),"/",SYS(1),"/",SYS(3)-1900
00030 PRINT X$

* RUN
5/14/91
```

4. Print the time of day with delimiters.

```
* LIST
00010 DIM X$[8]
00020 LET X$=SYS(14),":",SYS(13),":",SYS(12)
00030 PRINT X$

* RUN
23:52:31
```

TAB

Command

Sets the tab width.

AOS/VS	DG/RDOS	UNIX

Format

TAB=*width*

Arguments

width	An integer expression greater than 0 and less than or equal to the width of the page.

What It Does

TAB sets tab stops on a line, allowing *width* number of characters between them. Output from **PRINT** that is separated by commas will fall into the tabulated print zones. See **PRINT** for more information and for terminal control of output.

How to Use It

TAB is a keyboard mode command. To set print zones in a program statement, use **STMA 4,9**. Set your width equal to the number of characters allowed in a print zone.

Example

Set TAB width at 10 and print using that width.

```
* TAB=10
* PRINT 1,2,3 1         2         3
* TAB=14
* PRINT 1,2,3 1             2             3
*
```

TINPUT

Statement and Command

Performs an INPUT instruction in a specified amount of time.

AOS/VS	DG/RDOS	UNIX

Format

TINPUT *time*,[FILE(*channel*),][USING"",][@(*number*),...] $\left\{\begin{array}{l} prompt[,variable...][;] \\ variable[,variable...][;] \end{array}\right\}$

Arguments

time	A numeric expression for the number of tenths of a second to wait for input. In AOS/VS and UNIX systems, *time* is rounded up to the nearest second. If -1 is used for *time*, **TINPUT -1** selects the default timeout for the device, which is the user's console. 65535 is the maximum value for *time*.
channel	The channel number of a file if you are inputting data from a file; the file must be in character format.
@(number)	Cursor positioning and terminal control expressions (@ expressions) are described with **PRINT**.
prompt	A string literal in quotation marks that is output as a prompt for an input request from a terminal. This prompt replaces the question mark (?) prompt. On AOS/VS and DG/RDOS systems, you can enclose ASCII values in angle <> brackets within the string literal. **Note:** Not all terminals use the same ASCII values to perform the same functions.
variable	A numeric variable, string variable or array variable, depending on the values you want to input. You can also use subscripted string variables and array variables. Dimension string variables and arrays before using them. Any combination of the above is allowed.

What It Does

TINPUT performs an **INPUT** within a specified time. **TINPUT USING** performs an **INPUT USING**. If the variables in **TINPUT** are not filled when time is up, they remain unchanged.

NOTE: **TINPUT FILE** and **TINPUT FILE USING** are supported only in Business BASIC on AOS/VS and UNIX systems. **TINPUT FILE** uses the multitasking facilities of the host operating system. When a **TINPUT FILE** command executes, the timer task counts down in tenths of a second until it reaches 0 or until the associated **TINPUT** command completes execution. The precise timing of the delay depends on the system load. Thus, devices queried for timed input should not rely on an exact timeout interval.

TINPUT

continued

How to Use It

Specify time as a numeric expression for seconds in tenths of a second (e.g., 100 is 10 seconds). The function **SYS(22)** gives you the remaining time on the most recent **TINPUT**. When **SYS(22)** equals 0, the most recent **TINPUT** has timed out.

Example

Allow 10 seconds for data input. Print the time required for input and print the input.

```
00010  DIM A$[132]
00020  TINPUT 100,USING " ","INPUT DATA: ",A$
00030  IF SYS(22)=0 THEN GOTO 00060
00040  PRINT USING "('YOU TOOK',D4.1,' SECONDS TO INPUT:',S80)",
         100-SYS(22),A$
00050  STOP
00060  PRINT "TIMEOUT.  TRY AGAIN."
00070  GOTO 00020
```

```
*RUN
INPUT DATA: TIMEOUT.  TRY AGAIN.
INPUT DATA: THIS IS DATA
YOU TOOK 6.0 SECONDS TO INPUT: THIS IS DATA

STOP AT 00050
*
```

TRACE

Statement and Command

Displays the line numbers as statements are executed.

AOS/VS	UNIX

Format

AOS/VS Systems:

$$\text{TRACE}\begin{Bmatrix}\text{ON}\\ \text{OFF}\end{Bmatrix}$$

UNIX Systems:

$$\text{TRACE}\begin{Bmatrix}\text{ON [, ``}\textit{filename}\text{''] }\\ \text{OFF}\end{Bmatrix}$$

Arguments

ON	Enable the **TRACE** capability. **TRACE** displays the line numbers as statements are executed.
OFF	Disable the **TRACE** feature.
filename	File to which **TRACE** writes its output. (UNIX only)

What It Does

When **TRACE** is turned on, Business BASIC displays the line numbers of subsequent statements as the statements are executed. The line numbers are enclosed in brackets to separate them from program output. The line numbers are displayed until a **TRACE OFF** statement or command is supplied.

TRACE also displays the program name and the date and time when execution begins or continues. This message varies slightly, depending on whether you are running, swapping, chaining, or returning from a **SWAP**.

On UNIX systems, when the optional filename argument is used with the **ON** argument, output from **TRACE** is directed to that file. The output file should not exist; if it does a `File already exists` error will be returned. If you execute **TRACE ON** with a *filename* argument while tracing is already on and being directed to a file, the first output file will be closed. For example,

```
00500 TRACE ON,"TRACEOUT1"
...
00900 TRACE ON,"TRACEOUT2"
```

TRACE

continued

TRACEOUT1 will be closed when line 900 is executed before tracing output begins going to TRACEOUT2.

How to Use It

TRACE is available only in Business BASIC interpreters that include the debugging features. For more information on how to generate Business BASIC, see your Business BASIC user's guide.

TRACE ON or **TRACE OFF** can be executed as a statement or command. You can track parts of a program by setting **TRACE ON** and **TRACE OFF** throughout the program.

Examples

1. This example illustrates the use of the **TRACE** statement.

```
* LIST
00010 REM This is a test of the TRACE statement/command
00020 TRACE ON
00030 LET X=1
00040 PRINT "X =";X
00050 GOTO 00100
00060 STOP
00100 TRACE OFF

* RUN
[30] [40]X = 1
[50] [100]
```

2. This example illustrates the use of the **TRACE** command when a program is run from keyboard mode.

```
* ENTER "TRACE$EX2
* LIST

00010 REM * TRACE example 2
00030 LET X=0
00040 PRINT "X =";X
00045 LET X=X+1
00050 IF X<2 THEN GOTO 00040
* TRACE ON
* RUN "TRACE$EX2

[10] [30] [40]X = 0
[45] [50] [40]X = 1
[45] [50]
```

TRUN$

Function

Truncates a string.

AOS/VS	DG/RDOS	UNIX

Format

string-variable1=TRUN$(*string-variable2*[,1])

Arguments

string-variable1	The variable for the string or substring receiving the truncated string.
string-variable2	The variable for the string or substring to be truncated. It can be the same as *string-variable1*.
1	An optional 1 causes **TRUN$** to truncate at the last nonspace character in the string.

What It Does

TRUN$ truncates *string-variable2* at the first null, form feed, or end-of-line character and assigns the truncated string to *string-variable1*. The end-of-line character is Carriage Return in DG/RDOS systems and New Line in AOS/VS and UNIX systems.

TRUN$(*string-variable2*,1) truncates *string-variable2* when it reaches the last nonspace character, or when it reaches a null, form feed, or end-of-line character. It does this by going as far as the first form feed, null, or end-of-line character and moving backwards to the first nonspace character, at which point it truncates *string-variable2*.

How to Use It

Use **TRUN$** only in **LET** statements and commands to assign the truncated string to string-variable1. Use **TRUN$** without the 1 to reduce your string to its current length if it is padded to the end with nulls. Use **TRUN$** with the 1 to shorten a string that is padded to the end with nulls or spaces or use it to cut off trailing spaces.

Examples

1. This example shows an untruncated string.

```
* DIM A$(512)
* LET A$="ABC DEF GHI  ",FILL$(0)
* PRINT LEN(A$) 512
```

2. This example truncates the same string.

```
* LET A$=TRUN$(A$)
* PRINT LEN(A$) 13
```

3. This example truncates trailing blanks.

```
* LET A$=TRUN$(A$,1)
* PRINT LEN(A$) 11
```

UCALL

Statement and Command

Calls a subroutine from Business BASIC.

AOS/VS	DG/RDOS	UNIX

Format

UCALL *subroutine*[,*expression*,...]

Arguments

subroutine	A positive integer representing an assembly language subroutine number.
expression	As many as eight optional arguments, separated by commas, to be passed to the subroutine. They can be numeric or string variables or expressions. They cannot be arrays, only array elements. You cannot use statement line numbers.

What It Does

On AOS/VS and DG/RDOS systems, **UCALL** can call an assembly language subroutine if your system is generated to include user-written assembly language subroutines in **USERSUBS**. See the manual *Business BASIC System Manager's Guide* for more information.

On UNIX systems, UCALL can call C language subroutines. See *Using Business BASIC on DG/UX and INTERACTIVE UNIX Systems* for more information.

How to Use It

You must initialize all variable arguments before using them with a **UCALL**, or an error message occurs. You can pass only array elements—not arrays—as arguments.

See your system manager for a description of any **UCALL** statements that may be implemented on your system.

UCM$

Function

Uncrams a crammed string.

AOS/VS	DG/RDOS	UNIX

Format

LET *string-variable1*=UCM$(*string-variable2*)

Arguments

string-variable1	A string variable or substring to receive the uncrammed string. It can be the same as *string-variable2*.
string-variable2	The string variable or substring (already crammed) that you want to uncram.

What It Does

CRM$** shortens a string to save space, and **UCM$ is the only way you can "unshorten" it correctly.

Every two bytes of *string-variable2* are uncrammed to three bytes and put into *string-variable1*. If *string-variable1* is not large enough to hold the uncrammed string, you will lose some of the string. You can unpack only the following characters: A-Z (not lowercase), 0-9, and four special characters, set either by using **STMA 10** or the default characters—comma (,), decimal point (.), minus sign (–), and space. Any character in *string-variable2* that is not in this set is uncrammed as a space by default (see **STMA 10**).

How to Use It

The **UCM$** function is used only in **LET** statements and commands because you must assign the uncrammed string to *string-variable1*. If *string-variable1* and *string-variable2* are the same, then you will lose the crammed version of the string when you perform the **UCM$** on it. Use the **UCM$** function only on strings that have been crammed using the **CRM$** function. If you use **UCM$** on a string that has not been crammed, *string-variable1* will contain garbage.

UCM$

continued

Examples

1. A legal string crammed and uncrammed.

```
* LIST
00010 DIM X$(6),Y$(9)
00020 LET Y$= "ABCDEFGHI"
00030 LET X$=CRM$(Y$)
00040 LET Y$(1,6)=""
00050 LET Y$=UCM$(X$)
00060 PRINT Y$

*RUN
ABCDEFGHI
*
```

2. An illegal string crammed and uncrammed. The uncrammed string displays blanks in place of the illegal characters.

```
* LIST
00010 DIM A$[9],B$[6]
00020 LET A$="AB@CD$EF."
00030 LET B$=CRM$(A$)
00040 LET A$=UCM$(B$)
00050 PRINT A$

* RUN
AB CD EF.


*
```

UERM$

Function

Puts an error message into a string.

UNIX

Format

LET *string-variable*=UERM$(*number*)

Arguments

string-variable	A string variable, substring, or string array element (UNIX only) used to receive the error message; it must be dimensioned large enough to contain the error message.
number	The record number in the **BASIC.ER** file for the error message you want.

What It Does

UERM$ retrieves the error message specified by *number*. The number is typically returned by **SYS(43)**.

How to Use It

When you trap errors in your program (by using **ON ERR**), you can have the program print the appropriate error message. Also, you can generate your own unique error messages by adding messages to the end of the **BASIC.ER** file and then using *number* to specify which message to retrieve.

You can use **UERM$** only in **LET** statements or commands because you have to assign the error message to *string-variable*. The largest error message is 64 bytes long.

Some error messages generated by a UNIX system have no AOS/VS or DG/RDOS equivalent. Use **SYS(43)** and **UERM$** to retrieve these error messages.

See **SYS**, **AERM$**, and **ERM$** for more information about how to use error functions.

Example

This example uses **ERM$**, **AERM$**, and **UERM$** to retrieve error messages from **SYS(41)**, **SYS(42)**, and **SYS(43)**.

```
01000 REM * error handler
01010 IF SYS(41)=-60 THEN             :Same as SYS(7) and SYS(40)
01012   IF SYS(42)=-276 THEN
01014     LET ER$=UERM$(SYS(43))
01016   ELSE
01020     LET ER$=AERM$(SYS(42))      :Same as SYS(31)
01025   END IF
01030 ELSE
01040   LET ER$=ERM$(SYS(41))         :Same as SYS(7) and SYS(40)
01050 END IF
```

UNPACK

Statement and Command

Decodes a record string.

AOS/VS	DG/RDOS	UNIX

Format

```
        { format-string-variable   }
UNPACK  { "format-string-literal"  } ,string-variable,variable-list
        { statement-number         }
```

Arguments

format-string-variable	A string expression containing format information that describes the composition of *string-variable*.
format-string-literal	A string literal in quotation marks that contains format information describing the composition of *string-variable*.
statement-number	The statement number of an **RFORM** statement.
string-variable	A string variable that contains the values for *variable-list*.
variable-list	A list of string expressions, numeric expressions, or both, to be decoded from *string-variable*.

What It Does

The **UNPACK** decodes string and numeric information from a record variable containing binary information into separate variables. The record string must be filled through the last byte to be used with **UNPACK**. The variable list can be string variables, substrings, numeric variables, and arrays. The variables in the variable list need not be assigned prior to use of **UNPACK**; they will be assigned automatically as in **LET** and I/O statements. The format string is a string expression containing format characters that define how the expression list should be encoded into the record string. The statement number, which can replace the format string, must be that of an **RFORM** statement with an appropriate format string. See **RFORM** for the list of format characters and their interpretation.

continued

UNPACK

Example

Read and decode an employee record into fields: employee number, pay rate, overtime rate, deduction array, and tax array.

```
00320 RFORM JLL+8L*10@179L*6
:           |||| |   |   *6 4-byte signed integer elements into TAXES
:           |||| |   *pick next field starting with byte 179
:           |||| *10 4-byte signed integer elements into DEDNS
:           |||*skip 8 bytes of record string
:           ||*4-byte signed integer for OTRATE
:           |*4-byte signed integer for REGRAT
:           *2-byte signed integer for EMPNO
. . .
01100 LREAD FILE[5,RECNO],EMPREC$
01110 UNPACK 00320,EMPREC$,EMPNO,REGRAT,OTRATE,DEDNS[1],TAXES
:
:The array TAXES must have already been dimensioned
```

VAL

Function

Converts a string of digits to a numeric value.

AOS/VS	DG/RDOS	UNIX

Format

VAL(*string-expression,error-code*)

Arguments

string-expression	A string variable or string literal in quotation marks that contains the string of digits. The special characters plus (+), minus (-) and decimal point (.) are allowed.
error-code	A numeric variable that either can receive a -1 if an error occurred in the conversion or indicates the number of digits to the right of the decimal point. Zero is returned if the string is a plus sign (+), a minus sign (-), a null string, or a string literal containing all spaces.

What It Does

VAL converts an alphanumeric string of digits to a numeric value.

For simple conversion, the string should contain only a signed integer or real number (leading spaces and trailing spaces are ignored when processing the string). If the string contains a negative number, the value returned will be negative. If the string contains a decimal point, the value returned will be an integer and the number of digits to the right of the decimal point will be returned in *error-code*. When the string contains an invalid character (other than a minus sign, plus sign, or decimal point), *error-code* returns -1. For example, if the string-literal is "5.32X", the number returned is 532 and *error-code* is -1. Note that *error-code* is not 2 as you would expect. If at least one invalid character is in the string, *error-code* does not define the decimal position.

VAL returns zero and *error-code* returns zero when the string is null (empty) or contains only spaces. If the string is filled with nulls, the value -1 is returned. For example, if the string is **FILL$(32)**, the error code is 0. If the string is **FILL$(0)**, the error code is -1. If the string equals "", the error code is 0.

How to Use It

Numbers expressed as string literals or string variables cannot be used in arithmetic expressions. They must be converted from their present ASCII code to a numeric value. Use **VAL** to convert a string of ASCII digits to a numeric variable that can be used in calculations or arithmetic expressions.

Example

1. Convert the string IN$ to a number and print the number with a decimal point inserted in the correct position. This example uses valid and invalid values for the string literal.

```
00010 DIM STRLIT$(10)
00020 ER = 0
00030 INPUT STRLIT$
00040 NUMBA = VAL(STRLIT$,ER)
00050 PRINT STRLIT$,NUMBA,ER
* RUN
? 36.521
36.521  36521  3
* RUN
? 42.729A
42.729A  42729  -1
```

2. Convert the string IN$ to a number. Print the number with a decimal point inserted in the correct position. This example uses valid and invalid values for the string literal.

```
*LIST
00010 LET M=0
00020 DIM A$[10],IN$[10]
00030 INPUT IN$
00040 LET OUT=VAL(IN$,M)
00050 IF M=-1 THEN GOTO 00090
00060 LET A$="D10.",M
00070 PRINT USING A$,OUT
00080 STOP
00090 PRINT "ERROR"
00100 GOTO 00030


* RUN
? 36.521
   36.521


STOP AT 00080
* RUN
? 42.729A
ERROR
? 42.729
   42.729


STOP AT 00080
*
```

VALUE

Statement and Command

Converts a string to a number.

AOS/VS	DG/RDOS	UNIX

Format

VALUE *value,string,position,scale*

Arguments

value	A numeric variable that receives the signed numeric value of *string*.
string	A string expression composed of the ASCII characters +-.0123456789 and space.
position	An optional numeric variable that, when specified, receives an error termination code. *position* is: • Positive if *string* is terminated with either a space or comma • Negative when *string* is terminated by any character other than a space or comma • 0 if the conversion reaches the end of the string without error
scale	An optional numeric variable that, when specified, receives the implied decimal point location of *string*. If scale is a negative value, no decimal point was found in *string*.

What it Does

VALUE converts an alphanumeric string of digits to a numeric value.

For simple conversion, the string should contain only a signed integer or real number (leading spaces and trailing spaces are ignored when processing the string). If the string contains a negative number, the value returned is negative. If the string contains a decimal point (.), the value returned is an integer and the number of digits to the right of the decimal point is returned in the *scale* argument, if it is supplied. If no decimal point is found, *scale* returns a negative number. When the string contains an invalid character (other than a minus sign (–), plus sign (+) or decimal point), *position* returns with a negative position. For example, if the string literal is "5.32X", the number is 532 and *position* is –5. *position* does not return with 0, which means that the conversion reached the end of the string without error.

How to Use It

Numbers expressed as string literals or string variables cannot be used in calculations or arithmetic expressions. They must be converted from their present ASCII codes to numeric values. Use VALUE to make the conversion. You must assign a starting value of zero to *value*, *position*, and *scale*, and dimension *string* before these variables are used in a VALUE statement.

Since **VALUE** on UNIX systems can return quad precision numbers, *string* can contain numbers up through the maximum size for quad precision. On AOS/VS and DG/RDOS systems, the number in *string* cannot exceed the maximum size allowed for a double precision number.

Example

Convert an input string, X$, to a number. Print the converted number, the decimal location (scale), and the error termination code.

```
00010 DIM X$[20]
00020 LET Y=0
00030 LET Z=0
00040 LET X=0
00050 PRINT "INPUT STRING VALUE, PRESS ESC TO EXIT"
00060 INPUT USING "",X$;
00070 VALUE X,X$,Z,Y
00080 PRINT ,"X=";X,"Y=";Y,"Z=";Z
00090 GOTO 00060
```

```
* RUN
INPUT STRING VALUE, PRESS ESC TO EXIT

? 123         X= 123       Y=-32765   Z= 0   Unsigned integer.
? 123.45      X= 12345     Y= 2       Z= 0   Unsigned decimal.
? 123.45,678  X= 12345     Y= 2       Z= 7   Signed, stop on comma.
? 987.        X= 987       Y= 0       Z= 0   No decimal places.
? .0012       X= 12        Y= 4       Z= 0   Decimal, leading zeroes.
? -235-75     X=-235       Y=-32765   Z=-5   More than one sign.
? 123-ABC     X=-123       Y=-32765   Z=-5   Error terminator.
? 123.4-56    X=-123456    Y= 3       Z= 0   Imbedded sign.
? +123.456    X= 123456    Y= 3       Z= 0   Leading sign.
?             X= 0         Y=-32768   Z= 0   Null input string.
? ABCDEF      X= 0         Y=-32768   Z=-1   No valid digits.
?
IKEY AT 00060
*
```

VAR DISPLAY

Command

Displays the variables used in a program.

UNIX

Format

VAR DISPLAY [*program-name*]

Arguments

program-name	The name of the program for which you want the list of variables. This argument is optional.

What It Does

VAR DISPLAY supplies an alphabetical list of all the variables in a SAVE file or in a program in working storage.

If you are running Business BASIC in DG mode (i.e., you included the **-D** option when you executed Business BASIC), then the output of **VAR DISPLAY** is similar to the output of the **VAR** utility. If you are running Business BASIC in non-DG mode, then the output of **VAR DISPLAY** appears in a window on the right side of your screen. The program name and the number of variables are displayed, and variables are listed in alphabetical order. The dimensions of string variables are shown.

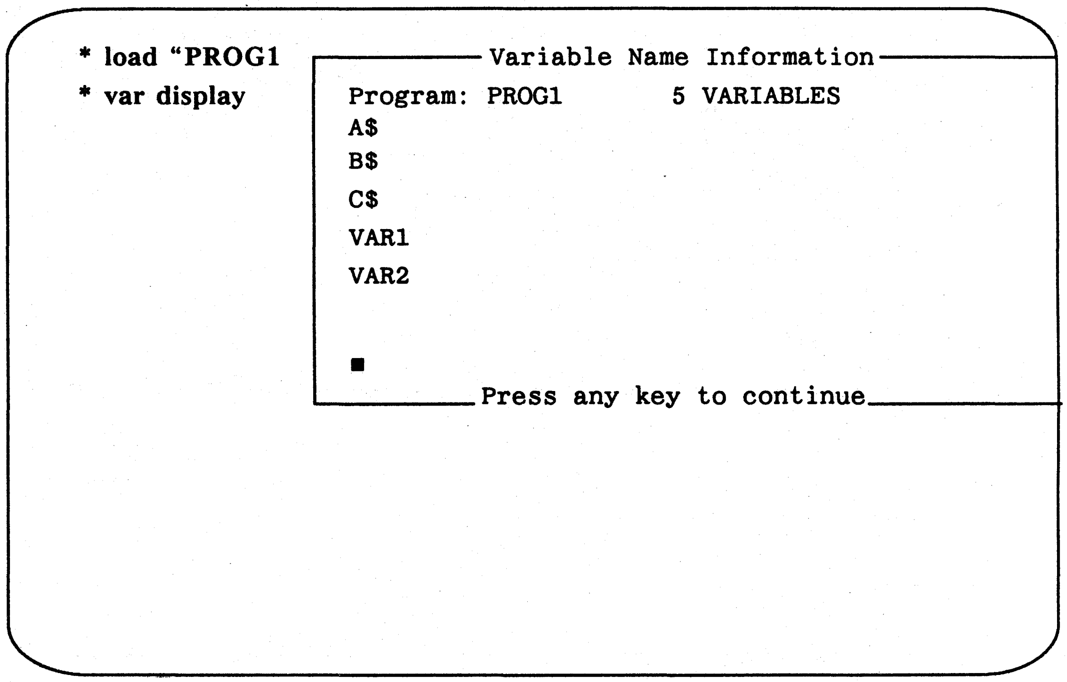
How to Use It

Use this command in keyboard mode. Press **q** when you are ready to exit the display and return to the Business BASIC prompt.

If there are more variables than can be displayed on one screen, you are prompted with the word MORE in the bottom border of the window. At this point, you can press **q** to quit the display, New Line to scroll down one line in the display, or the space bar to scroll down one screen.

Example

In this example, **PROG1** has been loaded into working storage but has not been executed. Since Business BASIC is executing in non–DG mode, the output of **VAR DISPLAY** appears in a window on the right side of the screen.



```
* load "PROG1      ┌──────────Variable Name Information──────────┐
* var display      │ Program: PROG1        5 VARIABLES            │
                   │ A$                                           │
                   │ B$                                           │
                   │ C$                                           │
                   │ VAR1                                         │
                   │ VAR2                                         │
                   │                                              │
                   │ ■                                            │
                   └──────────Press any key to continue──────────┘
```

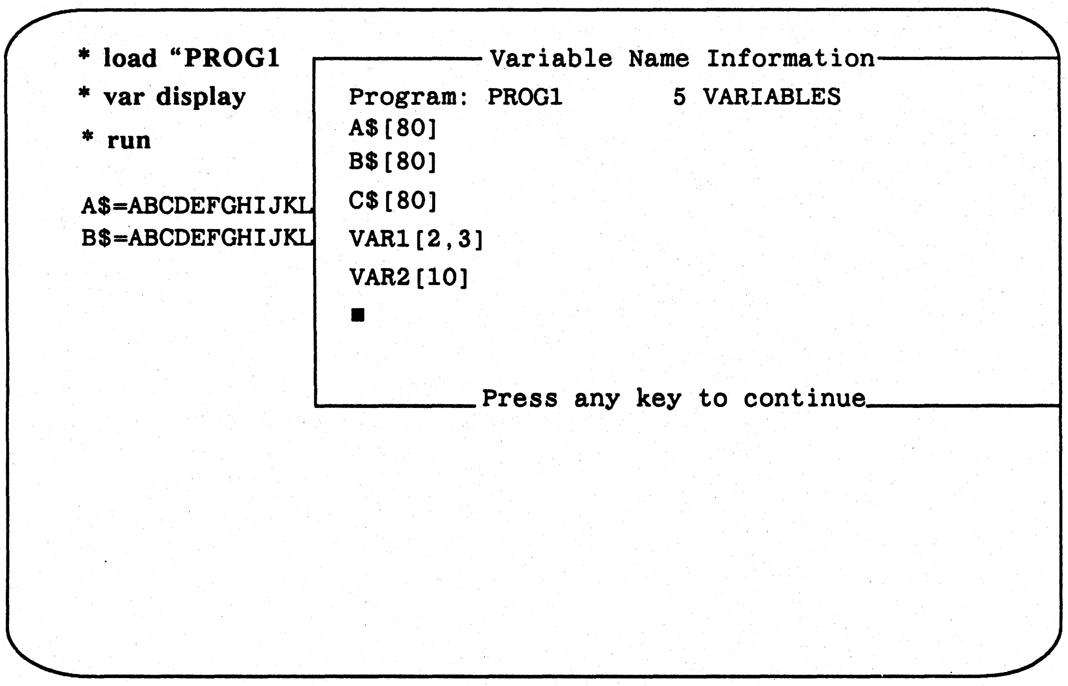
At this point, you can press a key to return to the * prompt and run **PROG1**.

```
* load "PROG1
* var display
* run

A$=ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
B$=ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
*
```

VAR DISPLAY *continued*

Now, when you issue the **VAR DISPLAY** command after **PROG1** has run, the following window is displayed at the right of the screen.



```
* load "PROG1       ─────────Variable Name Information──────────
* var display       │ Program: PROG1        5 VARIABLES
* run               │ A$[80]
                    │ B$[80]
A$=ABCDEFGHIJKL     │ C$[80]
B$=ABCDEFGHIJKL     │ VAR1[2,3]
                    │ VAR2[10]
                    │ ■
                    │
                    └──────────Press any key to continue──────────
```

When you press any key, the **VAR DISPLAY** window at the right of the screen disappears and the * prompt returns.

VAR RENAME

Command

Changes the names of program variables.

UNIX

Format

VAR RENAME *old-variable-name,new-variable-name*

Arguments

old-variable-name	The variable name that you want to change to *new-variable-name*.
new-variable-name	The new name you want to assign to *old-variable-name*.

What It Does

VAR RENAME changes the name of a program variable. It is similar to the **RNAM** utility.

How to Use It

Use this command in keyboard mode.

Since renaming of the variable is done in the current program's space, the old variable name is not immediately removed. Therefore, if you issue a **VAR DISPLAY** or **PROGRAM DISPLAY** command after a **VAR RENAME** but before listing the program and re-entering it, the old variable name and the new variable name are both in the variable list displayed by **VAR DISPLAY**. To remove the old variable name from the **VAR DISPLAY** and **PROGRAM DISPLAY** output, list the program to a file, execute a **NEW** statement, and then re-enter the program from the file.

Example

In this example, you enter an ASCII source file, rename a variable, list the program to your screen to verify the change, and then list the program to a file to save the change.

```
* enter "prog1.sf
* list
00010 DIM A$[26]
00020 LET A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00030 PRINT A$
00040 END

* var rename A$,ALPHA$
* list
00010 DIM ALPHA$[26]
00020 LET ALPHA$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00030 PRINT ALPHA$
00040 END

* list "new_prog1.sf
```

WRITE FILE

Statement and Command

Writes length-sensitive output.

AOS/VS	DG/RDOS	UNIX

Format

WRITE FILE (*channel*[,*byte-position*]),*expression*[,*expression*...]

Arguments

channel	A numeric expression for the channel number of a file opened for random or sequential output. Always enclose the channel number in parentheses or square brackets.
byte-position	A numeric expression or variable for the relative byte within a file to which you want to position the file pointer.
expression	Any valid numeric or string expression.

What It Does

WRITE FILE outputs binary data to any type of file or to your terminal. You determine the type of access when you open the file, but **WRITE FILE** works for sequential and random (direct) access. Data can later be retrieved from the file using **READ FILE.**

The size you specify for *byte-position* determines the number of bytes output to the file. For each variable in a **WRITE FILE** statement, a specific output will occur; so if you have five variables in a **WRITE FILE,** five separate outputs will occur in sequential order (i.e., each output is put after the preceding one in the file). The file pointer is moved to the end of the last output, so the next output will start where the last one stopped.

How to Use It

You can write to a file originally opened for sequential access and your program will write sequentially on the file. You can also write to a file opened for random access starting at some byte position in the file, writing sequentially from that point. If your file is a subfile and/or in linked-record format, then you have "records" that are fixed in length. A record is a fixed-length area in a linked available record file, subfile, or index file. When records have a fixed length, you can easily determine where to position the pointer: multiply the total number of bytes per record by the number of the record you want to write.

You can write part of a record or an entire record using one or more variables. If you are rewriting a record, you should write an entire record so that you rewrite every byte in the record. Since each variable in a **WRITE FILE** statement is a specific output, you can avoid an interrupt that occurs between outputs of a single **WRITE FILE** by using one large variable in a **WRITE FILE** (a single output cannot be interrupted).

continued

WRITE FILE

The *byte-position* argument enables you to use **POSITION FILE** and **WRITE FILE** in a single program line. This argument is ignored in statements that perform terminal I/O via channel 16. Specifying the argument when the file has been opened in a mode that does not allow **POSITION FILE** causes a runtime error.

WRITE and **WRITE FILE(16)** output to your terminal and may be used to output characters which have special meanings to the **PRINT** statement.

Examples

1. Sequentially fill a file with a string and the array information associated with the string.

```
*LIST
00010 OPEN FILE[0,1],"FBI"
00020 DIM S$[15],A[1,2]
00030 INPUT "SUSPECT'S NAME: ",S$
00040 IF S$="END" THEN GOTO 00150
00060 WRITE FILE[0],S$
00070 FOR I=0 TO 1
00080   FOR J=0 TO 2
00090     INPUT A[I,J];
00100     WRITE FILE[0],A[I,J]
00110   NEXT J
00120 NEXT I
00130 PRINT
00140 GOTO 00030
00150 CLOSE
00160 END
```

2. This example writes a six-byte value in both **WRITE FILE** statements (for triple precision systems only).

```
00010  OPEN FILE[0,0], "TRIP_DATA"
00020  LET X#=1234567890123
00030  WRITE FILE[0],X#
00040  WRITE FILE[0],X#+1
. . .
```

WRITE FILE

3. Offer a record for rewriting, display the record, then rewrite the record.

```
00010 DIM RECORD$(48)  :48 bytes + 2 bytes(status)=50-byte records
00020 OPEN FILE[2,0],"DATA"
00030 INPUT "WHAT RECORD DO YOU WANT TO REWRITE? ",R
00040 POSITION FILE(2,50*R)     :50 is record size, R is record
                                :you want.
00050 READ FILE[2],STAT%,RECORD$      :Read 2 bytes into STAT%,
                                      :rest into RECORD$,then
00060 PRINT RECORD$                   :look at RECORD$.
00070 INPUT "REWRITE IT? YES(0),NO(1): ",A
00080 IF A THEN GOTO 00400            :If "no", go to 400.
00090 INPUT "TYPE NEW RECORD: ",RECORD$
00100 POSITION FILE[2,50*R]     :Go back to beginning of record R.
00110 WRITE FILE[2],STAT%,RECORD$     :Write new record, over-
                                      :writing old one.
```

4. Read a record, display the record, and then rewrite the record. Do the position in the **READ FILE** and **WRITE FILE** statements.

```
00010 DIM RECORD$(48)                 :48 bytes + 2 bytes (status)
                                      = 50 bytes
00020 OPEN FILE(2,0),"DATA"
00030 INPUT "WHAT RECORD DO YOU WANT TO REWRITE: ",R
00040 READ FILE[2,50*R],STAT%,RECORD$ :50 is record size, R is
                                      record you want. Read 2
                                      bytes into STAT% and rest
                                      into RECORD$.
00050 PRINT RECORD$
00060 INPUT "REWRITE IT? YES(0), NO(1): ",A
00070 IF A THEN GOTO 00400            :If "no", go to 400.
00080 INPUT "TYPE NEW RECORD: ",RECORD$
00090 WRITE FILE[2,50*R],STAT%,RECORD$         :Position to
                                :beginning of record R. Write new
                                :record, overwriting old one.
. . .
```

XOR

Function

Performs an exclusive logical OR of two expressions.

AOS/VS	UNIX

Format

XOR(*expression1,expression2*)

Arguments

expression1 and *expression2*	The numeric expressions or variables you want compared.

What It Does

The exclusive **XOR** function is used to set bits in a binary expression. The binary representations of the two arguments are compared bit by bit. If a bit is set to 0 in both expressions, that bit is set to 0 in the result. If a bit is set to 1 in either expression, that bit is set to 1 in the result. If a bit is set to 1 in both expressions, that bit is set to 0 in the result.

How to Use It

Use the **XOR** function to set bit flags or to combine two sets of flags into a single expression. You can use the **XOR** function in any numeric expression.

Figure 1–10 shows the result when the bit values of two expressions are compared using **XOR**.

XOR (192, 127)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Power of 2
0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	$expr_1$ = 192
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	$expr_2$ = 127
0	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	RESULT = 191

Logical XOR of Two Numbers

Figure 1–10 Logical XOR of Two Numbers

XOR

continued

Example

The **XOR** function displays the value obtained by clearing the rightmost seven bits of X.

```
00010 INPUT "Initial value of X: ",X
00020 PRINT "Value of XOR(X,127): ",XOR(X,127)

* RUN
Initial value of X: 192
Value of XOR(X,127): 191
```

End of Chapter

Chapter 2
Statements Related to INFOS® II Files

The statements described in this chapter are used in interfaces between the INFOS® II file management system and Business BASIC. INFOS II software is sold as a separate product by Data General Corporation for AOS and AOS/VS operating systems.

The INFOS II system is an indexing system that organizes and manages records of information. If you have INFOS II software installed on your system, you can use **DB** statements to access it from within Business BASIC.

You cannot create an INFOS file with AOS/VS Business BASIC. To create your index and database, use the AOS/VS INFOS utility **ICREATE**. When this file structure is created, Business BASIC lets you manipulate it with all the functions described in *AOS/VS INFOS® II System User's Manual.*

NOTE: These statements are used to access and modify INFOS files only. Any attempt to modify channel strings with Business BASIC **LET** statements could cause inadvertent modification of an index or database.

Business BASIC imposes a limit of 64 open INFOS II channels for AOS/VS and 16 open INFOS II channels for AOS. If you exceed this limit, Business BASIC displays the error message `NO MORE CHANNELS AVAILABLE`.

Business BASIC's INFOS II interface statements are similar to those of other Data General products (e.g., COBOL, FORTRAN, and PL/I). The Business BASIC INFOS II interface consists of a set of statements that behave like regular Business BASIC statements; however, their formats are somewhat different. For easy identification, the name of each of these statements begins with DB.

The following table gives a brief description of each INFOS II statement:

Statement	Description
DBCLOSE	Closes an open INFOS II file.
DBDELETE	Deletes a key and/or record.
DBGET	Gets values returned by INFOS II.
DBOPEN INFOS	Opens an INFOS II index.
DBREAD	Reads from an INFOS II file.
DBREINS	Reinstates a logically deleted record.
DBRELEASE	Releases locks and/or position.

Statement	Description
DBRETRIEVE HIGHKEY	Retrieves the high key.
DBRETRIEVE KEY	Retrieves the key.
DBRETRIEVE SIDEF	Retrieves a subindex definition.
DBRETRIEVE STATUS	Returns the INFOS II status.
DBREWRITE	Rewrites an INFOS II database record.
DBSET	Permanently sets an INFOS II parameter.
DBSUBINDEX DEFINE	Defines a subindex.
DBSUBINDEX DELETE	Deletes a subindex.
DBSUBINDEX LINK	Links a subindex.
DBSUBINDEX LINKINIT	Initializes a link subindex string.
DBSUBINDEX LINKSET	Sets parameters in link subindex string.
DBWRITE	Writes to an INFOS II file.

Argument Pairs

Each INFOS II statement has a set of arguments called an argument pair. Argument pairs, comparable to switches, alter the statements. Some argument pairs are required for particular statements, while others are optional or not applicable. Some examples are:

Argument Pair	Definition
ACCESS=REL	Relative access is requested.
DUPKEY=NO	The specified key is not a duplicate.
MOTION=BACK	The direction of relative motion is backward.

Where possible, the statements take default values for argument pairs. You can redefine the default values by using the **DBSET** statement.

Creating an INFOS II File

When you create an INFOS II file, two logical structures are defined: an index and a database. INFOS II automatically links these two structures to form a single ISAM or DBAM file and, using AOS or AOS/VS, allocates space and physically constructs the file.

To create an INFOS II file, use the AOS/VS INFOS II utility **ICREATE**. You must execute **ICREATE** from the CLI. The format is:

ICREATE *filename*

where *filename* is the name of the index file. If you do not specify the filename, INFOS II prompts you for it.

Below is a sample **ICREATE** dialog. See the *AOS/VS INFOS® II System User's Manual* for a complete discussion of **ICREATE**.

```
****** INFOS FILE CREATION      6/16/82 9:54:15 ******

NAME OF FILE TO BE CREATED: ACCOUNTS
ACCESS METHOD (I=ISAM, D=DBAM) [D]  I

****** DEFINE INDEX FILE ******

PAGE SIZE (BYTES) [2048]: 2048
ROOT MODE SIZE [2042]:  2042
MAXIMUM KEY LENGTH [255]:  10
ALLOW DUPLICATE KEYS IN THIS INDEX? (Y OR [N]):  N
ENABLE SPACE MANAGEMENT? (Y OR [N]):  N
ENABLE KEY COMPRESSION? (Y OR [N]):  N
OPTIMIZE RECORD DISTRIBUTION? (Y OR [N]):  N

****** DEFINE INDEX VOLUME(S) ******

NUMBER OF VOLUMES TO DEFINE [1]:  1
VOLUME 1 NAME [VOL01]:  VOL01
SPECIFY MAXIMUM SIZE? (Y OR [N]):  N
SPECIFY FILE ELEMENT SIZE? (Y OR [N]):  N

****** DEFINE DATABASE FILE ******

DATABASE FILE NAME [ACCOUNTS.DB]:  ACCOUNTS.DB
PAGE SIZE (BYTES) [2048]:  2048
ENABLE SPACE MANAGEMENT? (Y OR [N]):  N
ENABLE DATA RECORD COMPRESSION? (Y OR [N]):  N
OPTIMIZE RECORD DISTRIBUTION? (Y OR [N]):  N

****** DEFINE DATABASE VOLUME(S) ******

NUMBER OF VOLUMES TO DEFINE [1]:  1
VOLUME 1 NAME [VOL01]:  VOL01
SPECIFY MAXIMUM SIZE? (Y OR [N]):  N
SPECIFY FILE ELEMENT SIZE? (Y OR [N]):  N
```

Accessing INFOS II Files

To access an INFOS II index file, you must open it using the **DBOPEN** statement from keyboard mode. Type

*** DBOPEN INFOS** *filename,string*

where *filename* is the name of the INFOS II file (named with **ICREATE**), and *string* is the name of the channel on which the INFOS II file is opened.

The string is called the channel string. It is similar to the channel number used in Business BASIC statements (such as **PRINT FILE, READ FILE,** and **WRITE FILE**). **DBOPEN** initializes the channel string, which you use throughout the program to refer to the INFOS II file. The following example illustrates how to use **DBOPEN.**

To open an INFOS II file called **ACCOUNTS** on channel string MASTER$, enter

*** DBOPEN INFOS "ACCOUNTS", MASTER$**

Now you can use any INFOS II statement to manipulate the file.

Two common statements are **DBWRITE** and **DBREAD**. **DBWRITE** writes records and/or keys to an INFOS II file; **DBREAD** reads records and/or keys from an INFOS II file.

To write a record to **ACCOUNTS**, enter

*** DBWRITE MASTER$**

where MASTER$ is the channel string referring to the INFOS II file **ACCOUNTS**. **DBWRITE** rewrites a record to an INFOS II database.

To read a record from **ACCOUNTS**, enter

*** DBREAD MASTER$**

where MASTER$ is the channel string referring to the INFOS II file **ACCOUNTS**.

To read the previous record in **ACCOUNTS** and store it in a string variable called MDATA$, enter

*** DBREAD MASTER$, ACCESS=REL, MOTION=BACK, REC=MDATA$**

ACCESS=REL indicates relative access (versus keyed). **MOTION=BACK** indicates that the direction of relative motion is backward because the system reads the previous record. **MOTION** cannot be used with the argument pair **ACCESS=KEY**. **REC=MDATA$** indicates that MDATA$ is the record string because it receives the data from the record that the system reads.

Other common INFOS II operations include deleting keys (**DBDELETE**) and reinstating logically deleted records (**DBREINS**). You may also want to use more advanced INFOS II features. These include defining subindexes (**DBSUBINDEX DEFINE**), retrieving subindex definitions (**DBRETRIEVE SIDEF**), and linking subindexes (**DBSUBINDEX LINK**). Two INFOS II statements, **DBSET** and **DBGET**, are not directly related to any particular INFOS II function. **DBSET** lets you set default values for subsequent program statements; **DBGET** retrieves values returned by previous statements.

The Business BASIC statements used in conjunction with the INFOS II system are presented in alphabetical order in the following pages. For more information about the INFOS II system, see the *AOS/VS INFOS® II System User's Manual.*

DBCLOSE

Statement

Closes an open INFOS II file.

AOS/VS

Format

DBCLOSE *channel-string*

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).

What it Does

DBCLOSE closes an INFOS II file that has been opened with **DBOPEN**. INFOS II automatically writes to the file all modified index and database information currently in the system buffer and unlocks any locked partial and complete data records. The **CLOSE** statement without arguments and the **NEW** statement also close INFOS II files.

Example

```
* 100  DBCLOSE X$
```

DBDELETE *Statement*

Deletes a key and/or record.

AOS/VS

Format

DBDELETE *channel-string*[*argument pair,...*]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant argument pair that can be used with the **DBSET** statement. Argument pairs used with **DBDELETE** affect the program for the **DBDELETE** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBDELETE deletes a key, partial record, or record.

To specify the scope of a deletion, use the **GLOBAL=YES** and **LOCAL=YES** argument pairs. **GLOBAL** argument pairs affect the database (see **DBSET**). **LOCAL** argument pairs affect the partial record on subsequent **DELETE** operations. To specify the type of deletion (logical or physical), use the **DELETE=LOG** or **DELETE=PHYS** (the default) argument pair.

When **DBDELETE** is used with the **DELETE=LOG** pair, then either or both of the **GLOBAL=YES** and **LOCAL=YES** pairs should also be specified. These pairs can be included with either the **DBDELETE** statement or the **DBSET** statement. If neither **GLOBAL=YES** nor **LOCAL=YES** is specified, no logical deletion occurs.

Example

In this example, the logically deleted key is the one contained in KEYID$.

```
*0100  DBDELETE X$,DELETE=LOG,GLOBAL=YES,KEY=KEYID$
```

DBGET

Statement

Gets values returned by INFOS II.

AOS/VS

Format

DBGET *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	You can use any of the following argument pairs:

FEEDBACK=*numeric-variable*
Stores the feedback value in *numeric-variable*.

KEYNO=*numeric-variable*
Used with the **OCCUR=***numeric-variable* argument pair; it specifies the key level to which the **OCCUR=***numeric-variable* pair applies.

If you do not use this pair, INFOS II returns the occurrence number for the first key (**KEYNO=1**).

Note: This pair does not return a value.

MAXKEYLEN=*numeric-variable*
Returns the maximum key length allowed for the last subindex for which a **DBRETRIEVE SIDEF** was performed.

OCCUR=*numeric-variable*
numeric-variable receives the occurrence number of the duplicate key.

PRECLEN=*numeric-variable*
Returns the partial record length for the last subindex for which a **DBRETRIEVE SIDEF** statement was performed.

RNSIZE=*numeric-variable*
Returns the next node size for the last subindex in which a **DBRETRIEVE SIDEF** was performed.

SIFLAG=*numeric-variable*
Returns the subindex definition flags for the last subindex for which a **DBRETRIEVE SIDEF** was performed.

SILEVEL=*numeric-variable*
Returns the subindex level where the last INFOS II access occurred.

STATUS=*numeric-variable*
Returns the INFOS II status.

DBGET

continued

What It Does

DBGET gets values returned by previous INFOS II statements. **DBGET** is not directly related to any INFOS II function; rather, it lets you access information stored in the channel string.

If an error occurred during execution of the previous INFOS II statement, you can still use **DBGET** to get information in certain cases. For example, if you execute a **DBRETRIEVE STATUS** on a record that has no database record, **DBGET** returns a **STATUS** value even though the INFOS II error DATABASE RECORD NOT PRESENT occurs.

The returned **STATUS** value consists of an aggregate of bits. If you use the **AND** function to compare the bit patterns of **STATUS** with one of the following masks and the result is not zero, the condition indicated by the particular mask is present.

Mask	If AND(mask, STATUS) < > 0 then
1	The data base record is logically deleted.
2	The partial record is logically deleted.
4	The last (or only) key is a duplicate.
32	The access key is linked to a subindex.
64	The record is longer than the buffer.

The returned **SIFLAG** value consists of an aggregate of bits. If you use the **AND** function to compare the bit pattern of one of the following masks with the bit pattern of the value returned from **DBGET SIFLAG** and the result is not zero, the error condition indicated by the particular mask is present.

Mask	If AND(mask, STATUS) < > 0 then
2048	Duplicate keys are not allowed.
16384	Subindexes are not allowed.

Example

This statement returns the occurrence number of a duplicate key to OKKUR and the status value to STAT.

```
7040 DBGET F$,OCCUR=OKKUR,STATUS=STAT
```

DBOPEN INFOS

Statement

Opens an INFOS II file.

AOS/VS

Format

DBOPEN INFOS *filename,channel-string*[*,argument pair,...*]

Arguments

filename	The name of the INFOS II index file you are opening. The filename can be a string, a substring, or a literal filename in quotes.
channel-string	A string variable used by the system to hold channel and other information concerning the INFOS II file. Use *channel-string* in all statements referring to the INFOS II file.
argument pair	You can use any of the following argument pairs:

ACCESS=EXCL
Opens the index and database exclusively.

ACCESS=EXCLDB
Opens the index in shared mode and the database exclusively.

ACCESS=EXCLI
Opens the index exclusively and the database in shared mode.

ACCESS=RDON
Opens the index and database in read-only mode.

ACCESS=SHARED
Opens the index and database in shared mode.

MAXKEYS=*numeric-variable*
Sets the maximum number of levels of keyed access to be used at one time (for this channel) to *numeric-variable*. *numeric-variable* can be a numeric variable, a subscripted numeric variable, or an unsigned numeric constant.

NLOCKS=*numeric-variable (AOS only)*
Sets the maximum number of simultaneous locks allowed to the number specified in *numeric-variable*. This argument can be a numeric variable, a subscripted numeric variable, or an unsigned numeric constant. (This argument pair is accepted but ignored on AOS/VS.)

To lock records, the NLOCKS=*numeric-variable* argument pair must be set to a value greater than zero. Failure to do this returns a LOCK LIMIT EXCEEDED error message. When **NLOCKS=0**, the record is still created but it has no locking capability.

DBOPEN INFOS

continued

What It Does

DBOPEN INFOS opens an existing INFOS II index file using a channel string. This is similar to the way **OPEN** opens a Business BASIC file using a channel number. When you open a Business BASIC file, you provide the channel numbers. When you use **DBOPEN** to open an INFOS file, you provide a dimensioned string that you will use in subsequent INFOS II statements to refer to the INFOS II index file. **DBOPEN INFOS** initializes *channel-string* to the following **DBSET** defaults:

```
ACCESS=REL, DELETE=PHYS, GLOBAL=NO, INVERT=NO, LOCAL=NO, LOCK=NO,
LRECLEN=YES, MOTION=FWD, RELLOCK=NO, RELPOS=NO, SDBASE=NO,
SETPOS=YES, SPREC=YES, UNLOCK=NO
```

How to Use It

To access an INFOS II file, use **DBOPEN INFOS.** Under AOS you can use this statement to open 16 files simultaneously; the limit on an AOS/VS system is 64. Before using the statement, you must:

1. Create the INFOS II index file using the INFOS II utility **ICREATE.**

   ICREATE creates two files: *filename* (index file) and *filename*.db (database). You can then use the statement **DBOPEN INFOS** *filename* to open *filename*. (For a complete discussion of ICREATE, see *INFOS® II System User's Manual.*)

2. Dimension the channel string.

   To dimension the channel string under AOS, use the formula:

   ```
   (16 * MAXKEYS) + 56
   ```

   To dimension the channel string under AOS/VS, use the formula:

   ```
   (44 * MAXKEYS) + 136
   ```

   MAXKEYS is the value of the **MAXKEYS** argument pair (default=1).

   A channel string is comparable to the channel number in other Business BASIC statements (such as **PRINT FILE, READ FILE,** and **WRITE FILE**). It provides a unique way to reference the INFOS II file. *channel-string* contains information Business BASIC needs to access INFOS II. Once you have opened the INFOS II file, use the channel string only in INFOS II statements. Do not alter *channel-string* until you have closed the INFOS II file. The Business BASIC INFOS II statements allow you to access a wide range of INFOS II features, so you should not need to change the contents of the string.

Examples

1. This statement opens the INFOS II index file **FILE** using CASES$ as the channel string.

   *** 100 DBOPEN INFOS FILE$,CASES$**

continued **DBOPEN INFOS**

2. This statement opens the INFOS II file ACCOUNTS using MASTER$ as the channel string. Both the index and database are opened in shared mode.

   *** 200 DBOPEN INFOS "ACCOUNTS",MASTER$,NLOCKS=1,MAXKEYS=1, ERR=900, ACCESS=SHARED**

DBREAD

Statement

Reads data from an INFOS II file.

AOS/VS

Format

DBREAD *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any argument pair that can be used with the **DBSET** statement. Argument pairs used with **DBREAD** are in effect for the **DBREAD** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBREAD reads data from an INFOS II file.

Examples

1. This statement reads records in ascending order from the INFOS II file opened on MASTER$.

   ```
   * 100 DBREAD MASTER$,MOTION=FWD
   ```

2. If you are above a subindex, you can use a **DBREAD** statement with the suppress database option (SDBASE=YES) to position to the subindex.

   ```
   * 200 DBREAD X$,ACCESS=REL,MOTION=DOWNFWD,SDBASE=YES
   ```

DBREINS

Statement

Reinstates a logically deleted record.

AOS/VS

Format

DBREINS *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any argument pairs that can be used with the **DBSET** statement. Argument pairs used with **DBREINS** affect the program for the **DBREINS** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBREINS reverses the effect of a logical deletion.

Example

```
* 100 DBREINS X$
```

DBRELEASE

Statement

Releases locks and/or current position.

AOS/VS

Format

DBRELEASE *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant argument pairs that can be used with the **DBSET** statement. Argument pairs used with **DBRELEASE** affect the program for the **DBRELEASE** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBRELEASE releases any locks and/or the current position.

To specify the functions to be performed, use the **RELLOCK** and **RELPOS** argument pairs (see **DBSET**).

Example

This statement releases any locks on the file referenced by X$ but maintains the current position within the file.

```
* 100 DBRELEASE X$,RELLOCK=YES,RELPOS=NO
```

DBRETRIEVE HIGHKEY

Statement

Retrieves the high key.

AOS/VS

Format

DBRETRIEVE HIGHKEY *channel-string*[,*argument pair* ...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any argument pairs that can be used with the **DBSET** statement. Argument pairs used with **DBRETRIEVE HIGHKEY** affect the program for the **DBRETRIEVE HIGHKEY** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBRETRIEVE HIGHKEY retrieves the highest key in the selected subindex.

INFOS II returns the retrieved key to the string variable specified with the **REC** argument pair (see **DBSET**), not to one of the string variables specified with the **KEY** argument pair.

If **SETPOS=YES** is specified either by a **DBSET** statement or by a **DBRETRIEVE HIGHKEY** statement, then the current position is set to the highest key in the subindex.

Example

```
* 100 DBRETRIEVE HIGHKEY X$
```

DBRETRIEVE KEY

Statement

Retrieves the key.

AOS/VS

Format

DBRETRIEVE KEY *channel-string*[,*argument pair*,. ..]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant argument pairs that can be used with the **DBSET** statement. Argument pairs used with **DBRETRIEVE KEY** affect the program for the **DBRETRIEVE KEY** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBRETRIEVE KEY retrieves the key specified by the **ACCESS**, **MOTION**, and/or **KEY** argument pairs (see **DBSET**). If the user fails to specify the **REC** argument pair in a prior **DBSET** or in the **DBRETRIEVE** statement, then the message `ERROR 2 – STATEMENT OR COMMAND SYNTAX IS INVALID` is displayed.

INFOS II returns the retrieved key to the string variable specified with the **REC** argument pair, not to one of the string variables specified with the **KEY** argument pair.

Example

This statement returns the first key in the subindex to RECORD$ when you are positioned above a subindex.

```
* 300 DBRETRIEVE KEY X$,MOTION=DOWNFWD,REC=RECORD$
```

DBRETRIEVE SIDEF

Statement

Retrieves a subindex definition.

AOS/VS

Format

DBRETRIEVE SIDEF *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant argument pairs that can be used with the **DBSET** statement. Argument pairs used with **DBRETRIEVE SIDEF** affect the program for the **DBRETRIEVE SIDEF** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBRETRIEVE SIDEF retrieves parameters of the subindex definition linked to the selected key. The returned INFOS II status indicates whether a given key is linked to a subindex. Once you retrieve the subindex definition, you must use **DBGET** to get the returned parameters.

Example

```
* 070 DBRETRIEVE SIDEF MASTER$
```

DBRETRIEVE STATUS

Statement

Returns the INFOS II status.

AOS/VS

Format

DBRETRIEVE STATUS *channel-string*[,*argument pair*,.. .]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant **DBSET** argument pairs. These argument pairs affect the program for the **DBRETRIEVE** statement only. Argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBRETRIEVE STATUS retrieves the INFOS II status for a key and/or record. You must assign the status value to a variable. To do this, use the **DBGET** statement with the STATUS=*numeric-variable* clause (see **DBGET**).

Example

In this program fragment, INFOS II retrieves the status of the file opened on the channel string X$ and assigns the status value to STAT.

```
* 100 DBRETRIEVE STATUS X$,ACCESS=REL,MOTION=FWD
* 110 DBGET X$,STATUS=STAT
```

DBREWRITE

Statement

Rewrites a record in an INFOS II data base.

AOS/VS

Format

DBREWRITE *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant **DBSET** argument pairs. These argument pairs affect the program for the **DBREWRITE** statement only. Argument pairs used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBREWRITE is used to update the contents of an existing data record or partial record. Also it is used to write a new data record or partial record for a key that currently has no record/partial record associated with it. The updated data record may be longer or shorter than its predecessor.

DBSET

Statement

Sets an INFOS II parameter.

AOS/VS

Format

DBSET *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	You can use any of the following argument pairs:

Argument Pair	Description
ACCESS=KEY	Uses keyed access.
ACCESS=REL	Uses relative access. This is the default for **DBOPEN**.
ACCESS=RELKEY	Uses relative keyed access.
APXKEY=NO	Does not use approximate keys.
APXKEY=YES	Uses approximate keys.
DELETE=LOG	Any future deletions will be logical deletions.
DELETE=PHYS	Any future deletions will be physical deletions. This is the default for **DBOPEN**.
DUPKEY=NO	The specified key is not a duplicate.
DUPKEY=YES	The specified key is a duplicate. Set the occurrence number (see **OCCUR**).
ERR=*line-number*	Sends control to *line-number* if an error occurs. (See the section "What It Does.")
FEEDBACK=*num-variable*	Sets the feedback numeric-variable for an inverted operation.
GENKEY=NO	Does not use a generic key.
GENKEY=YES	Uses a generic key.
GLOBAL=NO	Any future locks, deletions, or reinstatements will not affect the database. This is the default for **DBOPEN**.
GLOBAL=YES	Any future locks or deletions will affect the database.

Argument Pair	Description
INVERT=NO	Do not perform an inverted write or rewrite. This is the default for **DBOPEN**.
INVERT=YES	Perform an inverted write or rewrite. Use **FEEDBACK** to set the database pointer. If you do not use **SDBASE=YES**, INFOS II rewrites the record in the data base (see **SDBASE**).
KEY=*string-variable*	INFOS II uses the current value of *string-variable* at the time of the actual INFOS II statement that requires the key. It does not use the value of the string set in the **DBSET** statement (unless this is the same value in the statement that requires the key). You cannot modify *string-variable* by INFOS II statements. You can modify *string-variables* only used with **REC** and **PREC** (discussed below).
KEYNO=*numeric-variable*	This keyword allows you to specify which level of keys will be modified by the applicable keys. It only affects the following pairs: **APXKEY=NO** **GENKEY=NO** **APXKEY=YES** **GENKEY=YES** **DUPKEY=NO** **KEY=***string-variable* **DUPKEY=YES** **OCCUR=***numeric-variable* If the **KEYNO** keyword does not appear on a line, then the above keywords affect the first key (**KEYNO=1**).
LEVELS=*numeric-variable*	Set the number of keyed access levels you will use on the next INFOS II access.
LOCAL=NO	Do not affect the partial record on subsequent **LOCK**, **DELETE**, or **REINSTATE** operations. This is the default for **DBOPEN**.
LOCAL=YES	Affect the partial record on subsequent **LOCK**, **DELETE**, or **REINSTATE** operations.
LOCK=NO	Do not lock the accessed position. This is the default for **DBOPEN**.
LOCK=YES	Lock the accessed position. When using this option, specify either **GLOBAL=YES** or **LOCAL=YES**, but not both.
LRECLEN=NO	Do not limit the record length on the next INFOS II access.

DBSET

continued

Argument Pair	Description
LRECLEN=YES	Truncate the returned database record on any subsequent reads if it will not completely fit into the record string variable (see REC=*string-variable* below). This is the default for **DBOPEN**.
MOTION=BACK	The direction of relative motion is backward.
MOTION=DOWN	The direction of relative motion is down.
MOTION=DOWNFWD	The direction of relative motion is down and forward.
MOTION=FWD	The direction of relative motion is forward. This is the default for **DBOPEN**.
MOTION=STATIC	The direction of relative motion is static.
MOTION=UP	The direction of relative motion is up.
MOTION=UPBACK	The direction of relative motion is up and backward.
MOTION=UPFWD	The direction of relative motion is up and forward.
OCCUR=*numeric-variable*	Set the occurrence *numeric-variable* for use with duplicate keys.
PREC=*string-variable*	Use *string-variable* to read and write the "partial" record. INFOS II uses the current contents of *string-variable* when the record is accessed by INFOS II. On a read, if *string-variable* is not long enough to hold the entire partial record, INFOS II truncates the record.
REC=*string-variable*	Use the specified string to read and write the "database" record. As in **KEY** and **PREC**, INFOS II uses the current contents of the string when the record is accessed with INFOS II. On the Retrieve Key and Retrieve High Key operations, INFOS II stores the returned key in the record string variable.
RELLOCK=NO	Do not release locks on a subsequent **DBRELEASE** statement. This is the default for **DBOPEN**.
RELLOCK=YES	Release locks on a subsequent **DBRELEASE** statement.
RELPOS=NO	Do not release the current position on a subsequent **DBRELEASE** statement. This is the default for **DBOPEN**.

Argument Pair	Description
RELPOS=YES	Release the current position on a subsequent **DBRELEASE** statement.
SDBASE=NO	Do not suppress the database on the following operations. This is the default for **DBOPEN**.
SDBASE=YES	Suppress the database on the following operations. INFOS II automatically suppresses the database if you have not specified a string variable prior to the INFOS II request.
SETPOS=YES	Set current position. Sets the pointer to the current accessed key after the command operation has been performed.
SETPOS=NO	Do not set current position. This is the default for **DBOPEN**. INFOS sets the pointer to the last key where the current position was set after the command operation is performed.
SPREC=NO	Do not suppress the partial record on the following operations.
SPREC=YES	Suppress the partial record on the following operations. This is the default for **DBOPEN**. INFOS II automatically suppresses the partial record if you have not specified a string variable prior to the INFOS II request.
UNLOCK=NO	Do not unlock the specified position. This is the default for **DBOPEN**.
UNLOCK=YES	Unlock the specified position.

What It Does

DBSET sets INFOS II parameters for the duration of a program. The effect is permanent, but only for *channel-string* specified in the **DBSET** statement. The ERR=*line-number* argument pair affects the program for the duration of the **DBSET** statement only. It does not set a global ERR=*line-number* parameter for the duration of the program.

If you use **DBSET**'s argument pairs in an INFOS II statement other than **DBSET**, Business BASIC uses those argument pairs for the duration of that statement.

DBSET

continued

How to Use It

Do not use **DBSET** prior to **DBOPEN INFOS**. Since **DBOPEN INFOS** initializes values in *channel-string* that you can set with **DBSET**, **DBOPEN INFOS** would overwrite options specified first with **DBSET** (see **DBOPEN INFOS**).

You can retrieve certain parameters, particularly **FEEDBACK** and **OCCUR**, with the **DBGET** statement and then pass them to another *channel-string* with **DBSET**. However, if you use the same *channel-string* for **DBGET** and **DBSET**, then the correct value is already in place and you do not need the statements.

Example

In this program fragment, **DBSET** sets the record string to RECORD$, the key string to KEYID$, specifies that the pointer should be set to the current accessed key after each command operation, and that keyed access is to be used.

```
* 100 DBOPEN INFOS FILE$,X$
* 200 DBSET X$,REC=RECORD$,KEY=KEYID$,SETPOS=YES,ACCESS=KEY
```

DBSUBINDEX DEFINE

Statement

Defines a subindex.

AOS/VS

Format

DBSUBINDEX DEFINE *channel-string*[,*argument pair* ,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any that can be used with **DBSET**, plus any of the following:

ALLOWSI=NO
Does not allow you to create lower-level subindexes (i.e., subindexes below the subindex you are defining).

ALLOWSI=YES
Allows you to create lower-level subindexes (i.e., subindexes below the level you are defining).

ALLOWDK=NO
Does not allow duplicate keys in the subindex.

ALLOWDK=YES
Allows duplicate keys in the subindex.

MAXKEYLEN=*numeric-variable*
Sets the maximum key length of the subindex.

PRECLEN=*numeric-variable*
Sets the partial record length.

RNSIZE=*numeric-variable*
Sets the root node size.

What It Does

DBSUBINDEX DEFINE defines a new subindex level in the index. INFOS II links the key on which you are positioned to the subindex you are defining.

Example

This defines a new subindex for the file referenced by X$. Duplicate keys are not permitted in the subindex, nor may any subindexes be defined for the new subindex.

```
* 160   DBSUBINDEX DEFINE X$,ALLOWSI=NO,ALLOWDK=NO
```

DBSUBINDEX DELETE *Statement*

Deletes a subindex.

AOS/VS

Format

DBSUBINDEX DELETE *channel-string*[,*argument pair* ,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any that can be used with the **DBSET** statement. Argument pairs used with **DBSUBINDEX DELETE** affect the program for the **DBSUBINDEX DELETE** statement only, whereas argument pair used with **DBSET** affect the entire program (see **DBSET**).

What It Does

DBSUBINDEX DELETE performs one of two functions:

1. Deletes a subindex or its associated database.
2. Deletes the link between the accessed key and its associated subindex.

To delete a subindex, two conditions must be met:

1. No other keys can be linked to the subindex.
2. No keys in the subindex can be linked to another subindex.

If both conditions are met, INFOS II deletes the subindex and its associated database (except for data records also linked to a different subindex). If one or none of the conditions is met, INFOS II deletes the link between the accessed key and its associated subindex (it does nothing to the subindex).

Example

```
* 190   DBSUBINDEX DELETE X$
```

DBSUBINDEX LINK

Statement

Links a subindex.

AOS/VS

Format

DBSUBINDEX LINK *channel-string,linksi-string*[*,argument pair,...*]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
linksi-string	A string initialized by **DBSUBINDEX LINKINIT** and set up by **DBSUBINDEX LINKSET**.
argument pair	Any of the following optional keyword and argument pairs:

SACCESS=*access*
Specifies the type of access you use to select the source key. (See the **DACCESS** list below.)

SMOTION=*motion*
Specifies the direction of motion (see **DMOTION** list below) you use to select the source key.

SSETPOS=NO
Does not set position to the source key.

SSETPOS=YES
Sets position to the source key.

DACCESS=*access*
Specifies the type of access you use to select the destination key. *access* can be:

KEY	keyed
REL	relative
RELKEY	relative keyed

DMOTION=*motion*
Specifies the direction of motion you use to select the destination key, where *motion* is:

FWD	forward
BACK	backward
UP	up
DOWN	down
UPFWD	up and forward
DOWNFWD	down and forward
UPBACK	up and backward
STATIC	no motion

DBSUBINDEX LINK

continued

DSETPOS=NO
Does not set position to the destination key.

DSETPOS=YES
Sets position to the destination key.

What It Does

DBSUBINDEX LINK links a key to a subindex that has been previously defined. You must provide positioning information for two keys:

- Source key — a key linked to a subindex (S)
- Destination key — the key you want to associate with the subindex (D)

Use **DBSUBINDEX LINKSET** to select the source and destination keys (see **DBSUBINDEX LINKSET**).

Example

```
* 490   DBSUBINDEX LINK X$ LINKSI$
```

DBSUBINDEX LINKINIT

Statement

Initializes a link subindex string.

AOS/VS

Format

DBSUBINDEX LINKINIT *linksi-string*[,*argument pair* ,...]

Arguments

linksi-string	The string used to specify the index locations that are to be linked.
argument pair	Any of the following: **SLEVELS=***numeric-variable* Sets the maximum number of source key levels. **DLEVELS=***numeric-variable* Sets the maximum number of destination key levels.

What It Does

DBSUBINDEX LINKINIT initializes a string so you can use it with subsequent DBSUBINDEX LINKSET statements and with one or more **DBSUBINDEX LINK** statements.

You must dimension the string variable *linksi-string*. The size of the string is computed as follows:

AOS systems:	`16 * (DLEVELS + SLEVELS) + 40`
AOS/VS systems:	`44 * (DLEVELS + SLEVELS) + 92`

If you do not use parameters for **SLEVELS** and **DLEVELS**, INFOS II uses the default value 1.

Example

```
* 770 DBSUBINDEX LINKINIT MASTER$
```

DBSUBINDEX LINKSET

Statement

Sets parameters in a link subindex string.

AOS/VS

Format

DBSUBINDEX LINKSET *linksi-string*[,*argument pair* ,...]

Arguments

linksi-string	The string used to specify the index locations that are to be linked.
argument pair	Any of the following optional keyword and argument pairs: **SKEYNO=***numeric-variable* Specifies that the following parameters in the statement apply to the source key level given by *numeric-variable*. (See **DBSET**.) **DKEYNO=***numeric-variable* Specifies that the following parameters in the statement apply to the destination key level given by *numeric-variable*. **APXKEY=NO** **APXKEY=YES** **DUPKEY=NO** **DUPKEY=YES** **GENKEY=NO** **GENKEY=YES** **KEY=***string-variable* **OCCUR=***numeric-variable*

What It Does

DBSUBINDEX LINKSET sets parameters in *linksi-string* that are used in the **DBSUBINDEX LINK** statement.

Example

```
* 570 DBSUBINDEX LINKSET MASTER$
```

DBWRITE

Statement

Writes a key and/or a record to an INFOS II file.

AOS/VS

Format

DBWRITE *channel-string*[,*argument pair*,...]

Arguments

channel-string	A string used to refer to an open INFOS II file (see **DBOPEN INFOS**).
argument pair	Any relevant **DBSET** argument pairs. Argument pairs used with **DBWRITE** affect the program for the **DBWRITE** statement only, whereas argument pairs used with **DBSET** affect the entire program (see **DBSET**).

Example

This example causes the data in string RECORD$ to be written to a data record in the database file referenced by X$ and the key contained in KEYID$ to be written to the index file referenced by X$.

```
* 100 DBWRITE X$,ACCESS=KEY,KEY=KEYID$,REC=RECORD$
```

End of Chapter

Appendix A
System Call Summary

The Business BASIC system calls are listed in this appendix. A brief description is given for each system call and an indication of whether the system call can be used on AOS/VS, DG/RDOS, and/or UNIX systems.

The types of system calls are:

STMA	Examine or modify aspects of a job, terminal, or system.
STMB	Examine or modify portions of memory.
STMC	Perform operating system calls.
STMD	Send messages to user terminals and receive responses from them. (These are available only on DG/RDOS systems.)
STME	Perform operating system calls. (These are available only on AOS/VS and UNIX systems.)
STMU	Perform operating system calls. (These are available only on UNIX systems.)

Table A-1 STMA Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMA 1	Receives a value for a terminal type, error code, passed variable, or security code.	Yes	Yes	Yes
STMA 2	Sets a value for a terminal type, error code, passed variable, or security code.	Yes	Yes	Yes
STMA 3	Examines a value for a detach key (DG/RDOS only), line cancel key, unpend key,or interrupt key.	Yes	Yes	Yes
STMA 4	Sets a value for a detach key (DG/RDOS only),line cancel key, unpend key,or interrupt key.	Yes	Yes	Yes
STMA 5	Examines a status flag for a character code, lowercase or uppercase character, or column counter.	Yes	Yes	Yes

continues

Table A-1 STMA Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMA 6	Sets to 1 a status flag for a character code, lowercase or uppercase character, or column counter.	Yes	Yes	Yes
STMA 7	Sets to 0 a status flag for a character code, lowercase or uppercase character, or column counter.	Yes	Yes	Yes
STMA 8	Resets the **FOR/NEXT** stack or IKEY indicator, or clears the input buffer.	Yes	Yes	Yes
STMA 9	Examines a username or the name of an account, the current directory, the current program, the current default output device, the system library directory, or the system directory.	Yes	Yes	Yes
STMA 10	Sets the special characters allowed in crammed strings.	Yes	Yes	Yes
STMA 11	Converts a Julian date to month/day/year format.	Yes	Yes	Yes
STMA 12	Converts a date in month/day/year format to a Julian date.	Yes	Yes	Yes
STMA 13	Translates a string of characters using a table set up by the user.	Yes	Yes	Yes
STMA 14	Translates ASCII to EBCDIC, EBCDIC to ASCII, lowercase to uppercase, or uppercase to lowercase.	Yes	Yes	Yes
STMA 15	Compares two strings and retrieves a match if there is one.	Yes	Yes	Yes
STMA 16	Detaches a job.	--	Yes	--
STMA 17	Attaches a job or process to a specified port.	--	Yes	--
STMA 18	Examines, assigns, or frees a reserved file or device.	--	Yes	--
STMA 19	Generates system errors.	Yes	Yes	Yes
STMA 20	Passes a user library file to Business BASIC.	Yes	Yes	Yes
STMA 21	Passes information for the **SFORM.SL** subroutine.	Yes	Yes	Yes

concluded

Table A-2 STMB Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMB 0	Retrieves a word address.	Yes	Yes	Yes
STMB 1	Copies the contents of memory from an address.	Yes	Yes	Yes
STMB 2	Copies words into memory.	Yes	Yes	Yes
STMB 3	Copies bytes from memory.	Yes	Yes	Yes
STMB 4	Copies the contents of a string into memory.	Yes	Yes	Yes
STMB 5	Copies the contents of memory (narrow word or double word). (Not available on AOS.)	Yes	--	Yes
STMB 6	Copies 1 or 2 words (up to 32 bits) into memory. (Not available on AOS.) Attaches a detached job to your terminal (DG/RDOS and RDOS).	Yes	Yes	Yes
STMB 7	Forces a job to BYE off the system.	--	Yes	--
STMB 8	Resets ON IKEY for a job, clears the "ignore IKEY" flag, and stops the program.	--	Yes	--
STMB 9	Not used.	--	--	--
STMB 10	Sets or clears bits in a word.	Yes	Yes	Yes
STMB 11	Tallies the 1 bits; tallies contiguous 0 bits for a string.	Yes	Yes	Yes
STMB 12	Scans the job table for the first available job and executes **HELLO** for that job. Returns -1 if no available job exists.	--	Yes	--
STMB 13	Places the contents of a string into the input buffer for the specified job.	--	Yes	--
STMB 14	Sets the job to act as if an interrupt occurred.	--	Yes	--
STMB 15	Sets the multiplexor line characteristics.	--	Yes	--
STMB 16	Sets or clears the run only flag in your program.	Yes	Yes	Yes

continues

Table A-2 STMB Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMB 17	In DG/RDOS, shuts down the multiplexor and executes RTN. In AOS/VS and UNIX, identical to STMB 20.	Yes	Yes	Yes
STMB 18	Returns the word stored at a specified address in the operating system's address space.	--	Yes	--
STMB 19	Sets the specified multiplexor line to the specified modem status.	--	Yes	--
STMB 20	Performs an immediate BYE to log off the current job.	Yes	Yes	Yes
STMB 21	Not used.	--	--	--
STMB 22	Returns the byte address of a string.	Yes	--	Yes
STMB 23	Returns the word address of a numeric variable.	Yes	--	Yes
STMB 24	Performs a system call using information in an accumulator string.	Yes	--	--
STMB 25	Maps the user channel to the system channel.	Yes	Yes	Yes

concluded

Table A-3 STMC Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMC 0	Creates a contiguous file with all locations initialized to 0.	Yes	Yes	Yes
STMC 1	Creates a subdirectory.	Yes	Yes	Yes
STMC 2	Changes the attributes for the file open on a channel.	--	Yes	--
STMC 3	Changes the link attributes of the file open on a channel.	--	Yes	--
STMC 4	Returns the 36-byte status table for the file open on a specified channel.	--	Yes	--
STMC 5	In AOS/VS, creates a control point directory. In DG/RDOS, creates a subpartition. In UNIX, creates a subdirectory.	Yes	Yes	Yes
STMC 6	In AOS/VS and UNIX, creates a file. In DG/RDOS, creates a random file.	Yes	Yes	Yes
STMC 7	In AOS/VS and UNIX, creates a file. In DG/RDOS, creates a sequential file.	Yes	Yes	Yes
STMC 8	Deletes a file.	Yes	Yes	Yes
STMC 9	Changes the current system directory.	Yes	Yes	Yes
STMC 10	Renames a device.	--	Yes	--
STMC 11	Terminates Business BASIC, passing the error code to the previous level.	Yes	Yes	Yes
STMC 12	In AOS/VS and UNIX, creates a son process and blocks Business BASIC until the son terminates. In DG/RDOS, checkpoints the background and executes the program you specify.	Yes	Yes	Yes
STMC 13	In AOS/VS and UNIX, creates a son process without blocking Business BASIC. In DG/RDOS, executes a program in the foreground partition.	Yes	Yes	Yes
STMC 14	In AOS/VS and UNIX, determines whether a process has sons. In DG/RDOS, determines whether a foreground program is running.	Yes	Yes	Yes

continues

Table A-3 STMC Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMC 15	In AOS/VS and UNIX, returns "@INPUT." In DG/RDOS, returns the name of the console input device for this ground.	Yes	Yes	Yes
STMC 16	In AOS/VS and UNIX, this call returns "@OUTPUT." In DG/RDOS, returns the name of the console output device for this ground.	Yes	Yes	Yes
STMC 17	Returns the name of the current Business BASIC system directory.	Yes	Yes	Yes
STMC 18	Returns the attributes of the file open on channel.	--	Yes	--
STMC 19	Returns the current operating system's name.	--	Yes	--
STMC 20	Initializes a device into the system.	--	Yes	--
STMC 21	Creates a link entry.	Yes	Yes	Yes
STMC 22	In AOS/VS and UNIX, gives a string a length of zero. In DG/RDOS, returns the name of the current master device.	Yes	Yes	Yes
STMC 23	Disables console keyboard interrupts.	--	Yes	--
STMC 24	Enables console keyboard interrupts.	--	Yes	--
STMC 25	Renames a file.	Yes	Yes	Yes
STMC 26	Closes all files currently open in this ground.	--	Yes	--
STMC 27	Releases a previously initialized device.	--	Yes	--
STMC 28	Returns control to the program at the previous push level.	Yes	Yes	Yes
STMC 29	Sets the current system date.	--	Yes	--
STMC 30	Disables spooling for a device.	--	Yes	--
STMC 31	Enables spooling for a device.	--	Yes	--
STMC 32	Kills spooling for a device.	--	Yes	--
STMC 33	Returns the 18-byte status of the file you specify.	--	Yes	--
STMC 34	Sets the time of day.	--	Yes	--

continues

Table A-3 STMC Summary Table

System Call	Description	AOS/VS	DG/RDOS	UNIX
STMC 35	Removes a link entry.	Yes	Yes	Yes
STMC 36	Updates the disk resident copy of the file open on *channel.*	Yes	Yes	Yes
STMC 37	Returns the frequency of the system's real-time clock.	Yes	Yes	Yes
STMC 38	Opens a file for direct magnetic tape I/O.	Yes	Yes	--
STMC 39	Opens a file for appending.	Yes	Yes	Yes
STMC 40	Opens a file for exclusive use.	Yes	Yes	Yes
STMC 41	In AOS/VS, opens a file for input only. In DG/RDOS opens a file for read-only access. In UNIX, opens a file in read-only mode.	Yes	Yes	Yes
STMC 42	In AOS/VS and UNIX, opens a file for input and output. In DG/RDOS, opens a file for shared access.	Yes	Yes	Yes
STMC 43	Terminates Business BASIC and saves a break file.	Yes	Yes	Yes
STMC 44	In DG/RDOS, opens a file for exclusive access transparently. In AOS/VS and UNIX, opens the file for input and output.	Yes	Yes	Yes
STMC 45	Transparently creates a contiguous file.	--	Yes	--
STMC 46	Transparently creates a random file.	--	Yes	--
STMC 47	Transparently creates a sequential file.	--	Yes	--
STMC 48	Gets the current memory allocation for the current ground.	--	Yes	--
STMC 49	Not used.	--	--	--
STMC 50	Writes a line to the file open on a specified channel.	Yes	Yes	Yes
STMC 51	Returns the status of the resolution file when you specify a link entry.	--	Yes	--
STMC 52	Creates a contiguous file without initializing the file to nulls.	Yes	Yes	Yes
STMC 53	Closes the file open on a channel.	--	Yes	--
STMC 54	Gets the date last modified for a file.	Yes	--	Yes

concluded

Table A-4 STMD Summary Table (DG/RDOS Only)

System Call	Description
STMD 0	Sends a message and can receive a reply, but can't override a no-message flag set by the terminal.
STMD 1	Sends a message and can receive a reply; can also override a no-message flag set by the terminal.

Table A-5 STME Summary Table (AOS/VS and UNIX Only)

System Call	Description	AOS/VS	UNIX
STME 0	Starting from a specified position, returns the next filename matching the template in the directory open on *channel*.	Yes	--
STME 1	Returns the process's username.	Yes	Yes
STME 2	Returns the process's PID.	Yes	Yes
STME 3	Returns a CLI message.	Yes	--
STME 4	Returns the complete pathname of *filename*.	Yes	Yes
STME 5	Returns the 23-word (46-byte) status packet for *channel* in the string variable *buffer$*.	Yes	--
STME 6	Returns the status of a file.	Yes	--
STME 7	Returns the status of the resolution file when you specify a link entry.	Yes	--
STME 8	Creates a contiguous file.	Yes	--
STME 9	Creates a file.	Yes	--
STME 10	Returns the complete pathname of the file open on *channel*.	Yes	Yes
STME 11	Returns the resolution name of a link.	Yes	Yes
STME 12	Returns the device characteristics of @INPUT.	Yes	--
STME 13	Returns the device characteristics of @OUTPUT.	Yes	--
STME 14	Sets the device characteristics of @INPUT.	Yes	--
STME 15	Sends a message to a specified process.	Yes	--
STME 16	Sends a message to the process controlling the console.	Yes	--
STME 17	Returns the 256-bit (32-byte) delimiter table for @INPUT (AOS/VS) or `stdin` (UNIX) into a specified variable.	Yes	Yes

continues

Table A-5 STME Summary Table (AOS/VS and UNIX Only)

System Call	Description	AOS/VS	UNIX
STME 18	Sets the 256-bit (32-byte) delimiter table for @INPUT (AOS/VS) or `stdin` (UNIX).	Yes	Yes
STME 19	Executes a specified program passing a string to the new process as the initial IPC.	Yes	Yes
STME 20	Sends a string via interprocess communications.	Yes	Yes
STME 21	Receives the contents of a string via interprocess communications.	Yes	Yes
STME 22	Sends the contents of a string via interprocess communications, and then receives an interprocess communications message.	Yes	Yes
STME 23	Creates an IPC file entry with the appropriate name and local port.	Yes	Yes
STME 24	Finds the owner of a global port.	Yes	Yes
STME 25	Looks up a port number.	Yes	Yes
STME 26	Translates a port number.	Yes	Yes
STME 27	Moves contents of a string to an I/O buffer.	Yes	Yes

concluded

Table A-6 STMU Summary Table (UNIX Only)

System Call	Description
STMU 0	Returns the fstat status packet to *buffer$* for the file opened on *channel.*
STMU 1	Returns the status of a filename.
STMU 2	Returns the status of the resolution file for a link entry.
STMU 3	Creates a file with a time and date.
STMU 4	Returns the switches used during invocation of Business BASIC.
STMU 5	Returns the contents of the specified environment variable.

End of Appendix

Appendix B
Resource Limits

The Business BASIC resource limits vary depending on which operating system you use. This appendix shows the resource limits for Business BASIC running on AOS/VS, DG/RDOS, and UNIX systems.

Table B-1 Business BASIC Resource Limits

Feature	UNIX	AOS/VS	DG/RDOS
Attaching/detaching allowed	No	No	Yes
INFOS interface implemented	No	Yes	No
Combined program and data space	512 Kbytes	256 Kbytes	22–38 Kbytes
Number of channels per user[1]	Op. system	150	16
Number of channels systemwide	Op. system	255 x number of processes	256
Length of program statement	256	256	132
Screenedit available	Yes	Yes	No
Number of user variables per program	8192	348	348
Variable name length	32	6	6
GOSUB nesting levels	32	32	8
DEF nesting levels	26	4	4
FOR...NEXT nesting levels	32	32	8
Maximum line number[2]	99999	32767	32767

continues

1 You can have 150 channels under Business BASIC. If you use the INFOS interface, you can also have 64 INFOS channels.

2 DG/RDOS and RDOS can have up to 32,767 line numbers. The maximum for AOS is 9,999.

Table B-1 Business BASIC Resource Limits

Feature	UNIX	AOS/VS	DG/RDOS
Size of common area	2048	512	512
Highest precision allowed	Quad (8 bytes)	Triple (6 bytes)	Triple (6 bytes)
Maximum index file size	128 Mbytes	128 Mbytes	32 Mbytes
Maximum data file size	Op. system	2 Gbytes	32 Mbytes
Maximum string length	522 Kbytes	65,535 bytes	32,767 bytes
Number of expressions allowed in MAX and MIN comparisons	16	2	2
Number of variables allowed in DEF statements	16	1	1

concluded

End of Appendix

Index

Related Documents

Subroutines, Utilities, and the Business BASIC CLI	093-000389

Defines each Business BASIC subroutine, utility, and CLI command. It is an alphabetical reference manual for programmers.

Using Business BASIC on DG/UX™ and INTERACTIVE UNIX® Systems	093-000685

Describes how to load and generate Business BASIC on DG/UX and INTERACTIAVE UNIX systems. It is intended for the system manager or system operator.

Business BASIC System Manager's Guide	093-000388

Describes how to load and generate Business BASIC on AOS, AOS/VS, AOS/VS II, RDOS, and DG/RDOS systems. It is intended for the system manager or system operator.

Learning Business BASIC	093-000684

Acquaints experienced programmers with Business BASIC operations and programming procedures for DG/UX and 386/ix systems. It provides an overview of the commands, functions, subroutines, and utilities available to programmers.

Programming with Business BASIC	093-000480

Acquaints experienced programmers with Business BASIC operations and programming procedures for AOS, AOS/VS, RDOS, and DG/RDOS systems. It provides an overview of the commands, functions, subroutines, and utilities available to programmers.

AOS INFOS® II System User's Manual	093-000152

Provides information on using the AOS INFOS II file management system.

AOS/VS INFOS® II System User's Manual	093-000299

Provides information on using the AOS/VS INFOS II file management system.

Business BASIC Summary	069-000263

Summarizes Business BASIC commands, statements, functions, and utilities in a pocket-sized book for programmers using AOS, AOS/VS, RDOS, and DG/RDOS systems.

DASHER® D2 File Maintenance and Screen Maintenance Template	093-000212
DASHER® D200 File Maintenance and Screen Maintenance Template	093-000265
DASHER® D210/211 D410/D460 CFM and CSM Template	093-000409
DASHER® D210/211 D410/D460 SM and FM Template	093-000410

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to: Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME	COMPANY NAME
ATTN:	ATTN:
ADDRESS	ADDRESS (NO PO BOXES)
CITY	CITY
STATE ZIP	STATE ZIP

Priority Code ____________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable)

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is________. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION

Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES

Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS

Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY

DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY

EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY

A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL

A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)

Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket