jBASE BASIC Programmers Reference Guide

Functions and Statements

Contents

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
BOLD	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates JBASE commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates JBASE identifiers such as filenames, account names, schema names, and Windows NT filenames and pathnames.
UPPERCASE Italic	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, filenames, and pathnames.
`Courier`	`Courier` indicates examples of source code and system output.
`Courier Bold`	**`Courier Bold`** In examples, courier bold indicates characters that the user types or keys (for example, <Return>).
[]	Brackets enclose optional items. Do not type the brackets unless indicated.
{}	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
ItemA ¦ itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
. . .	Three periods indicate that more of the same type of item can optionally follow.
⇒	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File ⇒Exit**" means you should choose **File** from the menu bar, and then choose **Exit** from the File pull-down menu.

Syntax definitions and examples are indented for ease in reading.

All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.

Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

Preface

This manual is a comprehensive reference for jBASE BASIC and is intended for experienced programmers. The guide includes explanations of all jBASE BASIC statements and functions supported by jBASE and descriptive information regarding the use of jBASE BASIC in the UNIX environment.
If you have never used jBASE BASIC, read this manual before using any statements or functions.

Organization of this manual

This manual contains statements and functions in alphabetical order, each beginning on a new page. At the top of each page is the syntax for the statement or function, followed by a detailed description of its use, often including references to other statements or functions that can be used with it or are helpful to know about. Examples illustrate the application of the statement or function in a program.

jBASE BASIC Language Overview

- is a UNIX resident programming language supported by the jBASE Database Independent Management Engine
- can access database files of any UNIX resident, Open Systems database
- is aimed primarily at writing business applications, and contains all the constructs needed to access and modify files and their data efficiently
- is a sophisticated superset of Dartmouth BASIC supporting structured programming techniques
- is a flexible and user extendible language
- contains the functionality needed to write efficient UNIX applications. It can spawn child processes, access environment variables and interface to other UNIX programs
- programs can call external functions written in C or jBASE BASIC. C programs can be made to call functions written in jBASE BASIC
- programs can mix with Embedded SQL statements written allowing queries and updates on any SQL Database
- object code is link compatible with C and so a programmer has the tools of both available to him to produce the most efficient code for his application
- Allows the application programmer working in a UNIX environment to write code without needing to consider memory management, variable typing or floating-point arithmetic corrections: all of which need to be dealt with when using 'C'
- Has other advantages over C such as the in-built debugger and easy file I/O;
- Programs may declare external functions, which are linked into the application by the UNIX linker-loader. This means that jBASE BASIC offers access to specialized functions written in C or any language that is link compatible with C

Features of jBASE BASIC

- Optional statement labels
- Multiple statements on one line
- Local subroutine calls
- Branching on result of complex value testing
- String handling with variable lengths
- External calls to 'C' libraries
- External subroutine calls
- Direct and indirect calls
- Magnetic tape input and output
- String, number, and date data conversion capability;
- File access and update capability for any UNIX resident file, such as j-files or C-ISAM)
- File and record level locking capability
- Pattern matching capability
- Capability of processing file records in any format
- Sophisticated jBASE BASIC debugger
- Ability to EXECUTE any jBASE system or database enquiry command
- The standard UNIX command set is available to manage code libraries
- Support for networking and inter-process communication.

Benefits of using jBASE BASIC

- Applications are running on an Open Systems platform:
- Applications are very efficient as the execution speed of jBASE BASIC code is close to that of hand crafted 'C'
- Applications are portable between binary compatible environments, however moving applications to an alternative operating system requires that the application be recompiled on the target system. No modifications to the application source are required as any operating specific modifications will have been implemented by jBASE in the runtime libraries.
- Applications integrate easily with other UNIX systems
- Applications benefit from the steady improvements made in compiler optimization.
- Use of jBASE BASIC offers tremendous productivity improvements over 'C'
- The close compatibility with UNIX allows the jBASE BASIC developer to produce libraries of standard subroutines or function calls, which any program can use
- The standard UNIX command set is available to manage code libraries
- The provision of Database access is to applications through generic read/write/lock statements that divorce the application from the database itself. Locks are maintained across remote systems and communication links thus allowing the application programmer to concentrate on the application not the database or its location

- JBASE BASIC will import and compile BASIC code from Open Systems RDBMS systems with little or no modification
- Applications ported from PICK or Reality run as 'C' applications with all the related performance and seamless inter-operability advantages over running on an emulation type implementation written in C
- Investments in existing jBASE BASIC applications and development and programming skills in BASIC are fully retained
- No need for costly retraining of programmers to 'C', which can also be freely used within the application system, thus allowing more flexibility
- JBASE BASIC provides connection to external devices and external databases in a manner that is transparent to existing applications

jBASE BASIC Environment

- jBASE BASIC will run on any standard UNIX system and with any standard shell or editor. Also provided is an easy to use jSHELL.
- jBASE BASIC allows the programmer to choose his working environment to suit. It works equally with the Bourne, C or Korn shell. Kernel configuration is not required to use the jBASE BASIC-programming environment.
- You can write jBASE BASIC programs using any UNIX editor using the provided context sensitive screen editor (jED), designed specifically for jBASE BASIC programmers and jBASE users.
- Utilities are supplied to access database files created under jBASE.
- The final size of executable code is minimized, and duplication avoided, by sharing external object libraries between all application programs.
- Specify a file or directory to hold the entire jBASE BASIC source; you can hold the finished executables in a different file or directory if required.
- Use a global user library to hold globally accessible user routines.

jBASE BASIC Programming

- You can write the jBASE BASIC source code using any system editor. Users unfamiliar with UNIX editors may wish to use the jED editor
- Use the jBASE BASIC compiler to produce intermediate object code or a UNIX executable file; use Makefiles to simplify the compilation process, especially if many files are involved. Their use will also make upgrading and software maintenance an easier task
- If the system allows, use should be made of linked libraries when calling subroutines and functions. This will reduce the size of the compiled code that would otherwise be produced
- Applications accessing jBASE files should make use of the existing routines held in the /usr/jBASE BASIC/lib directory.

jBASE BASIC Comparisons

With BASIC

Derived from Dartmouth Basic jBASE BASIC is an enhanced variant of BASIC, which contains all the commands and constructs necessary for compatibility with other versions of BASIC. It also provides full interaction with UNIX system and database files. You can modify jBASE BASIC quickly to retain compatibility with any future enhancements to the BASIC language or its derivatives

On UNIX systems, the jBASE BASIC compiler produces code that runs many times faster than the same BASIC code compiled and run on any other UNIX based RDBMS environment. jBASE BASIC can access jBASE, C-ISAM, and UNIX files as well as records and files of other databases The jBASE BASIC debug facilities are greatly superior to those provided with other versions of BASIC

With 'C'

The jBASE BASIC compiler uses all the features of the cc compiler and can compile 'C' source and object files, as well as jBASE BASIC source code. You can halt the source compilation at any stage, to examine the resultant code:

External 'C', and jBASE library access is available

The executables produced by the jBASE BASIC compiler and cc is identical

- jBASE BASIC has a sophisticated debugger available as standard
- jBASE BASIC is able to provide full and easy access to UNIX or any third party database files
- jBASE BASIC has the tools to provide sophisticated string handling
- jBASE BASIC handles system signals and events automatically

File and Directory Organization

To run jBASE BASIC on a UNIX system, there are several directories and files already set up, which ensure the smooth and efficient use of the jBASE BASIC programming environment; all the jBASE BASIC files are held under the UNIX /opt/jBASIC directory.

The main body of the jBASE BASIC program and library files are held in the /opt/jBASIC directory, which contains all the run-time code, error and library files, as well as default system and terminal set-up limit.

XML Functions and Statements

JBASE is incorporating new XML capabilities built into jBASE BASIC based on the Xalan and Xerces libraries.

XML Functions

DYNTOXML

XMLTODYN

XMLTOXML

XML Statements

READXML

WRITEXML

jBASE Funtions and Statement @ Variables

@

Use the @ function to position the cursor to a specific point on the terminal screen

COMMAND SYNTAX

@ (col{, row})

SYNTAX ELEMENTS

col and row can be any expression that evaluates to a numeric value.

col specifies, to which column on the screen the cursor should be moved.

row specifies which row (line) on the screen to position the cursor.

Specifying col on its own will locate the cursor to the required column on whichever row it currently occupies.

NOTES

When specified values exceed either of the physical limits of the current terminal, then unpredictable results will occur.

The terminal address starts at (0,0), that being the top left hand corner of the screen.

Cursor addressing will not normally work when directed at a printer. If you wish to build printer independence into your programs, achieve this by accessing the terminfo database through the SYSTEM () function.

EXAMPLES

```
FOR I = 1 TO 5

CRT @(5, I):"*":

NEXT I

Home = @(0,0) ;* Remember the cursor home position

CRT Home:"Hi honey, I’m HOME!":
```

@ (SCREENCODE)

Use @(SCREENCODE) to output control sequences according to the capabilities of the terminal

COMMAND SYNTAX

@ (ScreenCode)

SYNTAX ELEMENTS

Control sequences for special capabilities of the terminal are achieved by passing a negative number as its argument. ScreenCode is therefore any expression that evaluates to a negative argument.

NOTES

The design of jBASE allows you to import code from many older systems. As these systems have traditionally not co-ordinated the development of this function they expect different functionality in many instances. In the following table, you should note that different settings of the JBASICEMULATE environment variable would elicit different functionality from this function. Where the emulate code is printed with strikethrough it indicates that the functionality is denied to this emulation.

Emulation	Code	Function
all	-1	clear the screen and home the cursor
all	-2	home the cursor
all	-3	clear screen from the cursor to the end of the screen
all	-4	clear screen from cursor to the end of the current screen line
ros	-5	turn on character blinking
ros	-6	turn off character blinking
ros	-7	turn on protected field mode
ros	-8	turn off protected field mode
all	-9	move the cursor one character to the left
all	-10	move the cursor one row up the screen
ros	-11	turn on the cursor (visible)
ros	-11	enable protect mode
ros	-12	turn off the cursor (invisible)
ros	-12	disable protect mode
ros	-13	status line on
ros	-13	turn on reverse video mode
ros	-14	status line off
ros	-14	turn off reverse video mode
ros	-15	move cursor forward one character
ros	-15	turn on underline mode
ros	-16	move cursor one row down the screen
ros	-16	turn off underline mode

Emulation	Code	Function
all	-17	turn on the slave (printer) port
all	-18	turn off the slave (printer) port
ros	-19	dump the screen to the slave port
ros	-19	move the cursor right one character
ros	-20	move the cursor down one character
ros	-311	turn on the cursor (visible)
ros	-312	turn off the cursor (invisible)
ros	-313	turn on the status line
ros	-314	turn off the status line

If a color terminal is in use, -33 to -64 will control colors.
The codes from -128 to -191 control screen attributes. Where Bit 0 is least significant, you may calculate the desired code by setting Bit 7 and Bits 0-4:

Bit 0	dimmed mode when set to 1
Bit 1	flashing mode when set to 1
Bit 2	reverse mode when set to 1
Bit 3	blanked mode when set to 1
Bit 4	underline mode when set to 1
Bit 5	bold mode when set to 1
Bit 7	always set to 1

Thus, Reverse and Flashing mode is -134.
To turn off all effects use -128

EXAMPLE

```
CRT @ (-1):@(30):@( 132):"jBASE Heading":@(-128):

CRT @ (5,5):@(-4):"Prompt: ": ; INPUT Answer
```

@APPLICATION.ID @ID Dataname used to reference the record-id in a query language statement:

SORT STOCK BY-DSND @ID

LIST STOCK WITH @ID = "1000"

LIST STOCK WITH @ID LIKE AB...

@CALLSTACK Returns current space information for DEBUG purposes

@CODEPAGE Returns cuurnt codepage config jbase_codepage

@DATA Data statements used in conjunction with INPUT statements are stored in a data stack or input queue. This stack is accessible in the @DATA variable

@DATE internal date returns the internal date – on some systems, this differs from the DATE function in that the variable is set when program execution starts, whereas the function reflects the current date

@DAY Day of month from @DATE

@EOF End of File character from TTY characteristics

@FILENAME Current filename

@FOOTER.BREAK For B options in heading

@HEADER.BREAK For B options in heading

@LEVEL The nesting level of execution statements – non stacked

@LOCALE Returns current Locale as jbase_locale

@LPTRHIGH Number of lines on the device to which you are printing (that is, terminal or printer).

@MONTH Current Month

@PARASENTENCE The last sentence or paragraph that invoked the current process.

@PATH Pathname of the current account

@PID Returns current process ID

@RECORD Entire current record

@SELECTED Number of elements from the last select list – Non stacked

@TERMTYPE The Terminal type

@TIME Returns the internal time – on some systems, this differs from the TIME function in that the variable is set when program execution starts, whereas the function reflects the current time

@TIMEZONE As per jBASE Timezone

@TTY Returns the terminal port name.

@UID Returns information from ROOT.THREAD for port @user

@USER.ROOT The use of the @USER.ROOT command allows a jBASE BASIC program to store and retrieve a string of up to 63 bytes that is unique to that user. The intention is to really "publish" information that other programs can find.

For example

```
@USER.ROOT = "Temenos T24 Financials"

.....

PRINT "root user declaration is " : @USER.ROOT
```

See attribute <28> , USER_PROC_USER_ROOT, in the section "Layout of user record"

The @USER.THREAD is similar except a value exists for each PERFORM level. So one program can set/retrieve it but if the program does a PERFORM of a second program then the second program gets a different set of values.

See attribute <52> , USER_PROC_USER_THREAD, in the section "Layout of user record"

The @USERSTATS allows a program to retrieve all sorts of miscellanous information about itself. For example if a program wants to find out how many database I/O's it performed it could do this ...

```
INCLUDE JBC.h

info1 = @USERSTATS
```

```
read1 = info1<USER_PROC_STATS_READ>

EXECUTE 'COUNT fb1 WITH *A1 EQ "x"'

info2 = @USERSTATS

read2 = info2<USER_PROC_STATS_READ>

PRINT "The COUNT command took ":(read2-read1):" READ's from the database"
```

So a program can set a user-definable string to whatever value it likes , up to 63 bytes, and other programs can use various methods (see "User Information Retrieval" below) to access this data.

User Information Retrieval

There are 3 ways of finding information about one or more users on a jBASE system

1. Using the @USER.ROOT, @USER.THREAD and @USERSTATS variables in your jBASE BASIC code you can find information about yourself. You cannot find information about other users.

2. The "WHERE (V)" command can be used to display the @USER.ROOT and @USER.THREAD data for specified users.

3. Using some jBASE BASIC code you can find out lots of information about each user on the system. This is exactly the mechanism that the WHERE command uses. For example to display all users logged on you could write this.

```
*

* Open the special jEDI file to access the user information.

*OPEN SYSTEM(1027) TO PROC ELSE STOP 201,SYSTEM(1027)

*

* For each user logged on read in their user information

*

   SELECT PROC
```

```
   LOOP WHILE READNEXT key DO
      READ rec FROM PROC,key THEN
*
         PRINT "Port ":rec<USER_PROC_PORT_NUMBER>:" is logged on by user ":rec<USER_PROC_ACCOUNT>
*
      END
   REPEAT
```

Layout of user record

The information retrieved by either the READ in the above example or the @USERSTATS is the same and is as follows.

The first 40 attributes are data attributes that correlate to the entire user. Attributes 41 onwards are multi-valued and have one value per program being PERFORM'ed by that user

All the numbers below can be replaced by symbolic references in JBC.h , look for those that begin USER_PROC_

<1> The port number

<2> The number of programs running in this port.

<3> Time the user started in Universal Co-ordinated Time or UTC (not a dyslexic mistake). This is raw UNIX time. You can convert this to jBASE internal time format using the U0FF0 conversion or to internal date format using the U0FF1 conversion.

<4> The process ID

<5> Account name

<6> User name. Normally the operating system name.

<7> Terminal name in jBASE format

<8> Terminal name in Operating system format.

<9> Database name

<10> TTY device name

<11> Language name.

<12> Time in UTC the listening thread last found the thread alive.

<13> Amount of heap space memory in free space chain on a process wide basis. Not real-time, only updated every 15 seconds.

<14> Amount of heap space memory in use on a process wide basis. Not real-time , only updated every 15 seconds

<15> Thread type as an internal integer.

<16> Type of thread as a text string.

<17> License counters

<18> Number of OPEN's performed.

<19> Number of READ's performed.

<20> Number of WRITE's performed.

<21> Number of DELETE's performed

<22> Number of CLEARFILE's performed

<23> Number of PERFORM/EXECUTE's performed.

<24> Number of INPUT's performed.

<25> Not used.

<26> Number of jBASE files the application thinks it has open at the moment.

<27> Number of jBASE files actually opened by the operating system at the moment.

<28> Any data set by the application using @USER.ROOT

<29> Process Identifier. A string created by the operating system to identify the process. It is O/S specific. Currenly on IBM i-series platform only.

<30> to <40> Reserved.

Attributes 41 onward are multi-valued, one value per perform level, and there are <2> perform levels active.

<41,n> Program name and command line arguments.

<42,n> The line number in jBASE BASIC the program is currently executing.

<43,n> The source name in jBASE BASIC the program is currently executing.

<44,n> Not used.

<45,n> Not used.

<46,n> Status of program execution as a readable text string.

<47,n> Status of program execution as an internal integer.

<48,n> User CPU time . Depending upon the hardware this will be either for the entire process or just the single thread.

<49,n> System CPU time.Depending upon the hardware this will be either for the entire process or just the single thread.

<50,n> User CPU time used by any external child processes it might have spawned.

<51,n> System CPU time used by any external child processes it might have spawned.

<52,n> Any data set by the application using @USER.THREAD

@USERSTATS

The @USERSTATS allows a program to retrieve miscellaneous information about itself. For example if a program wants to find out how many database I/O's it performed it could do this

```
info1 = @USERSTATS
read1 = info1<19>
EXECUTE 'COUNT fb1 WITH *A1 EQ "x"'
info2 = @USERSTATS
read2 = info2<19>
PRINT "The COUNT command took ":(read2-read1):" READ's
```

from the database"

The following definitions have been added to JBC.h file which defines the layout of data returned either through the @USERSTATS variable or by opening file SYSTEM(1027) and reading the items in like that.

* Definitions for the data returned from the @USERSTATS variable or from

* the record read in from the PROC file (using SYSTEM(1027) as file name)

*

EQUATE USER_PROC_PORT_NUMBER TO 1;* The port number

EQUATE USER_PROC_NUM_PROGRAMS TO 2;* Number of programs running in this port

EQUATE USER_PROC_START_TIME TO 3;* Time user started in UTC format

EQUATE USER_PROC_PID TO 4 ;* Process ID

EQUATE USER_PROC_ACCOUNT TO 5;* Name of the account

EQUATE USER_PROC_USER TO 6 ;* Name of the user

EQUATE USER_PROC_TERMINAL_JBASE TO 7;* Name of terminal according to jBASE

EQUATE USER_PROC_TERMINAL_OS TO 8;* Name of terminal as seen by OS

EQUATE USER_PROC_DATABASE TO 9;* Name of database connected to

EQUATE USER_PROC_TTY TO 10;* Name of TTY device

EQUATE USER_PROC_LANGUAGE TO 11;* Language

EQUATE USER_PROC_LISTENING_TIME TO 12;* Time in UTC the listening thread last worked

EQUATE USER_PROC_MEM_FREE TO 13;* Amount of

memory in heap space free chain

EQUATE USER_PROC_MEM_USED TO 14;* Amount of heap space memory in use

EQUATE USER_PROC_THREAD_TYPE_INT TO 15;* Thread type expressed as an integer

EQUATE USER_PROC_THREAD_TYPE_TXT TO 16;* Thread type expressed as a text string

EQUATE USER_PROC_LICENSE TO 17;* License counters

EQUATE USER_PROC_STATS_OPEN TO 18;* Number of OPEN's performed.

EQUATE USER_PROC_STATS_READ TO 19;* Number of READ's performed.

EQUATE USER_PROC_STATS_WRITE TO 20;* Number of WRITE's performed.

EQUATE USER_PROC_STATS_DELETE TO 21;* Number of DELETE's performed.

EQUATE USER_PROC_STATS_CLEARFILE TO 22;* Number of CLEARFILE's performed.

EQUATE USER_PROC_STATS_PERFORM TO 23;* Number of PERFORM's / EXECUTE's performed.

EQUATE USER_PROC_STATS_INPUT TO 24;* Number of INPUT's performed.

EQUATE USER_PROC_UNUSED_1 TO 25;* Unused

EQUATE USER_PROC_OPEN_FILES_VIRTUAL TO 26 ;* Number of files application thinks open

EQUATE USER_PROC_OPEN_FILES_REAL TO 27 ;* Number of files really open by OS

EQUATE USER_PROC_USER_ROOT TO 28;* Application data set by @USER.ROOT

EQUATE USER_PROC_PROCESS_TXT TO 29;* Text string to

identify process

EQUATE USER_PROC_PROGRAM TO 41;* Program name and command line arguments

EQUATE USER_PROC_LINE_NUMBER TO 42;* Line number currently being executed.

EQUATE USER_PROC_SOURCE_NAME TO 43;* Name of source currently being executed.

EQUATE USER_PROC_UNUSED_2 TO 44;* Unused

EQUATE USER_PROC_UNUSED_3 TO 45;* Unused

EQUATE USER_PROC_STATUS_TXT TO 46;* Status of program as a readable text

EQUATE USER_PROC_STATUS_INT TO 47;* Status of program as an integer

EQUATE USER_PROC_CPU_USR TO 48;* User CPU time

EQUATE USER_PROC_CPU_SYS TO 49;* System CPU time

EQUATE USER_PROC_CPU_USR_CHILD TO 50;* User CPU time used by child processes

EQUATE USER_PROC_CPU_SYS_CHILD TO 51;* System CPU time used by child processes

EQUATE USER_PROC_USER_THREAD TO 52;* Application data set by @USER.THREAD

1=PORT	2=count of programs on this port	3=Start time in UTC	4=Process ID	5=Account name
6=user name	7=terminal name 9Jbase)	8=terminal name (OS)	9=database name	10=tty device name
11=Language	12=time in UTC last found alive	13=free heap space (15 secs)	14heap space used (15 secs)	15=thread type
16=thread	17=license	18=count of opens	19=count of	20=count of writes

type (string)	counters		reads	
21= count of DELETES	22=count of Clear Files	23=count of PERFORMS/EXECUTES	24=count of INPUTS	25=NOT USED
26=number of files open (jBASE)	27=number of files open(Actual)	28-@USER.ROOT	29=Process identifier	30-40 reserved
41=program name and sentence	42=Current line number	42=source name	46=status of program text	47=status of program (flag)
48=USER CPU time	49=System CPU time	50=USER CPU from child processes	51=System CPU time from child procs	52=@USER.THREAD

@USER.THREAD

A value exists for each PERFORM level. So one program can set/retrieve it but if the program does a PERFORM of a second program then the second program gets a different set of values.

Allows an application to store simple statistical information about the thread level part of their data.

jBASE BASIC Functions and Statements A - X

The following pages show the syntax of every statement and function in the language together with examples of their use.

ABORT

The ABORT statement terminates the current running program and the program that called it.

COMMAND SYNTAX

ABORT {message.number{, expression ...}}

SYNTAX ELEMENTS

The optional message.number provided with the statement must be a numeric value, which corresponds to a record key in the jBASE error message file.
A single expression or a list of expression(s) may follow the message.number. Where more than one expression is listed, they must be delimited by the use of the comma character. The expression(s) correspond to the parameters that need passing to the error file record to print it.
The optional message.number and expression(s) given with the command are parameters or resultants provided as variables, literal strings, expressions, or functions.

NOTES

Use this statement to terminate the execution of a jBASE BASIC program together with any calling program. It will then optionally display a message, and return to the shell prompt.
The error file holds the optional message displayed on terminating the program. For successful printing of the message, parameters such as linefeeds, clearscreen, date and literal strings may also be required.
Setting the Command Level Restart option can alter operation of this command.

EXAMPLE

```
CRT "CONTINUE (Y/N) ?":; INPUT ANSIF ANS NE "Y" THEN ABORT 66,

"Aborted"
```

This will terminate the program and print error message 66 passing to it the string "Aborted", which will be printed as part of error message 66.

ABS

ABS returns the mathematical absolute of the ()expression

COMMAND SYNTAX

ABS (expression)

SYNTAX ELEMENTS

expression can be of any form that should evaluate to a numeric. The ABS function will then return the mathematical absolute of the expression. This will convert any negative number into a positive result.

NOTES

express this as: value < 0 ? 0 - value : value

EXAMPLES

```
CRT ABS (10-15)
```

Displays the value 5

```
PositiveVar = ABS (100-200)
```

Assigns the value 100 to the variable PositiveVar

ABSS

Use the ABSS function to return the absolute values of all the elements in a dynamic array. If an element in the dynamic array is null, it returns null for that element.

COMMAND SYNTAX

ABSS (dynamic.array)

EXAMPLE

```
Y = REUSE(300)

Z = 500:@VM:400:@VM:300:@SM:200:@SM:100

A = SUBS (Z,Y)

PRINT A

PRINT ABSS (A)
```

The output of this program is:

```
200]100]0\-100\-200
200]100]0\100\200
```

ADDS

Use ADDS to create a dynamic array of the element-by-element addition of two dynamic arrays. Added to each element of array1 is the corresponding element of array2, which returns the result in the corresponding element of a new dynamic array. If an element of one array has no corresponding element in the other array, it returns the existing element. If an element of one array is the null value, it returns null for the sum of the corresponding elements.

COMMAND SYNTAX

ADDS (array1, array2)

EXAMPLE

```
A=2:@VM:4:@VM:6:@SM:10
B=1:@VM:2:@VM:3:@VM:4
PRINTADDS (A,B)
```

The output of this program is:

```
3]6]9\10]4
```

ALPHA

The ALPHA function will check that the expression consists entirely of alphabetic characters.

COMMAND SYNTAX

ALPHA (expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ALPHA function will then return TRUE (1) if the expression consists entirely of alphabetic characters else returns false (0) if any character in expression is non alphabetic.

INTERNATIONAL MODE

When using the ALPHA function in International Mode it determines the properties of each character in the expression according to the Unicode Standard, which in turn describes whether the character is alphabetic or not.

NOTES

Alphabetic characters are in the set a-z and A-Z

EXAMPLE

```
Abc = "ABC"

IF ALPHA (Abc) THEN CRT "alphabetic"

Abc = "123"

IF NOT (ALPHA(Abc)) THEN CRT "non alphabetic"
```

Displays:
alphabetic
non alphabetic

ANDS

Use the ANDS function to create a dynamic array of the logical AND of corresponding elements of two dynamic arrays.

Each element of the new dynamic array is the logical AND of the corresponding elements of array1 and array2. If an element of one dynamic array has no corresponding element in the other dynamic array, it returns a false (0) for that element.

If both corresponding elements of array1 and array2 are null, it returns null for those elements. If one element is the null value and the other is zero or an empty string, it returns false for those elements.

COMMAND SYNTAX

ANDS (array1, array2)

EXAMPLE

```
A = 1:@SM:4:@VM:4:@SM:1

B = 1:@SM:1-1:@VM:2

PRINT ANDS (A,B)
```

The output of this program is: 1\0]1\0

ASCII

The ASCII function converts all the characters in the expression from the EBCDIC character set to the ASCII character set.

COMMAND SYNTAX

ASCII (expression)

SYNTAX ELEMENTS

The expression may return a data string of any form. The function will then assume that the characters are all members of the EBCDIC character set and translate them using a character map. The original expression is unchanged while the returned result of the function is now the ASCII equivalent.

EXAMPLES

```
READT EbcdicBlock ELSE CRT "Tape failed!"; STOP

AsciiBlock = ASCII (EbcdicBlock) ;* convert to ASCII
```

ASSIGNED

The ASSIGNED function returns a Boolean TRUE or FALSE result depending on whether or not a variable has an assigned value.

COMMAND SYNTAX

ASSIGNED (variable)

SYNTAX ELEMENTS

ASSIGNED returns TRUE if the variable named has an assigned value before the execution of this statement. If the variable has no assigned value then the function returns FALSE.

NOTES

Provision of this function is due to its implementation in older versions of the language. You are advised to program in such a way, to avoid using this statement.
See also: UNASSIGNED.

EXAMPLES

```
IF ASSIGNED (Var1) THEN

    CRT "Var1 has been assigned a value"

END
```

BITAND

Use the BITAND function to perform the bitwise AND comparison of two integers specified by numeric expressions.

SYNTAX

BITAND (expression1, expression2)

DESCRIPTION

The bitwise AND operation compares two integers bit by bit. It returns a bit of 1 if both bits are 1; else, it returns a bit of 0.

If either expression1 or expression2 evaluates to the null value, null is returned.

Non integer values are truncated before the operation is performed.

The BITAND operation is performed on a 32-bit twos-complement word.

NOTE: Differences in hardware architecture can make the use of the high-order bit non portable.

EXAMPLE

PRINT BITAND(6,12)

* The binary value of 6 = 0110

* The binary value of 12 = 1100

This results in 0100, and the following output is displayed:

4

BITCHANGE

BITCHANGE toggles the state of a specified bit in the local bit table, and returns the original value of the bit.

COMMAND SYNTAX

BITCHANGE (table_no)

SYNTAX ELEMENTS

table_no specifies the position in the table of the bit to be changed.

NOTES

For each process, it maintains a unique table of 128 bits (numbered 1 to 128) and treats each bit in the table as a two-state flag - the value returned will always be zero or one.
BITCHANGE returns the value of the bit before it was changed. You can therefore check and set (or reset) a flag in one step.
BITCHANGE also provides some special functions if you use one of the following table_no values:

-1 toggles (enables/disables) the BREAK key Inhibit bit.
-2 toggles (enables/disables) the Command Level Restart feature.
-3 toggles (enables/disables) the Break/End Restart feature.

EXAMPLE

```
OLD.VAL = BITCHANGE (100)

CRT OLD.VAL
```

If bit 100 in the table is zero, it sets to one and displays zero; the reverse will apply if set to one..

BITCHECK

BITCHECK returns the current value of a specified bit from the local bit table.

COMMAND SYNTAX

BITCHECK (table_no)

SYNTAX ELEMENTS

table_no specifies the position in the table of the bit for checking.

NOTES

For each process, it maintains a unique table of 128 bits (numbered 1 to 128) and treats each bit in the table as a two-state flag - the value returned will always be zero or one.
BITCHECK also provides some special functions if you use one of the following table_no values:

-1	returns the setting of the BREAK key Inhibit bit
-2	returns the setting of the Command Level Restart feature
-3	returns the setting of the Break/End Restart feature

EXAMPLE

```
BIT.VAL = BITCHANGE (100)

CRT BIT.VAL
```

If bit 100 in the table is zero, it displays zero; if set to one, it displays one.

BITLOAD

BITLOAD assigns all values in the local bit table, or retrieves all the values.

COMMAND SYNTAX

BITLOAD({bit-string})

SYNTAX ELEMENTS

bit-string is an ASCII string of characters, which represent a hexadecimal value. It is interpreted as a bit pattern and used to assign values to the table from left to right. Assignment stops at the end of the string or when a non-hexadecimal character is found.
If the string represents less than 128 bits, the remaining bits in the table are reset to 0 (zero).
If bit-string is omitted or evaluates to null, an ASCII hex character string is returned, which defines the value of the table. Trailing zeroes in the string are truncated.

NOTES

A unique table of 128 bits (numbered 1 to 128) is maintained for each process. Each bit in the table is treated as a two-state flag - the value will always be 0 (zero) or 1.

EXAMPLE 1

```
NEW.VALUE = "0123456789ABCDEF"
OLD.VALUE = BITLOAD(X)

Loads the bit table with the value of ASCII hex string NEW.VALUE
After assignment, the contents of the bit table is:

0000 0001 0010 0011
0100 0101 0110 0111
1000 1001 1010 1011
1100 1101 1110 1111
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
0000 0000 0000 0000
```

NOTE: that all values beyond the 64th bit have been reset to 0 (zero).

EXAMPLE 2

```
TABLE.VALUE = BITLOAD()
```

Loads variable TABLE.VALUE with the hexadecimal values of the bit table

BITNOT

Use the BITNOT function to return the bitwise negation of an integer specified by any numeric expression.

COMMAND SYNTAX

BITNOT (expression □,bit#□)

DESCRIPTION

bit# is an expression that evaluates to the number of the bit to invert. If bit# is unspecified, BITNOT inverts each bit. It changes each bit of 1 to a bit of 0 and each
bit of 0 to a bit of 1. This is equivalent to returning a value equal to the following: ~(expression~)1
If expression evaluates to the null value, null is returned. If bit# evaluates to the null value, the BITNOT function fails and the program terminates with a run-time error message.
Non integer values are truncated before the operation is performed.
The BITNOT operation is performed on a 32-bit twos-complement word.

NOTE: Differences in hardware architecture can make the use of the high-order bit non portable.

EXAMPLE

PRINT BITNOT(6),BITNOT(15,0),BITNOT(15,1),BITNOT(15,2)
This is the program output:
~7 14 13 11

BITOR

Use the BITOR function to perform the bitwise OR comparison of two integers specified by numeric expressions.

COMMAND SYNTAX

BITOR (expression1, expression2)

DESCRIPTION

The bitwise OR operation compares two integers bit by bit. It returns the bit 1 if the bit in either or both numbers is 1; else, it returns the bit 0.

If either expression1 or expression2 evaluates to the null value, null is returned.

Non integer values are truncated before the operation is performed.

The BITOR operation is performed on a 32-bit twos-complement word.

NOTE: Differences in hardware architecture can make the use of the high-order bit non portable.

EXAMPLE

PRINT BITOR(6,12)

* Binary value of 6 = 0110

* Binary value of 12 = 1100

This results in 1110, and the following output is displayed:

14

BITRESET

BITRESET resets the value of a specified bit in the local bit table to zero and returns the previous value of the bit.

COMMAND SYNTAX

BITRESET (table_no)

SYNTAX ELEMENTS

table_no specifies the position in the table of the bit for reset. If table_no evaluates to zero, it resets all elements in the table to zero and returns the value zero.

NOTES

For each process, it maintains a unique table of 128 bits (numbered 1 to 128) and treats each bit in the table as a two-state flag - the value returned will always be zero or one.
BITRESET returns the previous value of the bit – you can reset and check a flag in one step.
BITRESET also provides some special functions if you use one of the following table_no values:

-1 resets the BREAK key Inhibit bit
-2 resets the Command Level Restart feature
-3 resets the Break/End Restart feature

See also: BITSET.

EXAMPLE

```
OLD.VALUE = BITRESET (112)

PRINT OLD.VALUE
```

If table entry 112 is one, it returns a value of one, resets bit 112 to 0, and prints one. If table entry 112 is zero, returns a value of 0, and prints 0.

BITSET

BITSET sets the value of a specified bit in the bit table to one and returns the value of the bit before it was changed.

COMMAND SYNTAX

BITSET (table_no)

SYNTAX ELEMENTS

table_no specifies the bit to be SET. If table_no evaluates to zero, it sets all elements in the table to one and the returned value is one.

NOTES

For each purpose, it maintains a unique table of 128 bits (numbered 1 to 128) and treats each bit in the table as a two-state flag - the value returned will always be zero or one.
BITSET returns the previous value of the bit - you can check and set a flag in one step.
BITSET also provides some special functions if you use one of the following table_no values:

-1	sets the BREAK key Inhibit bit
-2	sets the Command Level Restart feature
-3	sets the Break/End Restart feature

See also: BITRESET.

EXAMPLE

```
OLD.VALUE = BITSET (112)

PRINT OLD.VALUE
```

If table entry 112 is zero, returns a value of zero, sets bit 112 to one, and prints zero. If table entry 112 is one, returns a value of one, and prints one.

BITTEST

Use the BITTEST function to test the bit number of the integer specified by expression.

COMMAND SYNTAX

BITTEST (expression, bit#)

DESCRIPTION

The function returns 1 if the bit is set; it returns 0 if it is not; Bits are counted from right to left. The number of the rightmost bit is 0.

If expression evaluates to the null value, null is returned. If bit# evaluates to null, the BITTEST function fails and the program terminates with a run-time error message.

Non integer values are truncated before the operation is performed.

EXAMPLE

PRINT BITTEST(11,0),BITTEST(11,1),BITTEST(11,2),BITTEST(11,3)

* The binary value of 11 = 1011

This is the program output:

1 1 0 1

BITXOR

Use the BITXOR function to perform the bitwise XOR comparison of two integers specified by numeric expressions. The bitwise XOR operation compares two integers bit by bit. It returns a bit 1 if only one of the two bits is 1; else, it returns a bit 0.

COMMAND SYNTAX

BITXOR (expression1, expression2)

DESCRIPTION

If either expression1 or expression2 evaluates to the null value, null is returned.

Non integer values are truncated before the operation is performed.

The BITXOR operation is performed on a 32-bit twos-complement word.

NOTE: Differences in hardware architecture can make the use of the high-order bit nonportable.

EXAMPLE

PRINT BITXOR(6,12)

* Binary value of 6 = 0110

* Binary value of 12 = 1100

This results in 1010, and the following output is displayed:

10

BREAK

Allows configuration of the BREAK statement

COMMAND SYNTAX

BREAK / BREAK ON / BREAK OFF / BREAK expression

SYNTAX ELEMENTS

When used with an expression or the keywords ON or OFF the BREAK statement enables or disables the BREAK key for the current process. In UNIX terms, the BREAK key is known more commonly as the interrupt sequence intr defined by the stty command.
Used as a standalone statement, BREAK will terminate the currently executing loop. The EXIT statement is functionally equivalent to the BREAK statement used without arguments.

NOTES

The use of BREAK is to terminate the innermost loop, which it ignores if used outside a loop construct. The compiler will issue warning message 44, and ignore the statement.

EXAMPLES

```
LOOP
    READNEXT KEY FROM LIST1 ELSE BREAK

......
REPEAT
* Program resumes here after BREAK
```

BYTELEN

The BYTELEN function will return the length of the expression as the number of bytes rather than the number of characters.

COMMAND SYNTAX

BYTELEN (expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The BYTELEN function will then return the byte count of the expression.

NOTES

The BYTELEN function will always return the actual byte count for the expression; irrespective of the International Mode in operation at the time. This compares with the LEN function, which will return a character count. The character count may differ from the byte count when processing in International Mode.

CALL

The CALL statement transfers program execution to an external subroutine.

COMMAND SYNTAX

CALL {@}subroutine.name {(argument {, argument ... })}

SYNTAX ELEMENTS

The CALL statement transfers program execution to the subroutine called subroutine.name, which can be any valid string either quoted or unquoted. The CALL @ variant of this statement assumes that subroutine.name is a variable that contains the name of the subroutine to call.
The CALL statement may optionally pass a number of parameters to the target subroutine. These parameters can consist of any valid expression or variable name. If a variable name is used then the called program may return a value to the variable by changing the value of the equivalent variable in its own parameter list.

NOTES

When using an expression to pass a parameter to the subroutine, you cannot use the built-in functions of jBASE BASIC (such as COUNT), within the expression.
An unlimited number of parameters can be passed to an external subroutine. The number of parameters in the CALL statement must match exactly the number expected in the SUBROUTINE statement declaring the external subroutine.
It is not required that the calling program and the external subroutine be compiled with the same PRECISION. However, any changes to precision in a subroutine will not persist when control returns to the calling program.
Variables passed, as parameters to the subroutine may not reside in any COMMON areas declared in the program.

EXAMPLES

```
CALL MySub
```

```
SUBROUTINEMySub
CALL Hello("World")
```

```
SUBROUTINE Hello (Message)
```

```
CALL Complex(i, j, k)
```

```
SUBROUTINE Complex(ComplexA, ComplexB, ComplexC)
```

CALLC

The CALLC command transfers program control to an external function (c.sub.name).
The second form of the syntax calls a function whose name is stored in a jBASE BASIC variable (@var). The program could pass back return values in variables. CALLC arguments can be simple variables or complex expressions, but not arrays. Use CALLC as a command or function.

COMMAND SYNTAX

CALLC c.sub.name [(argument1[,argument2]...)]
CALLC @var [(argument1[,argument2]...)]

Calling a C Program in jBASE

You must link the C program to jBASE before calling it from a BASIC program. Perform the following procedure to prepare jBASE for CALLC:

- Write and compile the C program.
- Define the C program call interface
- Build the runtime version of jBASE (containing the linked C program).
- Write, compile, and execute the Basic program

Calling a Function in Windows NT

The CALLC implementation in jBASE for Windows NT or Windows 2000 uses the Microsoft Windows Dynamic Link Library (DLL) facility. This facility allows separate pieces of code to call one another without permanently binding together. Linking between the separate pieces occurs at runtime (rather than compile time) through a DLL interface.
For CALLC, developers create a DLL and then call that DLL from jBASE.

EXAMPLES

In the following example, the called subroutine draws a circle with its center at the twelfth row and twelfth column and a radius of 3:

```
RADIUS = 3

CENTER = "12,12"

CALLC DRAW.CIRCLE(RADIUS,CENTER)
```

In the next example, the subroutine name is stored in the variable SUB.NAME, and is indirectly called:

```
SUB.NAME = DRAW.CIRCLE

CALLC @SUB.NAME(RADIUS,CENTER)
```

The next example uses, CALLC as a function, assigning the return value of the subroutine

PROGRAM.STATUS in the variable RESULT:

```
RESULT = CALLC PROGRAM.STATUS
```

CALLdotNET

The CALLdotNET command allows BASIC to call any .NET assembly and is useful when using third party applications.

COMMAND SYNTAX

CALLdotNET NameSpaceAndClassName, methodName, param SETTING ret [ON ERROR errStatment]

In order to use CALLdotNET, you need:

The .NET Framework
The dotNETWrapper.dll installed somewhere to where your PATH points.

NOTE:

The dotNETWrapper is loaded dynamically at runtime; therefore, a compiled basic application has no dependencies on the .NET Framework. Loading the framework takes between (~5 –7 sec.). However, this only occurs when calling the .NET method for the first time.

SYNTAX ELEMENTS

NameSpaceAndClassName The “full” NameSpace (e.g., myNameSpace.myClass)

methodName The name of the .NET in this class (e.g., “myMethod”)

Param Any parameter (eg DynArray)

EXAMPLE

In C#:

```
using System;
using System.Windows.Forms;
namespace myNameSpace
{
        public class Class1
        {
                public string sayHello(string str)
                {
                        return “Thank you, I received : “ + str;
                }
                public Class1(){}
        }
```

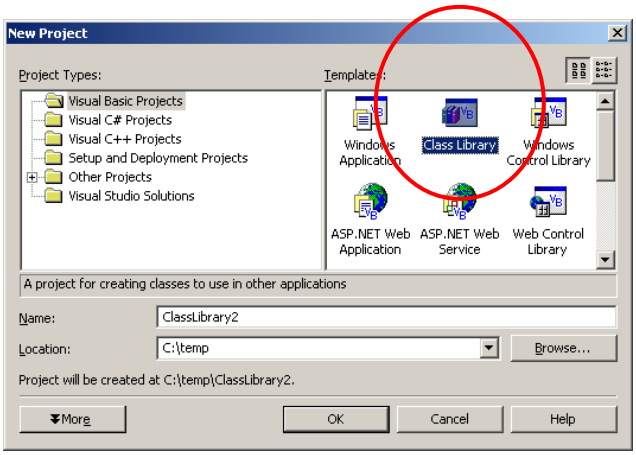
```
}
```

In VB.NET:

```
Namespace myNameSpace
    Public Class Class1
        Public Function sayHello(ByVal str As String) As String
            Dim sAnswer As String
            sAnswer = InputBox(str)
            sayHello = sAnswer
        End Function
    End Class
End Namespace
```

Note: Create the .NET project as a 'Class Library'.

If using the visual studio IDE, this option is on selected when creating a new project:

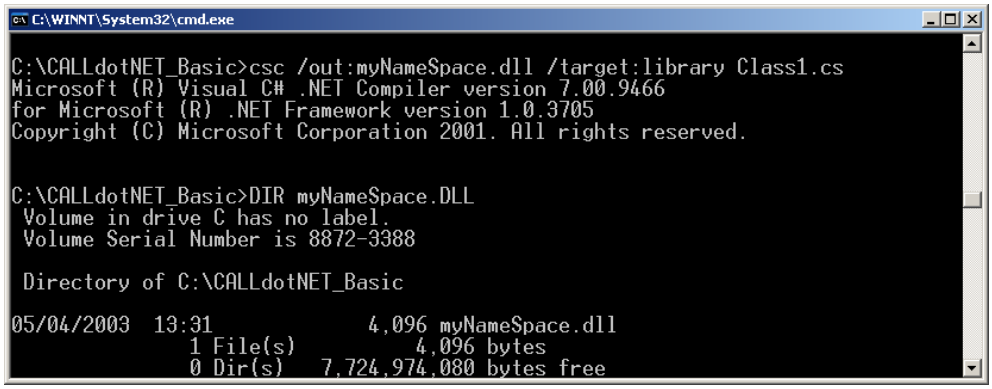


If using .NET SDK (instead of the IDE) to compile class libraries into a 'DLL' file, the 'csc' (C# Compiler) or 'vbc' (Visual Basic .NET compiler) command can be used from the command line:

```
csc /out:myNameSpace.dll /target:library
sourcefile.cs
```

The name of the '.DLL' created must be the same as the 'namespace' as used in the class library t locate the 'dotNetWrapper.dll' library:

After creating the library, place it in the same private directory as the application. (i.e. the same directory as the jBASE BASIC executable that will call the class) This is a requirement of the .NET paradigm and not jBASE. The directory should also be in the PATH environment variable.



To call these methods from Basic:

CALLdotNET "myNameSpace.Class1","mymethod", p SETTING ret

CRT ret

ON ERROR

You can manage any errors, which occur during the call, at the BASIC level by getting the SYSTEM(0) variable.

This variable can have the following values:

1. Not a Windows platform.
2. Cannot load the dotNETWrapper
3. Cannot get assembly
4. Cannot get Class
5. Cannot get Method
6. Cannot Create Instance
7. Unhandled Error in the .NET library

EXAMPLE

BASIC code using the ON ERROR would look like this:

```
PROGRAM testCALLdotNET

	ns.className = ''

	methodName = ''

	param = ''

	CRT "Please enter NameSpace.ClassName : "

	INPUT ns.className

	CRT "Please enter a Method Name : "

	INPUT methodName

	CRT "Please enter a Parameter : "

	INPUT param

	CALLdotNET ns.className, methodName, param SETTING ret ON ERROR
GOSUB errHandler

	CRT "Received back from .NET : " : ret
```

```
	STOP
errHandler:
	err = SYSTEM(0)
	BEGIN CASE
		CASE err = 2
			CRT "Cannot find dotNETWrapper.dll”
		CASE err = 3
			CRT "Class " : className : "doesn't exist !"
		CASE err = 5
			CRT "Method " : methodName : "doesn't exist !"
	END CASE
	RETURN
```

CALLJ

The CALLJ command allows BASIC to call a Java method. CALLJ is useful when using third party applications offering a Java API (for example, publish and subscribe, messaging, etc.)

COMMAND SYNTAX

```
CALLJ packageAndClassName, [$]methodName, param SETTING ret [ON
ERROR] errStatment
```

In order to use CALLJ, you need:

- A Java virtual machine
- CLASSPATH environment variable set to point on the class you want to invoke

NOTES

The Java virtual machine is loaded dynamically at runtime, so a compiled basic application has no dependencies on any Java virtual machine. By default, the program will search for:
jvm.dll on Windows platforms
libjvm.sl on HP UNIX
libjvm.so for other platforms
Although it is not usually necessary, it is possible to specify a Java library by setting the JBCJVMLIB environment variable:

```
set JBCJVMLIB= C:\jdk1.3.1\jre\bin\classic\jvm.dll
```

PERFORMANCE CONSIDERATIONS

The first call to CALLJ carries the overhead of loading the Java Virtual Machine into memory. Susequent calls do not have this overhead and it is recommended that programs are structured in such a way that the Java Virtual Machine is only loaded once.

In addition, calls to non static methods carry the overhead of calling the constructor for the class. Wherever possible, static methods should be used.:

SYNTAX ELEMENTS

packageAndClassName The “full” class name (e.g., com.jbase.util.utilClass)
methodName The name of the Java method in this class (e.g., “myMethod”)
NOTE: If the method is static, you must append a ‘$’ before the name. This ‘$’ will be removed from the method name before calling it.
Param Any parameter (eg DynArray)

EXAMPLE

In Java:

```
package mypackage;

public class mytestclass {

       static int i = 0;

        private mytestclass() {

    }

    public String mymethod(String s){

          return (“Java Received : “ + s) ;

    }

    public static String mystaticmethod(String s){

            i++;

            return s + " "  + i;

    }

}
```

To call these methods from jBASE BASIC:

```
CALLJ "mypackage.mytestclass","mymethod", p SETTING ret

CRT ret

CALLJ "mypackage/mytestclass","$mystaticmethod",p SETTING ret

CRT ret
```

ON ERROR

Use the SYSTEM(0) variable to manage any errors at the BASIC level, which occur during the call.

This variable can have the following values:

1	Fatal error creating thread
2	Cannot create JVM
3	Cannot find class
4	Unicode conversion error
5	Cannot find method
6	Cannot find object constructor

7	Cannot instantiate object

EXAMPLE

jBASE BASIC code using the ON ERROR will look like this:

```
PROGRAM testcallj

	className = ''

	methodName = ''

	param = ''

	CRT "Please enter a Class Name : " INPUT className

	CRT "Please enter a Method Name : " INPUT methodName

	CRT "Please enter a Parameter : " INPUT param

	CALLJ className,methodName, param SETTING ret ON ERROR GOTO
errHandler

	CRT "Received batch from Java : " : ret

RETURN

errHandler:

	err = SYSTEM(0)

	IF err = 2 THEN

		CRT "Cannot find the JVM.dll !"

		RETURN

	END

	IF err = 3 THEN

		CRT "Class " : className : "doesn't exist !"

		RETURN

	END

	IF err = 5 THEN
```

```
            CRT "Method " : methodName : "doesn't exist !"

            RETURN

        END

END
```

The CALLJ function provides access to a JavaVM from within the BASIC environment. For it to be able to start a JavaVM (JVM) the environment needs to know where the JVM is located. Specifically it needs to know where certain libraries are located.

WINDOWS

Windows: looking for 'jvm.dll'

Add "c:\jdk1.3.1_07\jre\bin\server" to the PATH environment variable.
A generic format might be %JDKDIR%\jre\bin\server.

UNIX

For UNIX it is possible to configure generic symbolic links to make profiles portable.

Location of JDK export JDKDIR=/opt/java1.3

Symbolic link for JRE libs /opt/java1.3/jrelib
Symbolic link for JVM library /opt/java1.3/jvmlib

Linux

/opt/java1.3/jrelib -> /opt/java1.3/jre/lib/i386
/opt/java1.3/jvmlib -> /opt/java1.3/jre/lib/i386/server

.profile:

Add “/opt/java1.3/jrelib:/opt/java1.3/jvmlib” to the LD_LIBRARY_PATH

HP-UX

/opt/java1.3/jrelib -> /opt/java1.3/jre/lib/PA_RISC2.0
/opt/java1.3/jvmlib -> /opt/java1.3/jre/lib/PA_RISC2.0/server

.profile:

Add “/opt/java1.3/jrelib:/opt/java1.3/jvmlib” to the SHLIB_PATH

AIX -- (IBM JDK)

/opt/java1.3/jrelib -> /opt/java1.3/jre/bin
/opt/java1.3/jvmlib -> /opt/java1.3/jre/bin/classic

.profile:

Add “/opt/java1.3/jrelib:/opt/java1.3/jvmlib” to the LIBPATH

Solaris

/opt/java1.3/jrelib -> /opt/java1.3/jre/lib/sparc

/opt/java1.3/jvmlib -> /opt/java1.3/jre/lib/sparc/server

.profile:

Add “opt/java1.3/jrelib:/opt/java1.3/jvmlib” to the LD_LIBRARY_PATH

Examples using JVM WITHOUT symbolic links as above:

Linux: looking for 'libjvm.so'

Add 2 directories to LD_LIBRARY_PATH.
/opt/java1.3/jre/lib/i386/server:/opt/java1.3/jre/lib/i386

Solaris: looking for 'libjvm.so'

Add 2 directories to LD_LIBRARY_PATH.
/opt/java1.3/jre/lib/sparc/server:/opt/java1.3/jre/lib/sparc

HP-UX 11: looking for 'libjvm.sl'

Add 2 directories to SHLIB_PATH.
/opt/java1.3/jre/lib/PA_RISC2.0/server:/opt/java1.3/jre/lib/PA_RISC2.0

OPTIONS:

JBCJVMLIB
If the searched for library appears incorrect for your platform, then you can override it by setting the JBCJVMLIB environment variable.

e.g. "export JBCJVMLIB=jvm.shared_lib"

and then CALLJ will try to locate the library 'jvm.shared_lib' at runtime.

JBCJVMPOLICYFILE
You can specify a policy file for the JMV. The policy for a Java application environment (specifying which permissions are available for code from various sources) is represented by a Policy object. More

specifically, it is represented by a `Policy` subclass providing an implementation of the abstract methods in the `Policy` class (which is in the `java.security` package). You can override it by setting the JBCJVMPOLICYFILE environment variable.
The source location for the default policy information is

WINDOWS

```
%JBASERELEASEDIR%\config\policy.all
```

UNIX

```
$JBASERELEASEDIR/config/policy.all
```

e.g. "export JBCJVMPOLICYFILE =/usr/jbase/mypolicy.all"

JBCJVMENCODING

Internally, the Java virtual machine always operates with data in Unicode. However, as data transfers to or from the Java virtual machine, the Java virtual machine converts the data to other encodings. If the you want to change the default encoding of the JVM on your platform, then you can override it by setting the JBCJVMENCODING environment variable.

e.g. "export JBCJVMENCODING = Cp1257"

JBCJVMNOOPTS

Internally, the CALLJ is optimum to start the JVM with options (see below the table). If the you don't want to pass these options for the JVM, then you can override it by setting the JBCJVMNOOPTS environment variable. In this case no more options will be pass to the JVM.

DEFAULT OPTIONS

Win32:	-Xrs
TRUE64:	-Xcheck:jni
Solaris:	-XX:+AllowUserSignalHandlers
Linux:	-Xrs -XX:+AllowUserSignalHandlers
AIX 32 bits:	-Xrs -Xnocatch
AIX 64 bits:	-Xrs -d64
HPUX 32 bits:	
HPUX 64 bits:	-Xrs -XX:+AllowUserSignalHandlers

JBCJVMOPT[1..5]

If the you want to pass some options for the JVM, then you can set by setting the JBCJVMOPT[1..5] environment variable

e.g. "export JBCJVMOPT1=-Xrs "

KNOWN LIMITATIONS

HP-UX

There is a problem with HP-UX due to it's dynamic loader. See man dlopen(3C) for detail of the TLS limitation.

This means that the JVM library must be linked against the calling program, there are no known problems caused by this.

'ldd progname' lists current external library references and we need to add libjvm.

The result looks like this:

JVM: dl_error [Can't dlopen() a library containing Thread Local Storage: libjvm.sl]

If the program is built with the required link as below then it works.

jbc –Jo callj.b -ljvm -L/opt/java1.3/jre/lib/PA_RISC2.0/server

If the CALLJ statement is inside a subroutine, then the program that calls the subroutine must be built as above.

Examples using JVM WITHOUT symbolic links as above:

Linux: searching for 'libjvm.so'

Add 2 directories to LD_LIBRARY_PATH.

```
/opt/java1.3/jre/lib/i386/server:/opt/java1.3/jre/lib/i386
```

Solaris: searching for 'libjvm.so'

Add 2 directories to LD_LIBRARY_PATH.

```
/opt/java1.3/jre/lib/sparc/server:/opt/java1.3/jre/lib/sparc
```

HP-UX 11: searching for 'libjvm.sl'

Add 2 directories to SHLIB_PATH.

```
/opt/java1.3/jre/lib/PA_RISC2.0/server:/opt/java1.3/jre/lib/PA_RISC2.
0
```

OPTIONS

JBCJVMLIB

If the search for the library appears incorrect for your platform, then you can override it by setting the **JBCJVMLIB** environment variable.

e.g. "`export JBCJVMLIB=jvm.shared_lib`"

and then CALLJ will try to locate the library 'jvm.shared_lib' at runtime.

JBCJVMPOLICYFILE

You can specify a policy file for the JMV. The policy for a Java application environment (specifying which permissions are available for code from various sources) is represented by a Policy object. More specifically, it is represented by a Policy subclass providing an implementation of the abstract methods in the Policy class (which is in the java.security package). You can override it by setting the JBCJVMPOLICYFILE environment variable.
The source location for the default policy information is:

WINDOWS

```
%JBASERELEASEDIR%\config\policy.all
```

UNIX

```
$JBASERELEASEDIR/config/policy.all
```

e.g. "export JBCJVMPOLICYFILE =/usr/jbase/mypolicy.all"

JBCJVMENCODING

Internally, the Java virtual machine always operates with data in Unicode. However, as data transfers to or from the Java virtual machine, the Java virtual machine converts the data to other encodings. If the you want to change the default encoding of the JVM on your platform, then you can override it by setting the JBCJVMENCODING environment variable.

```
e.g.  "export JBCJVMENCODING = Cp1257"
```

JBCJVMNOOPTS

Internally, CALLJ is optimized to start the JVM with options (see the table below). If you don't want to pass these options for the JVM, then you can override it by setting the **JBCJVMNOOPTS** environment variable. In this case no more options will be passed to the JVM.

DEFAULT OPTIONS :

Win32:	-Xrs
TRUE64:	-Xcheck:jni
Solaris:	-XX:+AllowUserSignalHandlers
Linux:	-Xrs -XX:+AllowUserSignalHandlers
AIX 32 bits:	-Xrs -Xnocatch
AIX 64 bits:	-Xrs -d64
HPUX 32 bits:	
HPUX 64 bits:	-Xrs -XX:+AllowUserSignalHandlers

JBCJVMOPT[1..5]

If the you want to pass some options for the JVM, then set the **JBCJVMOPT[1..5]** environment variable

```
e.g.  "export JBCJVMOPT1=-Xrs "
```

KNOWN LIMITATIONS

HP-UX

There is a problem with HP-UX due to its dynamic loader. See man dlopen(3C) for detail of the TLS limitation.

This means that the JVM library must be linked against the calling program, there are no known problems caused by this.

'ldd progname' lists current external library references and we need to add libjvm.

The symptom looks like this:

JVM: dl_error [Can't dlopen() a library containing Thread Local Storage: libjvm.sl]

If the program is built with the required link as below then it works.

```
jbc –Jo callj.b -ljvm -L/opt/java1.3/jre/lib/PA_RISC2.0/server
```

If the CALLJ statement is inside a subroutine, then the program that calls the subroutine must be built as above.

CALLONEXIT

The CALLONEXIT function call allows you to specify the name of a SUBROUTINE to call when the program terminates.

COMMAND SYNTAX

```
rc = CALLONEXIT("ErrorExit")
```

The subroutine definition would look like this

```
SUBROUTINE CALLONEXIT(parm1)
```

You can add parameters to the error subroutine by adding multi-values to the parameter to CALLONEXIT, which are passed to the called subroutine in the first parameter.

If you execute CALLONEXIT multiple times with the same subroutine name, it discards other calls.
If you execute CALLONEXIT multiple times with a different subroutine name, then upon exit multiple subroutines will be called in the order that CALLONEXIT was called.

EXAMPLES

For example, consider the simple programs below. The program enters the debugger. If at this point the login session terminates for any reason (the line drops, the program is killed, the user enters 'off' at the debugger prompt) , the two specified subroutines (ErrorExit and EndProgram) will still be called just as they would if the program were allowed to terminate normally.

```
PROGRAM PROG1

rc = CALLONEXIT("ErrorExit")

EXECUTE "PROG2"


PROGRAM PROG2

rc = CALLONEXIT("EndProgram")

DEBUG
```

All efforts are made to call the subroutine under all circumstances. However, if a SIGKILL (signal 9) terminates the program, which cannot be trapped, it does not call the subroutine. This is a feature of operating systems, not a limitation. In addition, if the program terminates due to say a memory error, then calling the subroutines depends upon how badly the memory error has corrupted the memory.

CASE

The CASE statement allows the programmer to execute a particular sequence of instructions based upon the results of a series of test expressions.

COMMAND SYNTAX

BEGIN CASE
CASE expression statement(s)
CASE expression
statement(s)
. . .
END CASE

SYNTAX ELEMENTS

The BEGIN CASE and END CASE statements bound the CASE structure. Within this block, an arbitrary number of CASE expression statements may exist followed by any number of jBASE BASIC statements. The expression should evaluate to a TRUE or FALSE result. The evaluation of each expression at execution time is in order. If the expression returns a TRUE result, it then executes the statements below. On completion of the associated statements, execution will resume at the first statement following the END CASE.

NOTES: A default action (to trap error conditions for instance) may be introduced by using an expression that is always TRUE, such as CASE one. This should always be the last expression in the CASE block.

EXAMPLE

```
BEGIN CASE

CASE A = 1

    CRT "You won!"

CASE 1

    CRT "You came nowhere"

END CASE
```

A single comment is printed depending on the value of A.

NOTE: that if A is not 1 then the default CASE 1 rule will be executed as a "catch all".

CATALOG Command

Cataloging and Running your Programs

Use the CATALOG command to create UNIX executables and shared libraries from the application source code. Once you have cataloged your programs, you can run them like any other command on the system.

The RUN command which is sometimes used to execute compiled jBASE BASIC programs without cataloging them can still be used but is really only maintained for compatibility. Whenever possible, you should catalog your programs rather than RUN them.

The CATALOG command should be executed from the application directory rather than using link names and the application id should be used. The reasons for executing the CATALOG command from the application directory and application id are that the .profile script will have set up the required environment variables correctly and that the correct file permission will be used when creating and deleting UNIX executables and directories.

The format of the CATALOG command is as follows.

```
CATALOG SourceFilename Itemlist
```

When first invoked the CATALOG command will create a $HOME/bin directory into which the UNIX executables will be placed. A $HOME/lib directory will also be created into which any subroutines will be placed. The lib directory contains a jLibDefinition file, which describes how to build the subroutines into shared libraries. The entries in the jLibDefinition file are described below:

libname naming convention for shared object files.
exportname export list of shared objects. Used as cross reference to find subroutine functions.
maxsize maximum size of a shared object library before creating another.

When the maximum size of a shared library object is reached then a new shared library object will be created by the CATALOG command. The new shared library objects are named according to the definition of libname and are numbered sequentially. For example:

libname=lib%a%n.so
where
%a = account or directory name
%n = number in sequence.

If subroutines were cataloged in the user account name, fred then the shared object libraries produced would be named, libfred0.so libfred1.so libfred2.so and so on.

Note: To guard against libraries being cataloged incorrectly, perhaps under the wrong user account name, the definition of libname should be changed to libfred%n.so. This will ensure that any shared objects are created using the proper user account name.

The shared library objects, .so files, contain the UNIX executables for subroutine source code. The shared library objects are linked at runtime by the jBASE call function, which utilises the dynamic linker programming interface. The dynamic linker will link shared libraries at the start of program execution time, or when requested by the jBASE call function. For example, each executable created using the jBASE compiler will be linked with the jBASE jEDI library functions, libjedi.so, at compilation time. This shared library enables database record retrieval and update and will be loaded into memory by the dynamic linker when an application executable starts execution. However the shared library containing any subroutines required by the executing program will only be loaded into memory when initially requested by the subroutine call. Only one copy of any shared library is required in memory at any time, thus reducing program memory requirements.
The $HOME/lib directory also contains a directory where all the subroutine objects, .o files, are held. These are required for making the shared library, .so files.
The $HOME/lib directory also contains an export list, .el file, built by the CATALOG command, which is used as a cross reference when dynamically linking shared objects at run time.
The main application program executables are placed into the $HOME/bin directory.
To enable the application executables to be found the $HOME/bin path should be added to the PATH environment variable.
To enable the executing application to call the correct application subroutines the JBCOBJECTLIST or LD_LIBRARY_PATH environment variable should be assigned to the application shared library path, $HOME/lib. If the main application program or any subroutine programs make calls to subroutines in other directories then the path of the shared library directories should also be added to the JBCOBJECTLIST or LD_LIBRARY_PATH environment variable.
It is recommended that executables or subroutines of the same name are not available from different directories. This can make application execution very confusing and is reliant on assigning the lib or bin directories to the environment variable in the correct sequence. The assignment of the environment variables should be included and exported in the .profile script file.
Executables and shared library objects can be removed from the bin and lib directories by using the DECATALOG command.

CATS

The CATS function concatenates the corresponding elements in two dynamic arrays.

COMMAND SYNTAX

CATS (DynArr1, DynArr2)

SYNTAX ELEMENTS

DynArr1 and DynArr2 represent dynamic arrays.

NOTES

If one dynamic array supplied to the CATS function is null then the result of the CATS function is the non-null dynamic array.

EXAMPLES

```
X  = "a" : @VM : "b" : @VM : "c"

B = 1 : @VM : 2 : @VM : 3

Z = CATS(X, Y)
```

The assigned value to variable Z is:

```
a1 : @VM : b2 : @VM : c3

A = "a" : @SVM : "b" : @VM : "c": @VM : "d"

B = "x" : @VM : "y" : @SVM : "z"

C = CATS(A, B)
```

The assigned value to variable C is:

```
ax : @SVM : b : @VM : cy : @SVM : z : @VM : d
```

CHAIN

The CHAIN statement exits the current program and transfers process control to the program defined by the expression. Process control will never return to the originating program.

COMMAND SYNTAX

CHAIN expression

SYNTAX ELEMENTS

The expression should evaluate to a valid UNIX or Windows command (this may be another jBASE BASIC program). The command string may be suffixed with the (I option, which will cause any COMMON variables in the current program to be inherited by the new program (providing it is a jBASE BASIC program).

NOTES

There are no restrictions to the CHAIN statement and you may CHAIN from anywhere to anywhere. However, it is advisable that your program follows a logical path easily seen by another programmer. If the program, which contains the CHAIN command (the current program) was called from a JCL program, and the program to be executed (the target program) is another jBASE BASIC program, control will return to the original JCL program when the target program terminates. If the target program is a JCL program, control will return to the command shell when the JCL program terminates.

EXAMPLES

```
CHAIN "OFF" ;* exit via the OFF command

! Prog1

COMMON A,B

A = 50; B = 100

CHAIN "NEWPROG (I"

! NEWPROG

COMMON I,J

! I and J inherited

CRT I,J
```

CHANGE

The CHANGE statement operates on a variable and replaces all occurrences of one string with another.

COMMAND SYNTAX

CHANGE expression1 TO expression2 IN variable

SYNTAX ELEMENTS

expression1 - may evaluate to any result and is the string of characters that will be replaced.
expression2 - may also evaluate to any result and is the string of characters that will replace
expression1 - The variable may be any previously assigned variable in the program.

NOTES

There is no requirement that strings be of the same length. The jBASE BASIC language also supports the CHANGE function for compatibility with older systems.

EXAMPLES

```
String1 = "Jim"

String2 = "James"

Variable = "Pick up the tab Jim"

CHANGE String1 TO String2 IN Variable

CHANGE "tab" TO "check" IN Variable
```

CHANGETIMESTAMP

Use CHANGETIMESTAMP to adjust existing timestamp to return new timestamp value.

COMMAND SYNTAX

CHANGETIMESTAMP (Timestamp, Array)

SYNTAX ELEMENTS

The CHANGETIMESTAMP function generates a new timestamp by adjusting an existing timestamp value using the elements specified in the dynamic array.

The format of the adjustment array is as follows:
Years^Months^Weeks^Days^Hours^Minutes^Seconds^Milliseconds

CHAR

The CHAR function returns the ASCII character specified by the expression.

COMMAND SYNTAX

CHAR (expression)

SYNTAX ELEMENTS

The expression must evaluate to a numeric argument in the range 0-255, which is the entire ASCII character set.

INTERNATIONAL MODE

The CHAR function will return Unicode values encoded as UTF-8 byte sequences as follows:
Expression values 0 – 127 return UTF-8 single byte characters equivalent to ASCII.
Expression values 127 – 248 return UTF-8 double byte character sequences.
Expression values 249 – 255 return system delimiters 0xf8 – 0xff
Expression values > 255 return UTF-8 multi byte character sequences

When system delimiter values are not specifically required, generate UTF-8 byte sequences using the UTF8 function. i.e. X = UTF8(@AM) will generate a UTF-8 byte sequence in variable X for the system delimiter equating to Unicode value 0x000000fe.

NOTES

jBASE BASIC variables can contain any of the ASCII characters 0-255, thus there are no restrictions on this function.
Use this function to insert field delimiters within a variable or string; these are commonly equated to AM, VM, SV in a program.
See also: CHARS

EXAMPLES

```
EQUATE AM TO CHAR (254) ;* field Mark

EQUATE VM TO CHAR(253) ;* value Mark

EQUATE SV TO CHAR(252) ;* sub Value mark

CRT CHAR (7): ;* ring the bell
```

CHARS

The CHARS function accepts a dynamic array of numeric expressions and returns a dynamic array of the corresponding ASCII characters.

COMMAND SYNTAX

CHARS (DynArr)

SYNTAX ELEMENTS

Each element of DynArr must evaluate to a numeric argument in the range 0-255.

NOTES

If any of the dynamic array elements are non-numeric, a run-time error will occur.

See also: CHAR ().

EXAMPLE

```
y = 58 : @AM : 45 : @AM : 41

z = CHARS (y)

FOR i = 1 TO 3

      CRT z<i>:

NEXT i
```

This code displays: :-)

CHDIR

The CHDIR function allows the current working directory, as seen by the process environment, to be changed.

COMMAND SYNTAX

CHDIR (expression)

SYNTAX ELEMENTS

The expression should evaluate to a valid path name within the file system. The function returns a Boolean TRUE result if the CHDIR succeeded and a Boolean FALSE result if it failed.

EXAMPLES

```
IF CHDIR ("/usr/jBASIC/src") THEN

    CRT "jBASE development system INSTALLED"

END

IF GETENV("JBASICGLOBALDIR", jgdir) THEN

    IF CHDIR (jgdir:"\config") ELSE

        CRT "jBASE configuration cannot be found."

        ABORT
    END
END
```

CHECKSUM

The CHECKSUM function returns a simple numeric checksum of a character string.

COMMAND SYNTAX

CHECKSUM(expression)

SYNTAX ELEMENTS

The expression may evaluate to any result but will usually be a string. The function then scans every character in the string and returns a numeric addition of the characters within the string.

NOTES

The function calculates the checksum by summing the product of the ASCII value of each character and its position within the string.

EXAMPLES

```
INPUT DataBlock,128:

IF CHECKSUM(DataBlock) = ExpectedChk THEN

    CRT AckChar:

END

ELSE

......
```

CLEAR

The CLEAR statement will initialize all the variables to numeric 0.

COMMAND SYNTAX

CLEAR

NOTES

Use CLEAR at any time during the execution of the program.

EXAMPLES

```
Var1 = 99

Var2 = 50

CLEAR
```

CLEARCOMMON

The CLEARCOMMON statement initializes all unnamed common variables to a value of zero.

COMMAND SYNTAX

CLEARCOMMON

SYNTAX ELEMENTS

None

CLEARDATA

The CLEARDATA statement clears data stacked by the DATA statement.

COMMAND SYNTAX

CLEARDATA

SYNTAX ELEMENTS

None

CLEARFILE

Use the CLEARFILE statement to clear all the data from a file previously opened with the OPEN statement.

COMMAND SYNTAX

CLEARFILE {variable} {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

The variable should be the subject of an OPEN statement before the execution of CLEARFILE upon it. If the variable is omitted from the CLEARFILE statement, it assumes the default file variable as per the OPEN statement.

NOTES

The CLEARFILE statement will remove every database record on the file against which it is executed, therefore, use with caution.
If the variable argument does not describe a previously opened file, the program will enter the debugger with an appropriate message.
If the SETTING clause is specified and the CLEARFILE fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

EXAMPLES

```
OPEN "DATAFILE" ELSE ABORT 201, "DATAFILE"

OPEN "PROGFILE" TO FILEVAR ELSE ABORT 201, "PROGFILE"

CLEARFILE
CLEARFILE FILEVAR
```

CLEARINPUT

The CLEARINPUT command clears the terminal type-ahead buffer to allow the next INPUT statement to force a response from the user.

COMMAND SYNTAX

CLEARINPUT

EXAMPLE

In the following example, the CLEARINPUT statement clears the terminal type-ahead buffer to provoke a response from the user to the prompt:

```
CLEARINPUT
PRINT "DO YOU WANT TO DELETE THIS FILE?(Y OR N)"; INPUT X,1
```

NOTE: The CLEARINPUT command is synonymous with INPUTCLEAR.

CLEARSELECT

Use the CLEARSELECT statement to clear active select lists.

COMMAND SYNTAX

CLEARSELECT {ListName | ListNumber}

SYNTAX ELEMENTS

ListName must evaluate to a jBASE BASIC list variable. ListNumber is one of the numbered lists in the range 0 to 11. If neither ListName nor ListNumber are specified then it clears the default list (0).

EXAMPLE

```
A = "good" : @AM : "bad" : @AM : "ugly"

B = "night" : @AM : "day"

SELECT A TO 3

SELECT B TO blist

adone = 0; bdone = 0


LOOP
 READNEXT Ael FROM 3 ELSE adone = 1

   READNEXT Bel FROM blist ELSE bdone = 1

UNTIL adone AND bdone DO

   CRT Ael, Bel

   CLEARSELECT 3

   CLEARSELECT blist

REPEAT
```

This program displays: good night

CLOSE

Use the CLOSE statement to CLOSE an opened file, which is no longer required

COMMAND SYNTAX

CLOSE variable{, variable ...}

SYNTAX ELEMENTS

The variable list should contain a list of previously opened file variables that are no longer needed. The variables will be cleared and may be reused as ordinary variables.

NOTES: You can open an unlimited amount of files within jBASE BASIC; however leaving them open consumes valuable system resources.
Use good practice to hold open only those file descriptors to which you have constant access.

EXAMPLES

```
OPEN "DATAFILE" TO FILEVAR ELSE ABORT 201, "DATAFILE"
.....
CLOSE FILEVAR
```

CLOSESEQ

CLOSESEQ closes the file previously opened for sequential access.

COMMAND SYNTAX

CLOSESEQ FileVar

SYNTAX ELEMENTS

FileVar contains the file descriptor of the previously opened sequential file

COL1 and COL2

Use these functions in conjunction with the FIELD function to determine the character positions 1 position before and 1 position after the location of the last field.

COMMAND SYNTAX

COL1() / COL2()

NOTES

When a field has been located in a string, it is sometimes useful to know its exact position within the string to manipulate either it, or the rest of the string. COL1() will return the position of the character immediately before the last field located. COL2() will return the position of the character immediately after the end of the last field located. Use them to manipulate the string.

EXAMPLES

```
A = "A,B,C,D,E"

Fld = FIELD(A, ",", 2)

CRT COL1()

CRT COL2()
```

Displays the values 2 and 4

COLLECTDATA

Use the COLLECTDATA statement to retrieve data passed from the PASSDATA clause of an EXECUTE statement.

COMMAND SYNTAX

COLLECTDATAvariable

SYNTAX ELEMENTS

variable is the name of the variable, which is to store the retrieved data.

NOTES

Use the COLLECTDATA statement in any program, which is EXECUTEd (or PERFORMed) by another program where the calling program uses a PASSDATA clause. The EXECUTEd program uses a COLLECTDATA statement to retrieve the passed data.
If a PASSDATA clause is not in effect, variable will be assigned a value of null.

EXAMPLE

```
FIRST
001 EXECUTE "RUN JBASIC_PROGS SECOND" PASSDATA "Handover"

SECOND

001 COLLECTDATA PassedMessage

002 CRT PassedMessage
```

In the above example, program FIRST will EXECUTE program SECOND and will pass the string "Handover" in the PASSDATA clause. Program SECOND retrieves the string to a variable PassedMessage and prints the string on the Terminal screen.

COMMON

The COMMON statement declares a list of variables and matrices that can be shared among various programs. There can be many common areas including a default, unnamed common area.

COMMAND SYNTAX

COMMON {/CommonName/} variable{, variable ... }

SYNTAX ELEMENTS

The list of variables should not have been declared or referenced previously in the program file. The compiler will detect any bad declarations and display suitable warning or error messages. If the common area declared with the statement is to be named then the first entry in the list should be a string, delimited by the / character.

NOTES

The compiler will not, by default, check that variables declared in COMMON statements are initialized before they have been used as this may be beyond the scope of this single source code check. The -JCi option, when specified to the jBASE BASIC compiler, will force this check to be applied to common variables as well. The initialization of named common is controlled in the Config_EMULATE file.
Variables declared without naming the common area may only be shared between the program and its subroutines (unless CHAIN is used). Variables declared in a named common area may be shared across program boundaries. When any common area is shared, all programs using it should have declared the same number of variables within it.
Dimensioned arrays are declared and dimensioned within the COMMON statement.

EXAMPLES

```
COMMON A, B(2, 6, 10), c

COMMON/Common1/ A, D, Array(10, 10)
```

COMPARE

The COMPARE function compares two strings and returns a value indicating whether or not they are equal.

COMMAND SYNTAX

COMPARE(expression1, expression2{, justification})

SYNTAX ELEMENTS

expression1 is the first string for comparison

expression2 is the second string for comparison

justification specifies how the strings are to be compared. "L" indicates a left justified comparison. "R" indicates a right justified comparison. The default is left justification.

The function returns one of the following values:

-1	The first string is less than the second
0	The strings are equal
1	The first string is greater than the second

EXAMPLE

```
A = "XY999"

B = "XY1000"

R1 = COMPARE(A,B,"L")

R2 = COMPARE(A,B,"R")

CRT R1,R2
```

The code above displays 1 -1, which indicates that XY999 is greater than XY1000 in a left justified comparison and XY999 is less than XY1000 in a right justified comparison.

INTERNATIONAL MODE

When using the COMPARE function in International Mode, the function will use the currently configured locale to determine the rules by which each string is considered less than or greater than the other will.

CONTINUE

The CONTINUE statement is the complimentary statement to the BREAK statement without arguments.

COMMAND SYNTAX

Use the statement within a loop to skip the remaining code in the current iteration and proceed directly on to the next iteration.

NOTES

See also: BREAK, EXIT

The compiler will issue a warning message and ignore the statement if it is found outside an iterative loop such as FOR...NEXT, LOOP...REPEAT.

EXAMPLES

```
FOR I = 1 TO 30

    IF Pattern(I) MATCHES "0N" THEN CONTINUE

    GOSUB ProcessText

NEXT I
```

The above example will execute the loop 30 times but will only call the subroutine ProcessText when the current array element of Pattern is not a numeric value or null.

CONVERT

The CONVERT function is the function form of the CONVERT statement. It performs exactly the same function but may also operate on an expression rather than being restricted to variables.

COMMAND SYNTAX

CONVERT (expression1, expression2, expression3)

SYNTAX ELEMENTS

expression1 is the string to which the conversion will apply.
expression2 is the list of all characters to translate in expression1.
expression3 is the list of characters that will be converted to.

NOTE: For Prime, Universe and Unidata emulations:
expression1 is the list of all characters to translate in expression1.
expression2 is the list of characters that will be converted to.
expression3 is the string to which the conversion will apply.

See also: the CONVERT statement.

EXAMPLES

```
Value = CONVERT (Value, "#.,", "$,.")

Value = CONVERT(PartCode, "abc", "ABC")

Value = CONVERT(Code, "1234567890", "0987654321")
```

CONVERT (STATEMENT)

The CONVERT statement converts one or more characters in a string to their corresponding replacement characters.

COMMAND SYNTAX

CONVERT expression1 TO expression2 IN expression3

SYNTAX ELEMENTS

expression1 is the list of all characters to translate in expression3
expression2 is the list of characters that will be converted to.
expression3 is the string to which the conversion will apply.

NOTES

There is a one to one correspondence between the characters in expression1 and expression2. That is, conversion of character 1 in expression1 to character 1 in expression2, etc.
See also: the CONVERT function.

EXAMPLE

```
Value = 'ABCDEFGHIJ'

CRT 'Orignal:   ':Value

CONVERT 'BJE' TO '^+!' IN Value

CRT 'Converted: ':Value

Orignal:   ABCDEFGHIJ

Converted: A^CD!FGHI+
```

COS

The COS function calculates the cosine of any angle using floating point arithmetic, then rounds to the precision implied by the jBASE BASIC program, which makes it very accurate.

COMMAND SYNTAX

COS(expression)

This function calculates the cosine of an expression.

SYNTAX ELEMENTS

The expression must evaluate to a numeric result or a runtime error will occur.

NOTES

Assumes the value returned by expression is in degrees.

EXAMPLES

```
FOR I = 1 TO 360

    CRT COS(I) ;* print cos i for 1 to 360 degrees

NEXT I
```

COUNT

The COUNT function returns the number of times that one string occurs in another.

COMMAND SYNTAX

COUNT(expression1, expression2)

SYNTAX ELEMENTS

Both expression1 and expression2 may evaluate to any data type but logically they will evaluate to character strings.

NOTES

The count is made on overlapping occurrences as a pattern match from each character in expression1. This means that the string jjj occurs 3 times in the string jjjjj.
See also:DCOUNT.

EXAMPLES

```
Calc = "56 * 23 / 45 * 12"

CRT "There are ":COUNT(Calc, "*"):" multiplications"
```

COUNTS

Use the COUNTS function to count the number of times a substring is repeated in each element of a dynamic array. The result is a new dynamic array whose elements are the counts corresponding to the elements in the dynamic array.

COMMAND SYNTAX

COUNTS (dynamic.array, substring)

dynamic.array specifies the dynamic array whose elements are to be searched.
substring is an expression that evaluates to the substring to be counted. substring can be a character string, a constant, or a variable. Each character in an element is matched to substring only once. Therefore, when substring is longer than one character and a match is found, the search continues with the character following the matched substring. No part of the matched element is recounted toward another match. If substring does not appear in an element, a 0 value is returned. If substring is an empty string, the number of characters in the element is returned. If substring is null, the COUNTS function fails and the program terminates with a run-time error message. If any element in dynamic.array is null, null is returned.

EXAMPLE

```
ARRAY="A":@VM:"AA":@SM:"AAAAA"
PRINT COUNTS (ARRAY, "A")

PRINT COUNTS(ARRAY, "AA")
```

The output of this program is:

```
1]2\5
0]1\2
```

CREATE

Use the CREATE statement after an OPENSEQ statement to create a record in a jBASE directory file or to create a UNIX or DOS file. CREATE creates the record or file if the OPENSEQ statement fails. An OPENSEQ statement for the specified file.variable must be executed before the CREATE statement to associate the pathname or record ID of the file to be created with the file.variable. If file.variable is null, the CREATE statement fails and the program enters the debugger.

Use the CREATE statement when OPENSEQ cannot find a record or file to open and the next operation is to be a READSEQ or READBLK. If the first file operation is a WRITESEQ,WRITESEQ creates the record or file if it does not exist.

If the record or file is created, it executes the THEN statements; if no record or file is created, it executes the ELSE statements.

COMMAND SYNTAX

CREATE file.variable {THEN statements [ELSE statements] | ELSE statements}

EXAMPLE

In the following example, RECORD does not yet exist. When OPENSEQ fails to open RECORD to the file variable FILE, the CREATE statement creates RECORD in the type 1 file DIRFILE and opens it to the file variable FILE.

```
OPENSEQ 'DIRFILE', 'RECORD' TO FILE

ELSE CREATE FILE ELSE ABORT

WEOFSEQ FILE

WRITESEQ 'SOME DATA' TO FILE ELSE STOP
```

CRT

The CRT statement sends data directly to the terminal, even if a PRINTER ON statement is currently active.

COMMAND SYNTAX

CRT expression {, expression..} {:}

SYNTAX ELEMENTS

An expression can evaluate to any data type. The CRT statement will convert the result to a string type for printing. Expressions separated by commas will be sent to the screen separated by a tab character. The CRT statement will append a newline sequence to the final expression unless it is terminated with a colon ":" character.

NOTES

As the expression can be any valid expression, it may have output formatting applied to it.
A jBASE BASIC program is normally executed using buffered output mode. This means that data is not flushed to the terminal screen unless a newline sequence is printed or terminal input is requested. This makes it very efficient. However you can force output to be flushed to the terminal by printing a null character CHAR (0). This has the same effect as a newline sequence but without affecting screen output.
For compatibility, use DISPLAY in place of CRT.

EXAMPLES

```
CRT A "L#5"

CRT @ (8,20):"Shazza was here":

FOR I = 1 TO 200

    CRT @ (10,10):I:CHAR (0):

...
NEXT I
```

DATA

The DATA statement stacks the series of expressions on a terminal input FIFO stack. Terminal input statements will then treat this data as if entered at the keyboard.

COMMAND SYNTAX

DATA expression {, expression ...}

SYNTAX ELEMENTS

The expression may evaluate to any data type; views each comma-separated expression as one line of terminal input.

NOTES

The data stacked for input will subsequently be treated as input by any jBASE BASIC program. Therefore use it before PERFORM/EXECUTE, CHAIN or any other method of transferring program execution. Use also to stack input for the currently executing program; do not use to stack input back to an executing program.
When a jBASE BASIC program detects stacked data, it is taken as keyboard input until the stack is exhausted. The program will then revert to the terminal device for subsequent terminal input.
Stacked data delimited by field marks (xFE) will be treated as a series of separate terminal inputs.
See also: CLEARDATA

EXAMPLES

```
DATA "Y", "N", "CONTINUE" ;* stack input for prog

EXECUTE "PROGRAM1" ;* execute the program
```

DATE

The DATE( ) function returns the date in internal system form. This date is expressed as the number of days since December 31, 1967.

COMMAND SYNTAX

DATE( )

NOTES

The system and your own programs should manipulate date fields in internal form. They can then be converted to a readable format of your choice using the OCONV( ) function and the date conversion codes.

The year 2000 is a leap year

See also: TIMEDATE( )

EXAMPLES

```
CRT OCONV (DATE(), "D2")
```

displays today's date in the form: 14 JUL 64

DCOUNT

The DCOUNT( ) function counts the number of field elements in a string that are separated by a specified delimiter.

COMMAND SYNTAX

DCOUNT(expression1, expression2)

SYNTAX ELEMENTS

expression1 evaluates to a string in which fields are to be counted.

expression2 evaluates to the delimiter string used to count the fields.

NOTES

The delimiter string may consist of more than one character.

If expression1 is a NULL string, the function returns a value of zero.

The delimiter string may consist of any character, including system delimiters such as field marks or value marks.

See also: COUNT.

EXAMPLES

```
A = "A:B:C:D"

CRT DCOUNT(A, ":")
```

displays the value 4

DEBUG

The DEBUG statement causes the executing program to enter the jBASE BASIC debugger.

COMMAND SYNTAX

DEBUG

NOTES

Describes the debugger here.

EXAMPLES

```
IF FatalError = TRUE THEN

    DEBUG ;*enter the debugger

END
```

DECATALOG and DELETE-CATALOG Commands

The DECATALOG and DELETE-CATALOG commands are used to remove the run-time versions of cataloged jBASE BASIC programs.

COMMAND SYNTAX

```
DECATALOG SourceFilename ProgramName

DECATALOG ProgramName
```

DECRYPT

The DECRYPT function encrypts strings.

COMMAND SYNTAX

DECRYPT(string, key, method)

SYNTAX ELEMENTS

string specifies the string to be encrypted.
key is the value used to encrypt the string. Its use depends on method.
method is a value, which indicates the encryption mechanism to use (See below):

The ENCRYPT and DECRYPT functions that are part of jBASE BASIC now support the following cipher methods (Defined in JBC.h)

JBASE_CRYPT_GENERAL	General-purpose encryption scheme
JBASE_CRYPT_ROT13	Simple ROT13 algorithm. (Key not used)
JBASE_CRYPT_XOR11	XOR MOD11 algorithm. Uses the first character of a key as a seed value.
JBASE_CRYPT_RC2	RC2 algorithm
JBASE_CRYPT_DES	DES algorithm
JBASE_CRYPT_3DES	Three Key, Triple DES algorithm
JBASE_CRYPT_BLOWFISH	Blowfish algorithm
JBASE_CRYPT_BASE64	(See below)

BASE64 is not really an encryption method, but more of an encoding. The reason for this is that the output of an encryption often results in a binary string. It allows binary data to be represented as a character string. BASE64 operation is not required but is performed in addition to the primary algorithm. e.g. JBASE_CRYPT_RC2_BASE64
ENCRYPT with this method is the same as a DECRYPT with method JBASE_CRYPT_RC2 followed by DECRYPT with method JBASE_CRYPT_BASE64.
DECRYPT with this method is the same as a DECRYPT with method JBASE_CRYPT_BASE64 followed by DECRYPT with method JBASE_CRYPT_RC2.

JBASE_CRYPT_RC2_BASE64	RC2 algorithm
JBASE_CRYPT_DES_BASE64	DES algorithm
JBASE_CRYPT_3DES_BASE64	Triple DES algorithm
JBASE_CRYPT_BLOWFISH _BASE64	Blowfish algorithm

NOTES

See also: ENCRYPT

EXAMPLES

```
INCLUDE JBC.h

X = DECRYPT(X, Ekey, JBASE_CRYPT_GENERAL)

IF DECRYPT("rknzcyr”,"", JBASE_CRYPT_ROT13) = "example" THEN

CRT "ROT13 ok"

END

IF ENCRYPT("g{ehvkm","9", JBASE_CRYPT_XOR11) = "example" THEN

CRT "XOR.MOD11 ok"

END

cipher = JBASE_CRYPT_BLOWFISH_BASE64

key    = "Our Very Secret Key"

str    = "String to encrypt"

enc = ENCRYPT( str, key, cipher )

CRT "Encrypted: ":enc

dec = DECRYPT( enc, key, cipher )

CRT "Decrypted: ":dec
```

Displays as output:

```
Encrypted: xuy6DXxUkD32spyfsKEvUtXrsjP7mC+R

Decrypted: String to encrypt
```

DEFC

Use the DEFC statement to declare an external C function to the jBASE BASIC compiler, define its arguments, and return types. The DEFC statement assumes that the C functions will need to manipulate jBASE BASIC variables and hence will also require the thread data pointer. As such, all C functions require recoding to include the data pointer as an argument to the C function. The location of the data pointer argument depends upon the function return type.

COMMAND SYNTAX

DEFC {FuncType} FuncName ({ArgType {, ArgType ...}})

SYNTAX ELEMENTS

FuncType and ArgType are selected from one of INT, FLOAT or VAR. FuncType specifies the type of result that the function will return. Assumes INT if FuncType is omitted. The optional list of ArgTypes specifies the argument types that the C function will expect. The compiler must know this in advance, as it will automatically perform type conversions on these arguments.

EXAMPLE

```
#include <jsystem.h>

#include <assert.h>


#ifdef DPSTRUCT_DEF

#define JBASEDP         DPSTRUCT *dp,

#else

#define JBASEDP

#endif

VAR *MyString(VAR *Result, JBASEDP  VAR *VarPtr)

{

char *Ptr;

    assert(dp != NULL);

    Ptr = (char *) CONV_SFB(VarPtr);

    printf("MyString: %s - %d\n", Ptr, strlen(Ptr) );
```

```
    STORE_VBI(Result, strlen(Ptr) );

    return(Result);

}

INT32 MyCalc(INT32 Value1, INT32 Value2)

{

INT32 Result;

    Result =  (Value1 / Value2);

    printf("MyCalc: %d\n", Result);

    return(Result);

}
```

NOTES

Compile a DEFC for each C function before making any reference to it else the compiler will not recognize the function name.

The function is called in the same manner, as it would be in a C program, which means it can be used as if it was an intrinsic function of the jBASE BASIC language and therefore returns a value. However, specifying it as a standalone function call causes the compiler to generate code that ignores any returned values.

When passing jBASE BASIC variables to a C function, you must utilize the predefined macros to access the various data types it contains. C functions are particularly useful for increasing the performance of tight loops that perform specific functions. The jBASE BASIC compiler must cater for any eventuality within a loop (such as the controlling variable changing from integer to floating point). A dedicated C function can ignore such events, if they are guaranteed not to happen.

The jBASE BASIC programmer may freely ignore the type of argument used when invoking the C function, as the jBASE BASIC compiler will automatically perform type conversion.

DEFCE

With jBASE the DEFCE statement should be used, rather than the DEFC statement, for calling external C programs, which are pure 'C' code and do not use the jBASE library macro's and functions.

EXAMPLE 1

For C functions that do not require jBASE functions use the DEFCE statement, however the passing arguments can only be of type INT, FLOAT and STRING.

```
DEFCE INT MYFUNC3(INT)

INT32 MYFUNC3(INT32 Count)

{

INT32 Result;

      ….

      return Result;

}
```

EXAMPLE 2

```
DEFCE INT cfunc( INT, FLOAT, VAR)

Var1 = cfunc( A, 45, B)

cfunc( 34, C, J)
```

You can call standard UNIX functions directly by declaring them with the DEFC statement according to their parameter requirements. You can only call them directly providing they return one of the type int or float/double or that the return type may be ignored.

EXAMPLE 3

```
DEFCE INT getpid()

CRT "Process id =":getpid()
```

DEFFUN

Use the DEFFUN statement to declare an external jBASE BASIC function to the jBASE BASIC compiler and optionally define its arguments. Use DEFFUN in the program that calls the function.

COMMAND SYNTAX

DEFFUN FuncName ({ {MAT} Argument1, {MAT} Argument2...})

SYNTAX ELEMENTS

FuncName is the name used to define the function. It must be the same as the source file name. **Argument** specifies a value passed to the function by the calling program. To pass an array, the keyword you must use the MAT before the argument name. These parameters are optional (as indicated in the Command Syntax) but can be specified for clarity. Note that if the arguments are not initialized somewhere in the program you will receive a compiler warning.

NOTES

The DEFFUN statement identifies a user-written function to the jBASE BASIC compiler, which must be present in each program that calls the function, before the function is called. A hidden argument is passed to the function so that a value can be returned to the calling program. The return value is set in the function using the RETURN (value) statement. If the RETURN statement specifies no value then the function returns an empty string.

EXAMPLE 1

```
DEFFUN Add()

A = 10

B = 20

sum = Add(A, B)

PRINT sum

X = RND (42)

Y = RND(24

)
PRINT Add(X, Y)

FUNCTION Add(operand1, operand2)

result = operand1 + operand2

RETURN(result)
```

Call standard UNIX functions directly by declaring them with the DEFCE statement according to their parameter requirements. However, they may only be called directly providing they return one of the type int or float/double or that the return type may be ignored.

EXAMPLE 2

```
DEFCE INT getpid()

CRT "Process id =":getpid()
```

DEL

Use the DEL statement to remove a specified element of a dynamic array.

COMMAND SYNTAX

DEL variable<expression1{, expression2{, expression3}}>

SYNTAX ELEMENTS

The variable can be any previously assigned variable or matrix element. The expressions must evaluate to a numeric value or a runtime error will occur.

expression1 specifies the field in the array to operate upon and must be present.

expression2 specifies the multivalue within the field to operate upon and is an optional parameter.

expression3 is optionally present when expression2 has been included. It specifies which subvalue to delete within the specified multivalue.

NOTES

Truncates non-integer values for any of the expressions to integers

Ignores invalid numeric values for the expressions without warning

The command operates within the scope specified, i.e. if specifying only a field then it deletes the entire field (including its multivalues and subvalues). If specifying a subvalue, then it deletes only the subvalue leaving its parent multivalue and field intact.

EXAMPLES

```
FOR I = 1 TO 20

    Numbers<I> = I   ;*generate numbers

NEXT I

FOR I = 19 TO 1 STEP -2

    DEL Numbers<I>   ;*remove odd numbers

NEXT I
```

DELETE

Use the DELETE statement to delete a record from a jBASE file.

COMMAND SYNTAX

DELETE {variable,} expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

If specified, variable should have been the subject of a previous OPEN statement. If variable is omitted then it assumes the default file variable.

The expression should evaluate to the name of a record stored in the open file.

If the SETTING clause is specified and the delete fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

The statement will have no effect if the record name does not exist within the file.

If the program against the file record was holding a lock, it will release the lock.

EXAMPLES

```
OPEN "DAT1" TO DatFile1 ELSE ABORT 201, "DAT1"

DELETE DatFile1, "record1"
```

will delete the record "record1" from the file DAT1

DELETELIST

The DELETELIST statement will delete the previously stored list named by expression.

COMMAND SYNTAX

DELETELIST expression

SYNTAX ELEMENTS

The expression should evaluate to the name of a list that has been stored with either the WRITELIST statement or the SAVE-LIST command from the shell.

NOTES

If POINTER-FILE is accessible then it saves lists within else are saved in the jBASE work file.

EXAMPLES

```
List = "JobList"

DELETELIST List
```

Will delete the pre-saved list called JobList

DELETESEQ

DELETESEQ deletes a sequential file.

COMMAND SYNTAX

DELETESEQ Expression {SETTING setvar} {ON ERROR statements} {LOCKED statements} THEN | ELSE statements

or

DELETESEQ Expression, Filename {SETTING setvar} {ON ERROR statements} {LOCKED statements} THEN | ELSE statements

SYNTAX ELEMENTS

Expression specifies the variable to contain next record from sequential file.

FileVar specifies the file descriptor of the file opened for sequential access.

Statements conditional jBASE BASIC statements

DELETEU

Use the DELETEU statement to delete a record without releasing the update record lock set by a previous READU statement

See also:READ statements.

Use the OPEN statement to open a file. If specifying a file variable in the OPEN statement, use it in the DELETEU statement. You must place a comma between the file variable and the record ID expression. If specifying no file variable in the DELETEU statement, the statement applies to the default file.

See also: OPEN statement for a description of the default file.

DIMENSION

Use the DIM statement to declare arrays to the compiler before referencing.

COMMAND SYNTAX

DIM{ENSION} variable(number{, number... }){, variable(number {,number...}) ...}

SYNTAX ELEMENTS

The **variable** may be any valid variable name neither declared nor previously used. The numbers define the size of each dimension and must be either constants or the subject of an EQUATE statement. A single DIM statement may declare a number of arrays by separating their declarations with a comma.

NOTES

Declare the array before it is referenced in the program source (compilation as opposed to execution).
If using a variable as an undeclared dimensioned array the compiler will display an error message.
Do not use the array variable as a normal variable or dynamic array before dimensioning, as the compiler will detect this as an error.
A dimension size may not be specified as one as this has no logical meaning. The compiler will detect this as a warning.
When arrays are referenced directly as in A = Array(7), the compiler will optimize the reference as if it was a single undimensioned variable.
See also: COMMON

EXAMPLES

```
EQUATE DimSize1 TO 29

DIM Array1(10,10), Array2(5, 20, 5, 8)

DIM Age(DimSize1)
```

DIR

Use the DIR function to return information about a file.

COMMAND SYNTAX

DIR (filename)

The filename is a string argument representing the path and filename of a file. This function returns a dynamic array with four attributes.

Attribute	Description
1	File size in bytes
2	last modified date (in internal format)
3	last modified time (in internal format)
4	D if the filename is a directory, blank if the filename is a file

EXAMPLE

```
F = DIR(“.”)
PRINT F
```

“0{am}0{am}0{am}D”: is the output of this program.

DIV

See also: Floating point Operations

Use the DIV function to calculate the value of the quotient after division of the dividend by the divisor.

COMMAND SYNTAX

DIV (dividend, divisor)
The dividend and divisor expressions can evaluate to any numeric value. The only exception is that the divisor cannot be zero. If either dividend or divisor evaluates to null, it returns null.

EXAMPLE

```
I=400; K=200

J = DIV (I,K)

PRINT J
```

2: is the output of this program.

DIVS

See also: Floating point Operations

Use the DIVS function to create a dynamic array containing the result of the element-by-element division of two dynamic arrays.

COMMAND SYNTAX

DIVS (array1, array2)

The division of each element of array1 is by the corresponding element of array2, which returns the result in the corresponding element of a new dynamic array. If elements of array1 have no corresponding elements in array2, it pads array2 with ones and returns the array1 elements. If an element of array2 has no corresponding element in array1, it returns zero. If an element of array2 is zero, it prints a run-time error message and returns 0. If either element of a corresponding pair is null, it returns null.

EXAMPLE

```
A=10:@VM:15:@VM:9:@SM:4
B=2:@VM:5:@VM:9:@VM:2
PRINT DIVS(A,B)
```

The output of this program is: 5]3]1\4]0

DOWNCASE / UPCASE

DOWNCASE converts all uppercase characters in an expression to lowercase characters.
UPCASE converts all lowercase characters in an expression to uppercase characters.

COMMAND SYNTAX

DOWNCASE|LOWCASE(expression) / UPCASE (expression)

INTERNATIONAL MODE

When using the DOWNCASE or UPCASE function in International Mode the conversion from upper case to lower case or vice versa will be determined for each character in the expression by the Unicode standard, which describes the up or down case properties for the character.

SYNTAX ELEMENTS

expression in a string containing some alphabetic characters

NOTES

It ignores Non-alphabetic characters.

DROUND

See also: Floating point Operations

The DROUND function performs double-precision rounding on a value. Double-precision rounding uses two words to store a number, accommodating a larger number than in single-precision rounding, which stores each number in a single word.

COMMAND SYNTAX

DROUND(val.expr [,precision.expr])

NOTE: DROUND affects the internal representation of the numeric value. It performs the rounding without conversion to and from string variables. This increases the speed of calculation.

SYNTAX ELEMENTS

val.expr specifies the value to round.

precision.expr specifies the precision for the rounding. The valid range is 0 to 14. Default precision is four places.

EXAMPLE

In the following example, the DROUND statement results in 18.84955596. The equation is resolved, and rounds the result to eight decimal places.

```
A= DROUND((3.14159265999*2*3),8)


PRINT A
```

DTX

The DTX function will return the hexadecimal representation of a numeric expression.

COMMAND SYNTAX

DTX(expression)

SYNTAX ELEMENTS

expression must evaluate to a decimal numeric value or a runtime error will occur.

NOTES

See also: XTD.

EXAMPLES

```
Decimal = 254

CRT DTX(Decimal)
```

displays FE

DYNTOXML

COMMAND SYNTAX

DYNTOXML (array,xsl,result)

SYNTAX ELEMENTS

Convert the array to XML using the optimal xsl to transform

```
XML = (DYNTOXML(array,"",result)
```

Takes the contents of the dynamic array held in an array, and returns a generic XML representation of that array or an error

```
(result=0 OK; result<>0 Bad);
```

EXAMPLE

```
a = "Tom" : @AM : "Dick" : @AM : "Harry"
      xml = DYNTOXML(a,"",result)
      CRT xml
```

SCREEN OUTPUT

```
<?xml version="1.0" encoding ="ISO-8859-1"?>
<array>
  <data attribute="1" value="1" subvalue="1">Tom</data>
  <data attribute="2" value="1" subvalue="1">Dick</data>
  <data attribute="3" value="1" subvalue="1">Harry</data>
</array>
```

If a style sheet is passed in the second parameter, it performs a transform to give a different format of XML.

EXAMPLE

```
xml = DYNTOXML(a,xsl,result)
      CRT xml
```

SCREEN OUTPUT

```
<mycustomer>
  <firstname>Tom</firstname>
  <lastname>Dick</lastname>
  <address>Harry</address>
</mycustomer>
```

XSL CONTENTS

```
<xsl:template match="/">
<mycustomer>
```

```
<xsl:for-each select="array/data">
  <xsl:if test="@attribute=1">
    <firstname>
      <xsl:value-of select="."/>
    </firstname>
  </xsl:if>
  <xsl:if test="@attribute=2">
    <lastname>
      <xsl:value-of select="."/>
    </lastname>
  </xsl:if>
  <xsl:if test="@attribute=3">
    <address>
      <xsl:value-of select="."/>
    </address>
  </xsl:if>
  <xsl:if test="@attribute=4">
    <address2>
      <xsl:value-of select="."/>
    </address2>
 </xsl:if>
```

EBCDIC

The EBCDIC function converts all the characters in an expression from the ASCII character set to the EBCDIC character set.

COMMAND SYNTAX

EBCDIC(expression)

SYNTAX ELEMENTS

expression may contain a data string of any form. The function will convert it to a character string, assume that the characters are all members of the ASCII set and translate them using a character map. The original expression is unchanged while the returned result of the function is now the EBCDIC equivalent.

EXAMPLE

```
READT AsciiBlock ELSE CRT "Tape failed!"; STOP

EbcdicBlock = EBCDIC(AsciiBlock) ;* Convert to EBCDIC
```

ECHO

The ECHO statement will turn on or off the echoing of characters typed at the keyboard.

COMMAND SYNTAX

ECHO ON

ECHO OFF

ECHO expression

SYNTAX ELEMENTS

Use the statement with the keywords ON and OFF to specify echoing or not. If used with an expression, then the expression should evaluate to a Boolean TRUE or FALSE result.
TRUE: echoing on
FALSE: echoing off.

NOTES

Use the SYSTEM function to determine the current state of character echoing. SYSTEM(24) returns Boolean TRUE if enabled and returns Boolean FALSE if disabled.

EXAMPLES

```
ECHO OFF

CRT "Enter your password ":

INPUT Password

ECHO ON

.....
```

This will disable the character input echoing while typing in a password.

ENCRYPT

The ENCRYPT function encrypts strings.

COMMAND SYNTAX

ENCRYPT(string, key, method)

SYNTAX ELEMENTS

string specifies the string for encryption.
key is the value used to encrypt the string. Its use depends on method.
method is a value, which indicates the encryption mechanism to use (See below):

The ENCRYPT and DECRYPT functions that are part of jBASE BASIC now support the following cipher methods (Defined in JBC.h)

JBASE_CRYPT_GENERAL	General-purpose encryption scheme
JBASE_CRYPT_ROT13	Simple ROT13 algorithm. (Key not used)
JBASE_CRYPT_XOR11	XOR MOD11 algorithm. Uses the first character of a key as a seed value.
JBASE_CRYPT_RC2	RC2 algorithm
JBASE_CRYPT_DES	DES algorithm
JBASE_CRYPT_3DES	Three Key, Triple DES algorithm
JBASE_CRYPT_BLOWFISH	Blowfish algorithm
JBASE_CRYPT_BASE64	(See below)

BASE64 is more of an encoding method rather than an encryption method. The reason for this is that the output of an encryption often results in a binary string, which allows the representation of binary data as a character string. Although not required the BASE64 operation is performed in addition to the primary algorithm. E.g. JBASE_CRYPT_RC2_BASE64

ENCRYPT with this method is the same as an ENCRYPT with method JBASE_CRYPT_RC2 followed by ENCRYPT with method JBASE_CRYPT_BASE64.

DECRYPT with this method is the same as a DECRYPT with method JBASE_CRYPT_BASE64 followed by DECRYPT with method JBASE_CRYPT_RC2.

JBASE_CRYPT_RC2_BASE64	RC2 algorithm
JBASE_CRYPT_DES_BASE64	DES algorithm
JBASE_CRYPT_3DES_BASE64	Triple DES algorithm
JBASE_CRYPT_BLOWFISH _BASE64	Blowfish algorithm

NOTES

See also: DECRYPT.

EXAMPLES

```
INCLUDE JBC.h

X = DECRYPT(X, Ekey, JBASE_CRYPT_GENERAL)

IF DECRYPT("rknzcyr”,"", JBASE_CRYPT_ROT13) = "example" THEN

CRT "ROT13 ok"

END

IF ENCRYPT("g{ehvkm","9", JBASE_CRYPT_XOR11) = "example" THEN

CRT "XOR.MOD11 ok"

END

cipher = JBASE_CRYPT_BLOWFISH_BASE64

key    = "Our Very Secret Key"

str    = "String to encrypt"

enc = ENCRYPT( str, key, cipher )

 CRT "Encrypted: ":enc

dec = DECRYPT( enc, key, cipher )

CRT "Decrypted: ":dec
```

Displays as output:

```
Encrypted: xuy6DXxUkD32spyfsKEvUtXrsjP7mC+R

Decrypted: String to encrypt
```

NOTES

See also: DECRYPT.

ENTER

The ENTER statement unconditionally passes control to another executable program.

COMMAND SYNTAX

ENTER program_name

ENTER @variable_name

SYNTAX ELEMENTS

program_name is the name of the program for execution. The use of single or double quotes to surround program_name is optional.
@ specifies that the program name is contained in a named variable.
variable_name is the name of the variable, which contains the program name.

NOTES

The jBASE BASIC COMMON data area can be passed to another jBASE BASIC program by specifying the option "I" after the program name. Pass the COMMON data area only to another jBASE BASIC program.
Use ENTER to execute any type of program.
If the program which contains the ENTER command (the current program) was called from a JCL program, and the program for execution (the target program) is another jBASE BASIC program, control will return to the original JCL program when the target program terminates. If the target program is a JCL program, control will return to the command shell when the JCL program terminates.

EXAMPLES

```
ENTER "menu"

ProgName = "UPDATE"

ENTER @ ProgName
```

EQS

Use the EQS function to test if elements of one dynamic array are equal to the elements of another dynamic array.

COMMAND SYNTAX

EQS (array1, array2)

EQS compares each element of array1 with the corresponding element of array2 and returns, a one if the two elements are equal in the corresponding element of a dynamic array. It returns a zero if the two elements are not equal. It returns zero if an element of one dynamic array has no corresponding element in the other dynamic array. If either element of a corresponding pair is null, it returns null for that element.

EXAMPLE

```
A=1:@VM:45:@SM:3:@VM:"one"

B=0:@VM:45:@VM:1

PRINT EQS(A,B)
```

The output of this program is: 0]1\0]0

EQUATE

Use EQUATE to declare a symbol equivalent to a literal, variable or simple expression.

COMMAND SYNTAX

EQU{ATE} symbol TO expression

SYNTAX ELEMENTS

symbol is the name of the symbol to use;.can be any name that would be valid for a variable.
expression can be a literal, a variable or a simple expression.

NOTES

Sensible use of EQUATEd symbols can make your program easier to maintain, easier to read, and more efficient.
Efficiency can be enhanced because the address of an EQUATEd value is computed during compilation and is substituted for each occurrence of the symbol name. Unlike the address of a variable, which must be computed for each access during run time, the address of a symbol is always known. This significantly reduces the processing overhead involved in accessing a particular value. See also: the example for a more detailed explanation of the other benefits.
Enhance Readability by referring to say, QTY rather than INV_LINE(4). You would simply "EQUATE QTY TO INV_LINE(4)" at an early stage in the program. This can also help with maintenance of the program, particularly in situations where record layouts might change. For example, if the quantity field moves to INV_LINE(6), you only have to change one line in your program.

EXAMPLE

```
COMMON FLAG

EQUATE NO_CHARGE TO FLAG

EQUATE CR TO CHAR (13), TRUE TO 1, FALSE TO 0

EQUATE PRICE TO INV_LINE(7), TAX TO 0.175

EQUATE DASHES TO "-------"

IF NO_CHARGE = TRUE THEN PRICE = 0

CRT "Tax =":PRICE * TAX:CR:DASHES
```

EREPLACE

Use the EREPLACE function to replace substring in an expression with another substring. If you do not specify an occurrence, it replaces each occurrence of a substring.

COMMAND SYNTAX

EREPLACE (expression, substring, replacement [,occurrence [,begin]])

SYNTAX ELEMANTS

occurrence specifies the number of occurrences of substring to replace. To replace all occurrences, specify occurrence as a number less than 1. **begin** specifies the first occurrence to replace. If begin is omitted or less than one, it defaults to one. If **substring** is an empty string, replacement is prefixed to expression. If replacement is an empty string, it removes all occurrences of substring. If **expression** evaluates to null, it returns null. If substring, replacement, occurrence, or begin evaluates to null, the EREPLACE function fails and the program terminates with a run-time error message. The EREPLACE function behaves like the CHANGE function except when substring evaluates to an empty string.

EXAMPLE

```
A = "AAABBBCCCDDDBBB"

PRINT EREPLACE (A,"BBB","ZZZ")

PRINT EREPLACE (A,"","ZZZ")

PRINT EREPLACE (A,"BBB","")
```

The output of this program is:

```
AAAZZZCCCDDDZZZ
ZZZAAABBBCCCDDDBBB
AAACCCDDD
```

EXECUTE

See also: Floating point Operations

The EXECUTE or PERFORM statement allows the currently executing program to pause and execute any other UNIX/NT program, including another jBASE BASIC program or a jBASE command.

COMMAND SYNTAX

EXECUTE|PERFORM expression {CAPTURING variable} {RETURNING|SETTINGvariable} {PASSLIST {expression}} {RTNLIST {variable}}{PASSDATA variable} {RTNDATA variable}

Passes Data, Dynamic Arrays and lists to programs written in jBASE BASIC, you can intercept screen output and error messages from any program.

SYNTAX ELEMENTS

The PERFORMed expression can be formed from any jBASE construct. The system will not verify that the command exists before executing it. Use a new Bourne Shell to execute a command (sh) by default. The shell type can be changed by preceding the command with a CHAR(255) concatenated with either "k", "c", or "s" to signify the Korn shell, C shell or Bourne Shell.

Variables used to pass data to the executed program should have been assigned to a value before using. You can use any variable name to receive data.

CAPTURING variable

The capturing clause will capture any output that the executing program would normally send to the terminal screen and place it in the variable specified. A field mark in the variable replaces every newline normally sent to the terminal.

RETURNING variable or SETTING variable

The returning and setting clauses are identical. Both clauses will capture the output associated with any error messages the executing program issues. The first field of the variable will be set to the exit code of the program.

PASSLIST variable

The PASSLIST clause allows jBASE programs to exchange lists or dynamic arrays between them. The variable should contain the list that the program wishes to pass to the jBASE program it is executing. The program to be executed should be able to process lists, otherwise the list will just be ignored. If the variable name is not specified then the clause will pass the default select list to the executing program.

RTNLIST variable

If the executed program sets up a list then use the RTNLIST clause to place that list into a specified variable. It places the list in the default list variable if omitted.

PASSDATA variable

Passes the data in the specified variable to another jBASE BASIC program, the executing jBASE BASIC program should retrieve the data using the COLLECTDATA statement.

RTNDATA variable

The RTNDATA statement returns any data passed from an executing jBASE BASIC program in the specified variable. The executing jBASE BASIC program should use the RTNDATA statement to pass data back to the calling program.

NOTES

The clauses may be specified in any order within the statement but only one of each clause may exist.

EXAMPLES

```
OPEN "DataFile" ELSE ABORT 201, "DataFile"

SELECT
PERFORM "MyProg" SETTING ErrorList PASSLIST

EXECUTE "ls" CAPTURING DirListing
```

EXIT

The EXIT statement halts the execution of a program and returns a numeric exit code to the parent process. For compatibility with older versions of the language, use the EXIT statement without an expression. In this case, it is synonymous with the BREAK statement.

COMMAND SYNTAX

EXIT (expression)
EXIT

SYNTAX ELEMENTS

Any expression provided must be parenthesized and evaluate to a numeric result. The numeric result is used as the UNIX or Windows exit code, which is returned to the parent process by the C function exit(). If the expression does not evaluate to a numeric result the program will enter the debugger and display a suitable error message.

NOTES

The expression has been forced to be parenthesized to avoid confusion with the EXIT statement without an expression as much as is possible. The authors apologize for having to provide two different meanings for the same keyword

See also:BREAK.

EXAMPLE

```
READ Record FROM FileDesc, RecordKey ELSE

    CRT "Record ":RecordKey:" is missing"

    EXIT(1)
END ELSE

    CRT "All required records are present"

    EXIT(0)
END
```

EXP

The EXP function returns the mathematical constant to the specified power.

COMMAND SYNTAX

EXP(expression)

SYNTAX ELEMENTS

The expression may consist of any form of jBASE BASIC expression but should evaluate to a numeric argument or a runtime error occurs and the program enters the debugger.

NOTES

The function returns a value that is accurate to as many decimal places specified by the PRECISION of the program.

EXAMPLE

```
zE10 = EXP(10) ;* Get e^10
```

EXTRACT

The EXTRACT function is an alternative method of accessing values in a dynamic array other than using the <n,n,n> syntax described earlier.

COMMAND SYNTAX

EXTRACT(expression1, expression2 {, expression3 {, expression4}})

SYNTAX ELEMENTS

expression1 specifies the dynamic array to work with and will normally be a previously assigned variable.

The expressions 2 through 4 should all return a numeric value or a runtime error will occur and the program will enter the debugger.

expression2 specifies the field to extract, expression3 the value to extract and expression4 the sub-value to extract.

EXAMPLES

```
A = "0"; A<2> = "1"; A<3> = "2"

CRT EXTRACT(A, 2)
```

Will display the value "1".

FADD

The FADD function performs floating point addition of two numeric values.

COMMAND SYNTAX

FADD(expression1, expression2)

SYNTAX ELEMENTS

Both expression1 and expression 2 must evaluate to non-null numeric values.

NOTES

If either of the arguments evaluates to null then a run time "non-numeric" error will occur.

EXAMPLES

```
PRECISION 7
CRT FADD(0.5044,23.7290002)

displays 24.2334002
```

FDIV

The FDIV function performs floating point division on two numeric values.

COMMAND SYNTAX

FDIV(expression1, expression2)

SYNTAX ELEMENTS

Both expression1 and expression 2 must evaluate to non-null numeric values.

NOTES

If either of the arguments evaluates to null then a run time "non-numeric" error will occur.
If the second argument evaluates to zero then a run time "divide by zero" error will occur.
The calculation is not subject to the PRECISION setting.

EXAMPLES

```
CRT FMUL(1,7)

displays 0.1428571429
```

FIELD

The FIELD function returns a multi-character delimited field from within a string.

COMMAND SYNTAX

FIELDS(string, delimiter, occurrence{, extractCount})

SYNTAX ELEMENTS

string specifies the string, from which the field(s) is to be extracted.
delimiter specifies the character or characters that delimit the fields within the dynamic array.
occurrence should evaluate to an integer of value 1 or higher. It specifies the delimiter used as the starting point for the extraction.
extractCount is an integer that specifies the number of fields to extract. If omitted, assumes one.

NOTES

If the emulation option, jbase_field, is set then the field delimiter may consist of more than a single character, allowing fields to be delimited by complex codes.
See also: GROUP

EXAMPLES

```
Fields = "AAAA:BBJIMBB:CCCCC"

CRT FIELD(Fields, ":", 3)

CRT FIELD(Fields, "JIM", 1)
```

displays:

```
CCCCC
AAAA:BB
```

FIELDS

The FIELDS function is an extension of the FIELD function. It returns a dynamic array of multi-character delimited fields from a dynamic array of strings.

COMMAND SYNTAX

FIELDS(DynArr, Delimiter, Occurrence{, ExtractCount})

SYNTAX ELEMENTS

DynArr should evaluate to a dynamic array.
Delimiter specifies the character or characters that delimit the fields within the dynamic array.
Occurrence should evaluate to an integer of value 1 or higher. It specifies the delimiter used as the starting point for the extraction.
ExtractCount is an integer that specifies the number of fields to extract. If omitted, assumes one.

NOTES

If the emulation option, jbase_field, is set then the field delimiter may consist of more than a single character, allowing fields to be delimited by complex codes.

EXAMPLES

The following program shows how each element of a dynamic array can be changed with the FIELDS function.

```
t = ""

t<1> = "a:b:c:d:e:f"

t<2> = "aa:bb:cc:dd:ee:ff" : @VM: "1:2:3:4" : @SVM: ":W:X:Y:Z"

t<3> = "aaa:bbb:ccc:ddd:eee:fff":@VM:@SVM

t<4> = "aaaa:bbbb:cccc:dddd:eeee:ffff"

r1 = FIELDS(t,":",2)

r2 = FIELDS(t,":",2,3)

r3 = FIELDS(t,"bb",1,1)
```

The above program creates three dynamic arrays.
V - represents a value mark.
s - represents a sub-value mark.

```
r1              <1>b
                <2>bb v 2 s W
```

<3>bbb

<4>bbbb

r2 <1>b:c:d

<2>bb:cc:dd v 2:3:4 s W:X:Y<3>bbb:ccc:ddd v s

<4>bbbb:cccc:dddd

r3 <1>a:b:c:d:e:f

<2>aa: v 1:2:3:4 s W:X:Y:Z

<3>aaa: v s

<4>aaaa:

FILEINFO

Use the FILEINFO function to return information about the specified file variable.

COMMAND SYNTAX

FILEINFO (file.variable, key)

This function is currently limited to return values to determine if the file variable is a valid file descriptor variable.

Key Return Status

01 if file.variable is a valid files variable zero otherwise.

FILELOCK

Use the FILELOCK statement to acquire a lock on an entire file. This prevents other users from updating the file until the program releases it. A FILELOCK statement that does not specify lock.type is equivalent to obtaining an update record lock on every record of the file. An open file is specified by file.variable. If no file.variable is specified, the default file is assumed; if the file is neither accessible nor open, the program enters the debugger.

COMMAND SYNTAX

FILELOCK filevar {LOCKED statements} {ON ERROR statements}
FILEUNLOCK filevar {ON ERROR statements}

DESCRIPTION

When the FILELOCK statement is executed, it will attempt to take an exclusive lock on the entire file. If there are any locks currently outstanding on the file, then the statement will block until there are no more locks on the file. The use of the LOCKED clause allows the application to perform an unblocked operation.
When the FILELOCK statement is blocked waiting for a lock, other processes may continue to perform database operations on that file, including the removal of record locks and the taking of record locks. Once the FILELOCK is taken, it will block ALL database accesses to the file whether or not the access involves record locks. i.e. a READ will block once it has been executed, as will, CLEARFILE etc,. The lock continues until the file is closed, the program terminates, or a FILEUNLOCK statement is executed.
NOTE: The FILELOCK statement might differ to those found on other vendors systems. You should also not that the use of these statements for other than administration work, for example, within batch jobs, is not recommended. The replacement of such with more judicious use of item locks is advised.

IMPLEMENTATION NOTES

The FILELOCK command is implemented using the native locking mechanism of the operating system and is entirely at its mercy. Because of this, you may see some slight implementation differences between operating systems. These comments on native locking do not apply to the NT platform as jBASE uses the NT locking mechanism.
The uses of the native (UNIX) locking mechanism means the file in question MUST NOT use the jBASE locking mechanism. You can set a file to use the native locking mechanism by using the jchmod command:

```
jchmod +N filename {filename ...}
```

Alternatively, like this when the file is originally created:

```
CREATE-FILE filename 1,1 23,1 NETWORK=TRUE
```

If the file continues to use the jBASE record locking, then the ON ERROR clause will be taken and the SYSTEM(0) and STATUS() functions will set to 22 to indicate the error.

EXAMPLES

```
OPEN '','SLIPPERS' TO FILEVAR ELSE STOP "CAN'T OPEN FILE"

FILELOCK FILEVAR LOCKED STOP 'FILE IS ALREADY LOCKED'

FILEUNLOCK DATA

OPEN '','SLIPPERS' ELSE STOP "CAN'T OPEN FILE"

FILELOCK LOCKED STOP 'FILE IS ALREADY LOCKED'

PRINT "The file is locked."

FILEUNLOCK
```

FILEUNLOCK

Use the FILEUNLOCK statement to release a file lock set by the FILELOCK statement.

COMMAND SYNTAX

FILEUNLOCK [file.variable] [ON ERROR statements]

file.variable specifies a file previously locked with a FILELOCK statement. If file.variable is not specified, the default file with the FILELOCK statement is assumed .If file.variable is not a valid file variable then the FILEUNLOCK statement will enter the debugger.

The ON ERROR Clause

The ON ERROR clause is optional in the FILELOCK statement. The ON ERROR clause lets you specify an alternative for program termination when encountering a fatal error during processing of the FILELOCK statement. If a fatal error occurs, with no ON ERROR clause specified, the program enters the debugger.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

EXAMPLE

In the following example, the first FILEUNLOCK statement unlocks the default file. The second FILEUNLOCK statement unlocks the file variable FILE.

```
OPEN '','SLIPPERS' ELSE STOP "CAN'T OPEN SLIPPERS"

FILELOCK

FILEUNLOCK

OPEN 'PIPE' TO FILEVAR ELSE STOP

FILELOCK FILEVAR

FILEUNLOCK FILEVAR
```

FIND

The FIND statement allows the location of a specified string within a dynamic array.

COMMAND SYNTAX

FIND expression1 IN Var1 {, expression2} SETTING Var2 {, Var3 {, Var4}} THEN | ELSE statement(s)

SYNTAX ELEMENTS

expression1 evaluates to the string with which to compare every element of the dynamic array. **Var1** is the dynamic array that will be searched. The FIND command will normally find the first occurrence of **expression1** unless expression2 is specified. If specified then expression2 will cause a specific occurrence of expression1 to be located. The three variables Var2, Var3, Var4 are used to record the Field, Value and Sub-Value positions in which expression1 was found.
If expression1 is found in any element of Var1 then Vars 2, 3 and 4 are set to the position in which it was found and any THEN clause of the statement is executed. If expression1 is not found within any element of the dynamic array then Vars 2, 3 and 4 are undefined and the ELSE clause of the statement is executed.

NOTES

The statement may omit either the THEN clause or the ELSE clause but may not omit both. It is valid for the statement to contain both clauses if required.
See also: LOCATE, FINDSTR

EXAMPLES

```
Var = "ABC":VM:"JAC":AM:"CDE":VM:"WHO"

FIND "JAC" IN Var SETTING Ap, Vp THEN

    CRT "JAC is in Field ":Ap:", value ":Vp

END ELSE

    CRT "JAC could not be found"

END
```

Will display: JAC is in Field 1, value 2

FINDSTR

The FINDSTR statement locates a string as a substring of a dynamic array element. It is similar in operation to the FIND statement.

COMMAND SYNTAX

FINDSTR expression1 IN Var1 {, expression2} SETTING Var2 {,Var3 {, Var4}} THEN | ELSE statement(s)

SYNTAX ELEMENTS

expression1 evaluates to the string with which to search every element of the dynamic array. **Var1** is the actual dynamic array that will be searched. FINDSTR will normally locate the first occurrence of expression1 unless expression2 is specified. If specified then expression2 will cause a specific occurrence of expression1 to be located. The three variables Var2, Var3, Var4 are used to record the Field, Value and Sub-Value positions in which expression1 was found.
If expression1 is found as a substring of any element of Var1 then Vars 2, 3 and 4 are set to the position in which it was found and the THEN clause of the statement is executed if it is present. If expression1 is not found within any element of the dynamic array then Vars 2,3 and 4 are undefined and the ELSE clause of the statement is executed.

NOTES

The statement may omit either the THEN clause or the ELSE clause but may not omit both. It is valid for the statement to contain both clauses if required.

EXAMPLES

```
Var = "ABC":VM:"OJACKO":AM:"CDE":VM:"WHO"

FINDSTR "JAC" IN Var SETTING Ap, Vp THEN

CRT "JAC is within Field ":Ap:", value ":Vp

END ELSE

 CRT "JAC could not be found"

END
```

Displays: JAC is within Field 1, value 2

FORMLIST

The FORMLIST statement creates an active select list from a dynamic array.

COMMAND SYNTAX

FORMLIST variable1 {TO variable2 | listnum}

SYNTAX ELEMENTS

variable1 specifies the dynamic array from which the active select list is to be created

If **variable2** is specified then the newly created list will be placed in the variable. Alternatively, a select list number in the range 0 to 10 can be specified with listnum. If neither variable2 nor listnum is specified then the default list variable will be assumed.

NOTES

See also: DELETELIST, READLIST, WRITELIST

EXAMPLES

```
MyList = "key1":@AM:"key2":@AM:"key3"

FORMLIST MyList TO ListVar

LOOP
   READNEXT Key FROM ListVar ELSE EXIT

   READ Item FROM Key THEN

   * Do whatever processing is necessary on Item

   END
REPEAT
```

FLUSH

Writes all the buffers for a sequential I/O file immediately. Normally, sequential I/O uses buffering for input/output operations, and writes are not immediately flushed.

COMMAND SYNTAX

FLUSH file.variable {THEN statements [ELSE statements] | ELSE statements}

file.variable specifies a file previously opened for sequential processing. If file.variable evaluates to null, the FLUSH statement fails and the program enters the debugger. After the buffer is written to the file, it executes the THEN statements, ignoring the ELSE statements.

If none of the above can be completed, it executes the ELSE statements.

EXAMPLE

```
OPENSEQ 'DIRFILE', 'RECORD' TO FILE THEN

PRINT "'DIRFILE' OPENED FOR SEQUENTIAL PROCESSING"

END ELSE STOP
WEOFSEQ FILE

*
WRITESEQ 'NEW LINE' ON FILE THEN

FLUSH FILE THEN

PRINT "BUFFER FLUSHED"

END ELSE PRINT "NOT FLUSHED"

ELSE ABORT

*
CLOSESEQ FILE

END
```

FMT

Join lines on U in mask code definition.

Expand on syntax to formatting superset. i.e. we now allow [Width] [Background] [Justification]

INTERNATIONAL MODE

When using the FMT function in International Mode the “Width” fields refer to character display widths, such that a character may take up more than a single display position. This is typical of the Japanese, Chinese, and characters whereby the character display requires possibly two display positions.

Additional date formatting codes have been provided for use in Internationalized programs.

See also: OCONV / FMTS as per FMT

GE - Operator similar to EQ. compares two expressions for greater than or equal

GT - Greater than

GTS - Add as per GES, except just greater than for dynamic array comparison.

INTERNATIONAL MODE

When using the “GE/GT/GES/GTS” function in International Mode, the “operator/function” will use the currently configured locale to determine the rules by which each string is considered greater or equal to the other.

Mask Code	Description
j	Justification R Right Justified L Left Justified U Left Justified, Break on space. Note: This justification will format the output into blocks of data in the variable and it is up to the programmer to actually separate the blocks. D Date (OCONV)
n	Decimal Precision: A number from 0 to 9 that defines the decimal precision. It specifies the number of digits for output following the decimal point. The processor inserts trailing zeros if necessary. If n is omitted or is 0, a decimal point will not be output.
m	Scaling Factor: A number that defines the scaling factor. The source value is descaled (divided) by that power of 10. For example, if m=1, the value is divided by 10; if m=2, the value is divided by 100, and so on. If m is omitted, it is assumed equal to n (the decimal precision).
Z	Suppress leading zeros. NOTE: fractional values, which have no integer, will

Mask Code	Description
	have a zero before the decimal point. If the value is zero, a null will be output.
,	The thousands separator symbol. It specifies insertion of thousands separators every three digits to the left of the decimal point. You can change the display separator symbol by invoking the SET-THOU command. Use the SET-DEC command to specify the decimal separator.
c	Credit Indicator. NOTE: If a value is negative and you have not specified one of these indicators, the value will be displayed with a leading minus sign. If you specify a credit indicator, the data will be output with either the credit characters or an equivalent number of spaces, depending on its value.
	C Prints the literal CR after negative values.
	D Prints the literal DB after positive values.
	E Encloses negative values in angle brackets < >
	M Prints a minus sign after negative values.
	N Suppresses embedded minus sign.
$	Appends a Dollar sign to value.
Fill Character and Length	#n Spaces. Repeat space n times. Output value is overlaid on the spaces created.
	*n Asterisk. Repeat asterisk n times. Output value is overlaid on the asterisks created.
	%n Zero. Repeat zeros n times. Output value is overlaid on the zeros created.
	&x Format. x can be any of the above format codes, a currency symbol, a space, or literal text. The first character following & is used as the default fill character to replace #n fields without data. Format strings are enclosed in parentheses "( )".

EXAMPLES

Format Expression	Source Value (X)	Returned Value (columns) (V) 12345678901234567890 12345678901234567890 12345678901234567890
V = FORMAT(X, "R2#10")	1234.56	1234.56
V = FORMAT(X, "L2%10")	1234.56	1234.56000
V = FORMAT(X, "R2%10")	1234.56	0001234.56
V = FORMAT(X, "L2*10")	1234.56	12.34*****
V = FORMAT(X, "R2*10")	1234.56	*****12.34
V = FORMAT(X, "R2,$#15")	123456.78	$123,456.78
V = FORMAT(X, "R2,&$#15")	123456.78	$$$$$123,456.78

V = FORMAT(X, "R2,& $#15")	123456.78	$ 123,456.78
V = FORMAT(X, "R2,C&*$#15")	-123456.78	$***123,456.78CR
V = FORMAT(X, "R((###) ###-###)")	1234567890	(123) 456-7890
V = FORMAT(X, "R((#3) #2-#4)")	1234567890	(123) 456-7890
V = FORMAT(X, "L& Text #2-#3")	12345	Text 12-345
V = FORMAT(X, "L& ((Text#2) #3)")	12345	(Text12) 345
V = FORMAT(X, "T#20")	This is a test of the American Broadcasting System	This is a test of the American Broadcasting System
V = FORMAT(X, "D4/")	12260	07/25/2001

FMTS

Use the FMTS function to format elements of dynamic.array for output. Each element of the array is independently acted upon and returned as an element in a new dynamic array.

COMMAND SYNTAX

FMTS (dynamic.array, format)

SYNTAX ELEMENTS

format is an expression that evaluates to a string of formatting codes. The Syntax of the format expression is:
[width] [background] justification [edit] [mask]
The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking. For complete syntax details,
See also: FMT function.
If dynamic.array evaluates to null, it returns null. If format evaluates to null, the FMTS function fails and the program enters the debugger.

GE OPERATOR SIMILAR TO eq. compares two expressions for greater than or equal
GT As Above, except Greater than
GTS Add as per GES, except just greater than for dynamic array expression
FMUL/FDIV/FADD/FSUB

FOLD

The FOLD function re-delimits a string by replacing spaces with attribute marks at positions defined by a length parameter.

COMMAND SYNTAX

FOLD(expression1, expression2)

SYNTAX ELEMENTS

expression1 evaluates a string to be re-delimited.
expression2 evaluates to a positive integer that represents the maximum number of characters between delimiters in the resultant string.

NOTES

The FOLD function creates a number of sub-strings such that the length of each sub-string does not exceed the length value in expression2. It converts spaces to attribute marks except when enclosed in sub-strings and removes extraneous spaces.

EXAMPLES

The following examples show how the FOLD function delimits text based on the length parameter. The underscores represent attribute marks.

```
q = "Smoking is one of the leading causes of statistics"

CRT FOLD(q, 7)

Smoking_is one_of the_leading_causes_of_statist_ics

q = "Hello world"

CRT FOLD(q, 5)

Hello_world

q = "Let this be a reminder to you all that this organization will
not

 tolerate failure."

CRT FOLD(q, 30)

let this be a reminder to you_all that this organization_will not

tolerate failure.

q = "the end"
```

```
CRT FOLD(q, 0)

t_h_e_e_n_d
```

FOOTING

The FOOTING statement halts all subsequent output to the terminal at the end of each output page. The statement allows the evaluation and display of an expression at the foot of each page. Output, which is current, and being sent to the terminal, the output is paused until the entry of a carriage return at the terminal (unless the N option is specified either in the current HEADING or in this FOOTING).

COMMAND SYNTAX

FOOTING expression

SYNTAX ELEMENTS

The expression should evaluate to a string, which is printed at the bottom of every page of output. The string could contain a number of interpreted special characters, replaced in the string before printing. The following characters have special meaning within the string:

"C{n}"	center the line, if n is specified the output line is assumed to be n characters long
"D" or \\	replace with the current date
"L" or]	replace with the newline sequence
"N"	terminal output does not pause at the end of each page
"P" or ^	replace with the current page number
"PP" or ^^	replace with the current page number in a field of 4 characters; the field is right justified
"T" or \	replace with the current time and date
"	replace with a single " character

NOTES

If the output is to the printer a PRINTER ON statement is in force; output sent to the terminal with the CRT statement is not paged; if output is to the terminal then all output is paged.

EXAMPLE

FOOTING "Programming staff by weight Page "P"

FOR

The FOR statement allows the construction of looping constructs within the program, which is controlled by a counting variable; this can be terminated early by expressions tested after every iteration.

COMMAND SYNTAX

FOR var=expression1 TO expression2 {STEP expression3} {WHILE | UNTIL expression4}...NEXT {var}

SYNTAX ELEMENTS

var is the counting variable used to control the loop. The first time the loop is entered var is assigned the value of expression1, which must evaluate to a numeric value. After each iteration of the loop, var is automatically incremented by one.

expression2 must also evaluate to a numeric value as it causes the loop to terminate when the value of var is greater than the value of this expression. expression2 is evaluated at the start of every iteration of the loop and compared with the value of expression1.

If the STEP expression3 clause is included within the statement, var will automatically be incremented by the value of expression3 after each iteration of the loop. expression3 is evaluated at the start of each iteration.

expression3 may be negative, in which case the loop will terminate when var is less than expression2.

The statement may optionally include either an evaluated WHILE or UNTIL clause (not both), before each iteration of the loop. When the WHILE clause is specified the loop will only continue with the next iteration if expression4 evaluates to Boolean TRUE. When the UNTIL clause is specified the loop will only continue with the next iteration if expression4 evaluates to Boolean FALSE.

NOTES

Because expression2 and expression3 must be evaluated upon each iteration of the loop, you should only code complex expressions here if they may change within each iteration. If the values they yield will not change then you should assign the value of these expressions to a variable before coding the loop statement. You can replace expressions 3 and 4 with these variables. This can offer large performance increases where complex expressions are in use.

See also: BREAK, CONTINUE.

EXAMPLES

```
Max =DCOUNT(BigVar, CHAR (254))

FOR I = 1 TO Max STEP 2 WHILE BigVar LT 2

5
 BigVar += 1
```

```
NEXT I
```

This example will increment every second field of the variable BigVar but the loop will terminate early if the current field to be incremented is not numerically less than 25.

FSUB

The FSUB function performs floating-point subtraction on two numeric values.

COMMAND SYNTAX

FSUB(expression1, expression2)

SYNTAX ELEMENTS

Both expression1 and expression 2 must evaluate to non-null numeric values.

NOTES

If either of the arguments evaluates to null then a run time "non-numeric" error will occur.

EXAMPLES

```
PRECISION 7
CRT FSUB(2.54,5.703358)

displays -3.163358
```

FUNCTION

Identifies a user-defined function, which can be invoked by other jBASE BASIC programs, arguments to the function can optionally be declared.

COMMAND SYNTAX

FUNCTION name {({MAT} variable, {MAT} variable...) }

SYNTAX ELEMENTS

Name is the name by which the function is invoked.
Variable is an expression used to pass values between the calling program and the function.

NOTES

Use the FUNCTION statement to identify user-written source code functions. Each function must be coded in separate records and the record Id must match that of the Function Name, which in turn should match the reference in the calling program.
The optional comma separated variable list can be a number of expressions that pass values between the calling programs and the function. To pass an array the variable name must be preceded by the MAT keyword. When a user-written function is called, the calling program must specify the same number of variables that are specified in the FUNCTION statement.
An extra 'hidden' variable is used to return a value from the user-written function. The value to be returned can be specified within the Function by the RETURN (value) statement. If using the RETURN statement without a value then by default it returns an empty string.
The calling program must specify a DEFFUN or DEFB statement to describe the function to be called and the function source must be cataloged and locatable similar to subroutines.

EXAMPLE

```
FUNCTION MyFunction(A, B)


    Result = A * B

RETURN (Result)
```

GES

Use the GES function to test if elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.

COMMAND SYNTAX

GES (array1, array2)

SYNTAX ELEMENTS

Compares each element of array1 with the corresponding element of array2, if the element from array1 is greater than or equal to the element from array2, it returns a one in the corresponding element of a new dynamic array. If the element from array1 is less than the element from array2, it returns a zero (0). If an element of one dynamic array has no corresponding element in the other dynamic array, it evaluates the undefined element as empty, and the comparison continues.
If either element of a corresponding pair is null, it returns null for that element.

GET

The GET statement reads a block of data directly from a device.

COMMAND SYNTAX

GET Var {,length} {SETTING Count} FROM Device {UNTIL TermChars} {RETURNING TermChar} {WAITING Timeout} THEN | ELSE statements

SYNTAX ELEMENTS

Var is the variable in which to place the input (from the previously open Device).
If length is specified, it limits the number of characters read from the input device.
If the optional Count option is used, it returns the number of characters actually read from the device.
Device is the file variable associated with the result from a successful OPENSEQ or OPENSER command.
TermChars specifies one or more characters that will terminate input.
TermChar The actual character that terminated input
Timeout is the number of seconds to wait for input. If no input is present when the timeout period expires, the ELSE clause (if specified) is executed.

NOTES

The GET statement does no pre-or post-processing of the input data stream - nor does it handle any terminal echo characteristics. If this is desired, the application - or device drive - will handle it.
If there are no specified length and timeout expressions, the default input length is one (1) character. If no length is specified, but TermChars are, there is no limit to the number of characters input.
The GET syntax requires a specified THEN or ELSE clause, or both. The THEN clause executes when the data received is error free; the ELSE clause executes when the data is unreceiveable (or a timeout occurs).
See: GETX

GETCWD

The GETCWD function allows a jBASE BASIC program to determine the current working directory of the program, which is normally be the directory in which execution of the program occurred but possibly changed using the CHDIR function.

COMMAND SYNTAX

GETCWD(Var)

SYNTAX ELEMENTS

When executed the Var will be set to the name of the current working directory; the function itself returns a Boolean TRUE or FALSE value to indicate whether the command was successful or not.

NOTES

Refer to your UNIX or Windows documentation for more information on the concept of the current working directory.

EXAMPLES

```
IF GETCWD(Cwd) THEN

   CRT "Current Working Directory = ":Cwd

END ELSE

   CRT "Could not determine CWD!"

END
```

GETENV

All processes have an environment associated with them that contains a number of variables indicating the state of various parameters. The GETENV function allows a jBASE BASIC program to determine the value of any of the environment variables associated with it.

COMMAND SYNTAX

GETENV(expression, variable)

SYNTAX ELEMENTS

The expression should evaluate to the name of the environment variable whose value is to be returned. The function will then assign the value of the environment variable to variable. The function itself returns a Boolean TRUE or FALSE value indicating the success or failure of the function.

.

See: PUTENV

EXAMPLE

```
IF GETENV("PATH", ExecPath) THEN

    CRT "Execution path is ":ExecPath

END ELSE


    CRT "Execution path is not set up"

END
```

GETLIST

GETLIST allows the program to retrieve a previously stored list (perhaps created with the SAVE-LIST command), into a jBASE BASIC variable.

COMMAND SYNTAX

GETLIST expression TO variable1 {SETTING variable2} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the variable into which the list will be read. expression should evaluate to the name of a previously stored list to retrieve, or null. If expression evaluates to null, the current default external select list (generated by a previous SELECT command for example) will be retrieved. If specified, **variable2** will be set to the number of elements in the list.
If the statement succeeds in retrieving the list, then the statements associated with any THEN clause will be executed. If the statement fails to find the list, then the statements associated with any ELSE clause will be executed.

NOTES

The GETLIST statement is identical in function to the READLIST statement.
See also: DELETELIST, WRITELIST

EXAMPLES

Find the list first

```
GETLIST "MyList" TO MyList ELSE STOP

LOOP
* Loop until there are no more elements

WHILE READNEXT Key FROM MyList DO

......
REPEAT
```

GETUSERGROUP

For UNIX, the jBASE BASIC GETUSERGROUP function returns the group number for the user ID specified by @uid. For Windows NT or Windows 2000, it returns zero.

COMMAND SYNTAX

GETUSERGROUP(uid)

EXAMPLES

In the following example, the program statement assigns the user group to variable X:

```
X = GETUSERGROUP(@UID)
```

In the next example, the program statement assigns the user group for 1023 to variable X:

```
X = GETUSERGROUP(1023)
```

GETX

The GETX statement reads a block of data (in ASCII hexadecimal format) directly from a device.

COMMAND SYNTAX

GETX Var {,length} {SETTING Count} FROM Device {UNTIL TermChars} {RETURNING TermChar} {WAITING Timeout} THEN | ELSE statements

SYNTAX ELEMENTS

Var is the variable in which to place the input (from the previously open Device).
If specifying a length it limits the number of characters read from the input device.
If the optional Count option is used, it returns the number of characters actually read from the device.
Device is the file variable associated with the result from a successful OPENSEQ or OPENSER command.
TermChars specifies one or more characters that will terminate input.
TermChar The actual character that terminated input
Timeout is the number of seconds to wait for input. If no input is present when the timeout period expires, the ELSE clause (if specified) is executed.

NOTES

The GETX statement does no pre-or post-processing of the input data stream nor does it handle any terminal echo characteristics. It is assumed that if this is desired the application - or device drive - will handle it.
If there are no specified length and timeout expressions, the default input length is one (1) character. If there is no length specified, but TermChars are, there is no limit to the number of characters input.
The GETX syntax requires a specified THEN or ELSE clause, or both. The THEN clause executes when the data received is error free; the ELSE clause executes when the data is unreceiveable (or a timeout occurs).

GETX will convert all input into ASCII hexadecimal format after input.
See also: GET

GOSUB

The GOSUB statement causes execution of a local subroutine, after which execution will continue with the next line of code.

COMMAND SYNTAX

GOSUB label

SYNTAX ELEMENTS

The label should refer to an existent label within the current source code, which identifies the start of a local subroutine.

EXAMPLES

```
GOSUB Initialize ;* open files etc..

GOSUB Main ;* perform main program

GOSUB Finish ;* close files etc..

STOP

...

Initialize: * open files


OPEN......

.
RETURN

....

Main: * main execution loop

......
RETURN

Finish: * clean up after execution

......
RETURN
```

GOTO

The GOTO statement causes program execution to jump to the code at a specified label.

COMMAND SYNTAX

GO{TO} Label

SYNTAX ELEMENTS

The label should refer to an existing label within the current source code.

NOTES

Warning: using the GOTO command obscures the readability of the code and is a hindrance to maintainability. All programs written using the GOTO construct can be written using structured statements such as LOOP and FOR. There are various opinions on this issue but the consensus is, avoid GOTO.
One possibly acceptable use of the GOTO statement is to transfer execution to an error handler upon detection of a fatal error that will cause the program to terminate.

EXAMPLE

```
GOTO Exception;* jump to the exception handler
.....
Exception:* exception handler
....STOP
```

GROUP

The GROUP function is equivalent to the FIELD function.

COMMAND SYNTAX

GROUP(Expression1, Expression2, Expression3, Expression4)

SYNTAX ELEMENTS

Expression1 evaluates to the string containing fields to be extracted.

Expression2 evaluates to the character(s) delimiting each field within Expression1.

Expression3 should evaluate to a numeric value specifying the number of the first field to extract from Expression1.

Expression4 evaluates to a numeric value specifying the number of fields to extract as a group.

NOTES

Expression2 may evaluate to more than a single character allowing fields to be delimited with complex expressions.

EXAMPLES

```
A = "123:-456:-789:-987:-"

CRT GROUP(A, ":-", 2, 2)
```

This example displays:

456:-789

on the terminal being the second and third fields and their delimiter within variable A

HEADING

Heading halts all subsequent output to the terminal at the end of each page. The statement evaluates and displays an expression at the top of each page. Current output sent to the terminal, is paused until entry of a carriage return at the terminal - unless the N option is specified.

COMMAND SYNTAX

HEADING expression

SYNTAX ELEMENTS

The expression should evaluate to a string printed at the top of every page of output. The string may contain a number of interpreted special characters, replaced in the string before printing. The following characters have special meaning within the string:

"C{n}"	Center the line. If n is specified the output line is assumed n characters long.
"D" or \\	Replace with the current date.
"L" or]	Replace with the newline sequence.
"N"	Terminal output does not pause at the end of each page.
"P" or ^	Replace with the current page number.
"PP" or ^^	Replace with the current page number in a field of 4 characters. The field is right justified.
"T" or \	Replace with the current time and date.
"	Replace with a single " character.

NOTES

If output is to the printer, a PRINTER ON statement is in use, and does not page output sent to the terminal with the CRT statement. Unless you specify the “N” option, all output sent to the terminal is paged.

EXAMPLES

HEADING "Programming staff by size of waist Page "P"

HEADINGE and HEADINGN

The HEADINGE statement is the same as the HEADING statement, which causes a page eject with the HEADING statement.

The HEADINGN statement is the same as the HEADING statement, and suppresses the page eject.

HUSH

Use the HUSH statement to suppress the display of all output normally sent to a terminal during processing. HUSH also suppresses output to a COMO file.

HUSH acts as a toggle. If it is used without a qualifier, it changes the current state. Do not use this statement to shut off output display unless you are sure the display is unnecessary. When you use HUSH ON, all output is suppressed including error messages and requests for information.

COMMAND SYNTAX

HUSH { ON | OFF | expression }

EXAMPLE

HUSH ON

ICONV

The ICONV function converts data in external form such as dates to their internal form.

COMMAND SYNTAX

ICONV(expression1, expression2)

SYNTAX ELEMENTS

expression1 evaluates to the data upon which the conversion is to be performed.

expression2 should evaluate to the conversion code that is to be performed against the data.

Add additional ICONV extensions for timestamp as per WDx/WTx

NOTES

If the conversion code used assumes a numeric value and a non-numeric value is passed then the original value in expression1 is returned unless the emulation option iconv_nonnumeric_return_null is set.

EXAMPLES

InternalDate = ICONV("27 MAY 1997", "D")

In this example, ICONV returns the internal form of the date May 27, 1997.

ICONVS

Use ICONVS to convert each element of dynamic.array to a specified internal storage format.

COMMAND SYNTAX

ICONVS (dynamic.array, conversion)

SYNTAX ELEMENTS

conversion is an expression that evaluates to one or more valid conversion codes, separated by value marks (ASCII 253).

Each element of **dynamic.array** is converted to the internal format specified by conversion and is returned in a dynamic array. If multiple codes are used, they are applied from left to right. The first conversion code converts the value of each element of dynamic.array. The second conversion code converts the value of each element of the output of the first conversion, and so on. If dynamic.array evaluates to null, it returns null. If an element of dynamic.array is null, null it returns null for that element. If conversion evaluates to null, the ICONV function fails and the program terminates with a run-time error message.

The STATUS function reflects the result of the conversion:

For information about converting elements in a dynamic array to an external format

See also: OCONVS function.

0	The conversion is successful.
1	An element of dynamic.array is invalid. It returns an empty string, unless dynamic.array is null, in which case it returns null.
2	Conversion is invalid.
3	Successful conversion of possibly invalid data.

IF (statement)

Allows other statements to be conditionally executed

COMMAND SYNTAX

IF expression THEN|ELSE statements

SYNTAX ELEMENTS

It evaluates the expression to a value of Boolean TRUE or FALSE. If the expression is TRUE executes then the statements defined by the THEN clause (if present). If the expression is FALSE executes the statements defined by the ELSE clause.
The THEN and ELSE clauses may take two different forms being single and multiple line statements. The simplest form of either clause is of the form:

```
IF A THEN CRT A

or

IF A ELSE CRT A
```

However, expand the clauses to enclose multiple lines of code using the END keyword as so:

```
IF A THEN

    A = A*6

    CRT A

END ELSE

    A = 76

    CRT A

END
```

You can combine the single and multi-line versions of either clause to make complex combinations of the command. For reasons of readability it is suggested that where both clauses are present for an IF statement that the same form of each clause is coded.

NOTES

IF statements can be nested within either clause to any number of levels

EXAMPLE

```
CRT "Are you sure (Y/N) ":

INPUT Answer,1_
```

```
IF OCONV (Answer, "MCU")= "Y" THEN
   GOSUB DeleteFiles
    CRT "Files have been deleted"
END ELSE
    CRT "File delete was ignored"
END
```

IFS

Use the IFS function to return a dynamic array whose elements are chosen individually from one of two dynamic arrays based on the contents of a third dynamic array.

COMMAND SYNTAX

IFS (dynamic.array, true.array, false.array)

IFS evaluate each element of the dynamic.array. If the element evaluates to true, it returns the corresponding element from true.array to the same element of a new dynamic array. If the element evaluates to false, it returns the corresponding element from false.array. If there is no corresponding element in the correct response array, it returns an empty string for that element. If an element is null, that element evaluates to false.

IN

The IN statement allows the program to receive raw data from the input device, which is normally the terminal keyboard, one character at a time.

COMMAND SYNTAX

IN Var {FOR expression THEN|ELSE statements}

SYNTAX ELEMENTS

Var will be assigned the numeric value (0 - 255 decimal) of the next character received from the input device. The statement will normally wait indefinitely (block) for a character from the keyboard. Specifying the FOR clause to the IN statement allows the statement to stop waiting for keyboard after a specified amount of time. The expression should evaluate to a numeric value, which will be taken as the number of deci-seconds (tenths of a second) to wait before abandoning the input.
The **FOR** clause must have either or both of the THEN or ELSE clauses If a character is received from the input device before the time-out period then Var is assigned its numeric value and the THEN clause is executed (if present). If the input statement times out before a character is received then Var is unaltered and the ELSE clause is executed (if present).

NOTES

See also: INPUT, INPUTNULL.

EXAMPLES

```
Char2 = "

IN Char

IF Char = 27 THEN ;* ESC seen

    IN Char2 FOR 20 THEN ;* Function Key?

        Char2 = CHAR(Char2) ;* ASCII value


    END
END
Char = CHAR(Char):Char2 ;* Return key sequence
```

INDEX

The INDEX function will return the position of a character or characters within another string.

COMMAND SYNTAX

INDEX(expression1, expression2, expression3)

SYNTAX ELEMENTS

expression1 evaluates to the string to be searched.

expression2 evaluates to the string or character that will be searched for within expression1.

expression3 should evaluate to a numeric value and specify which occurrence of expression2 should be searched for within expression1.

NOTES

If the specified occurrence of expression2 is not found in expression1 then it returns Zero (0).

EXAMPLE

```
ABet = "abcdefghijklmnopqrstuvwxyzabc"

CRT INDEX(ABet, "a", 1)

CRT INDEX(ABet, "a", 2

)
CRT INDEX(ABet, "jkl", 1)
```

The above code will display:

1

27

10

INMAT

The INMAT() function returns the number of dimensioned array elements.

COMMAND SYNTAX

INMAT({array})

DESCRIPTION

Using the INMAT() function, without the 'array' argument, returns the number of dimensioned array elements from the most recent MATREAD, MATREADU, MATREADL or MATPARSE statement. If the number of array elements exceeds the number of elements specified in the corresponding DIM statement, the INMAT() function will return zero.
Using the INMAT(), function with the 'array' argument, returns the current number of elements to the dimensioned 'array'.

NOTES

In some dialects the INMAT() function is also used to return the modulo of a file after the execution of an OPEN statement, which is inconsistent with its primary purpose and not implemented in jBASE. To achieve this functionality use the IOCTL() function with the JIOCTL_COMMAND_FILESTATUS command.

EXAMPLE

```
OPEN "CUSTOMERS" TO CUSTOMERS ELSE STOP 201, "CUSTOMERS"

DIM CUSTREC(99)

ELEMENTS = INMAT(CUSTREC) ; * Returns the value "99" to the variable
ELEMENTS

ID = "149"

MATREAD CUSTREC FROM CUSTOMERS, ID THEN

    CUSTREC.ELEMENTS = INMAT() ; * Returns the number of elements in
the CUSTRECarray to the variable CUSTREC.ELEMENTS

END
```

INPUT

The INPUT statement allows the program to collect data from the current input device, which will normally be the terminal keyboard but may be stacked input from the same or separate program.

COMMAND SYNTAX

INPUT {@ (expression1 {, expression2)}{:} Var{{, expression3}, expression4} {:}{_} {WITH expression5} {FOR expression6 THEN|ELSE statements}

SYNTAX ELEMENTS

@(expression1, expression2) allows the screen cursor to be positioned to the specified column and row before the input prompt is sent to the screen. The syntax for this is the same as the @( ) function described earlier.

Var is the variable in which the input data is to be stored.

expression3, when specified, should evaluate to a numeric value. This will cause input to be terminated with an automatic newline sequence after exactly this number of characters has been input. If the _ option is specified with expression4 then the automatic newline sequence is not specified but any subsequent input characters are belled to the terminal and thrown away.

expression4 when specified, should evaluate to a sequence of 1 to 3 characters. The first character will be printed expression3 times to define the field on the terminal screen. At the end of the input if less than expression3 characters were input then the rest of the field is padded with the second character if it was supplied. If the third character is supplied then the cursor will be positioned after the last character input rather than at the end of the input field.

The : option, when specified, suppress the echoing of the newline sequence to the terminal. This will leave the cursor positioned after the last input character on the terminal screen.

WITH expression5 allows the default input delimiter (the newline sequence) to be changed. When specified, expression5, should evaluate to a string of up to 256 characters, each of which may delimit the input field. If this clause is used then the newline sequence is removed as a delimiter and must be specified explicitly within expression5 as CHAR(10).

The "FOR" clause allows the "INPUT" statement to time out after a specified waiting period instead of blocking as normal Expression6 should evaluate to a numeric value, which will be taken as the number of deci-seconds (tenths of a second) to wait before timing out. The time-out value is used as the time between each keystroke and should a time-out occur, Var would hold the characters that were input until the time-out.

The FOR clause requires either the THEN and ELSE clauses or both; if no time-out occurs the THEN clause is taken. If a time-out does occur, the ELSE clause is taken.

NOTES

The INPUT statement will always examine the data input stack before requesting data from the input device. If data is present on the stack then it is used to satisfy INPUT statements one field at a time

until the stack is exhausted. Once exhausted, the INPUT statement will revert to the input device for further input. There is no way (by default) to input a null field to the INPUT@ statement. If the INPUT@ statement receives the newline sequence only as input, then the Var will be unchanged. Use the INPUTNULL statement to define a character that indicates a NULL input.
Use the CONTROL-CHARS command to control whether or not control characters (i.e. those outside the range x'1F' - x'7F') are accepted by INPUT.
See also: IN, INPUTNULL.

EXAMPLES

```
Answer = "

LOOP
WHILE Answer = " DO

    INPUT Answer,1 FOR 10 ELSE

        GOSUB UpdateClock

 END
REPEAT
```

The above example attempts to read a single character from the input device for 10 deci-seconds (1 second). The LOOP will exit when a character has been input otherwise every second it will call the local subroutine UpdateClock.

INPUTCLEAR

The INPUTCLEAR statement clears the type-ahead buffer.

COMMAND SYNTAX

```
INPUTCLEAR
```

SYNTAX ELEMENTS

None

NOTES

INPUTCLEAR only clears the type-ahead buffer. It does not clear data stacked with the DATA statement.

The INPUTCLEAR statement is synonymous with CLEARINPUT.

EXAMPLE

```
CRT "Start year end processing (Yes/No) :"
INPUTCLEAR
INPUT ans
IF ans # "Yes" THEN
CRT "year end processing not started"
END
```

INPUTNULL

The INPUTNULL statement allows the definition of a character that will allow a null input to be seen by the INPUT@ statement.

COMMAND SYNTAX

INPUTNULL expression

SYNTAX ELEMENTS

The **expression** should evaluate to a single character. Subsequently, any INPUT@ statement that sees only this character input before the new-line sequence will NULL the variable in which input is being stored.
If expression evaluates to the NULL string " then the default character of _ is used to define a NULL input sequence.

NOTES

The INPUT statement does not default to accepting the _ character as a NULL input, the programmer must explicitly allow this with the statement: INPUTNULL "

EXAMPLES

```
INPUTNULL "&"

INPUT @ (10,10):Answer,1

IF Answer = " THEN

   CRT "A NULL input was received"

END
```

INS

The INS statement allows the insertion of elements into a dynamic array.

COMMAND SYNTAX

INS expression BEFORE Var<expression1{, expression2{, expression3}}>

SYNTAX ELEMENTS

expression evaluates to the element to be inserted in the dynamic array.
expression1 expression2 and expression3 should all evaluate to numeric values and specify the Field, Value and Sub-Value before which the new element is to be inserted.

NOTES

Specifying a negative value to any of the expressions 1 through 3 will cause the element to append as the last Field, Value or Sub-Value rather than at a specific position. Only one expression may be negative otherwise only the first negative value is used correctly while the others are treated as the value 1.
The statement will insert NULL Fields, Values or Sub-Values accordingly if any of the specified insertion points exceeds the number currently existing.

EXAMPLE

```
Values = ''

FOR I = 1 TO 50

    INS I BEFORE Values<-1>

NEXT I

FOR I = 2 TO 12

    INS I*7 BEFORE Values<7,i>

NEXT I
```

INSERT

INSERT is the function form of the INS statement, with preference given to the use of INS.

COMMAND SYNTAX

INSERT(expression1, expression2{, expression3 {, expression4 }}; expression5)

SYNTAX ELEMENTS

expression1 evaluates to a dynamic array in which to insert a new element and will normally be a variable.

expression2 expression3 and expression4 should evaluate to numeric values and specify the Field, Value and Sub-Value before which the new element will be inserted.

expression5 evaluates to the new element to be inserted in expression1.

EXAMPLES

```
A = INSERT(B, 1,4; "Field1Value4")
```

INT

The INT function truncates a numeric value into its nearest integer form.

COMMAND SYNTAX

INT(expression)

SYNTAX ELEMENTS

expression should evaluate to a numeric value. The function will then return the integer portion of the value.

NOTES

The function works by truncating the fractional part of the numeric value rather than by standard mathematical rounding techniques. Therefore, INT(9.001) and INT(9.999) will both return the value 9.

EXAMPLES

```
CRT INT(22/7)
```

Displays the value 3

IOCTL

The jBASE BASIC language provides an intrinsic function called IOCTL that behaves in a similar manner to the C function ioctl(). Its purpose is to allow commands to be sent to the database driver for a particular file, and then to receive a reply from the database driver.
As with the C function ioctl, the use of IOCTL is highly dependent upon the database driver it is talking to. Each database driver may choose to provide certain common functionality, or may add its own commands and so on. This is especially true of user-written database drivers.
First, an example of a source program that opens a file and finds the type of file:

```
INCLUDE JBC.h
OPEN "MD" TO DSCB ELSE STOP 201,"MD"
status=""
IF IOCTL(DSCB,JIOCTL_COMMAND_FILESTATUS,status) THEN
  PRINT "Type of file = ":DQUOTE(status<1>)
END ELSE
  PRINT "IOCTL FAILED !! unknown file type"
END
```

If the ELSE clause is taken, it does not necessarily mean there is an error, it only means that the database driver for file "MD" does not support the command that was requested from it. The file JBC.h is supplied with jBASE in the directory JBCRELEASEDIR sub directory include. If the source is compiled with the jbc or BASIC command, this directory is automatically included in the search path and no special action is needed by the programmer for the "INCLUDE JBC.h" statement.

The format of the IOCTL function is:
IOCTL(Filevar, Command, Parameter)

Where:
filevar Is a variable that has had a file opened against it using the OPEN statement. However, if you want to use the default file variable, use -1 in this position. For example:

```
OPEN "MD" ELSE STOP
filevar = -1
IF IOCTL(filevar,JIOCTL_COMMAND_xxx,status) ...
```

command can be any numeric value (or variable containing a numeric). However, it is up to the database driver to support that particular command number. The remainder of this chapter describes the common IOCTL command numbers supported by the jBASE database drivers provided.

Status Pass here a jBASE BASIC variable. The use of this variable depends upon the command parameter, and will be described later for each command supported.
The return value is 0 for failure, or 1 for success. A value of -1 generally shows the command has not been recognized.
The remainder of this section will deal with the IOCTL commands that are supported by the provided jBASE database drivers, and the JBC_COMMAND_GETFILENAME command that is supported for all database drivers.

JBC_COMMAND_GETFILENAME COMMAND

Using this command to the IOCTL function, you can determine the exact file name that was used to open the file. This is helpful because jEDI uses Q pointers, F pointers and the JEDIFILEPATH environment variable to actually open the file, and the application can never be totally sure where the resultant file was really opened. Normally of course, this is of no concern to the application.

EXAMPLE

Open the file CUSTOMERS and find out the exact path that was used to open the file.

```
INCLUDE JBC.h
OPEN "CUSTOMERS" TO DSCB ELSE STOP 201,"CUSTOMERS"
filename = ""
IF IOCTL(DSCB,JBC_COMMAND_GETFILENAME,filename) ELSE
   CRT "IOCTL failed !!" ; EXIT(2)
END
PRINT "Full file path = ":DQUOTE(filename)
```

This command is executed by the jBASE BASIC library code rather than the jEDI library code or the database drivers, so it can be run against a file descriptor for any file type.

JIOCTL_COMMAND_CONVERT COMMAND

Some of the jBASE BASIC database drivers will perform an automatic conversion of the input and output record when performing reads and writes.
An example of this is when writing to a directory. In this case, the attribute marks will be converted to new-line characters and a trailing new-line character added. Similarly for reading from a directory the new-line characters will be replaced with attribute marks, and the trailing new-line character will be deleted.
The above example is what happens for the database driver for directories. It assumes by default that the record being read or written is a text file and that the conversion is necessary. It tries to apply some intelligence to reading files, as text files always have a trailing new-line character. Therefore, if a file is read without a trailing new-line character, the database driver assumes the file must be a binary file rather than a text file, and no conversion takes place.

This conversion of data works in most cases and usually requires no special intervention from the programmer.
There are cases however, when this conversion needs to be controlled and interrogated, and the IOCTL function call with the JIOCTL_COMMAND_CONVERT command provides the jBASE database drivers that support this conversion with commands to control it.
The call to IOCTL, if successful, will only affect file operations that use the same file descriptor.
Consider the following code:

```
INCLUDE JBC.h
OPEN "MD" TO FILEVAR1 ELSE ...
OPEN "MD" TO FILEVAR2 ELSE ...
IF IOCTL(FILEVAR1,JIOCTL_COMMAND_CONVERT,"RB")
```

In the above example, any future file operations using variable FILEVAR1 will be controlled by the change forced in the IOCTL request. Any file operations using variable FILEVAR2 will not be affected and will use the default file operation.
Input to the IOCTL is a string of controls delimited by a comma that tell the database driver what to do.
The output from the IOCTL can optionally be a string to show the last conversion that the driver performed on the file.
The descriptions of the available controls that can be passed as input to this IOCTL function are:

Code	Description
RB	All future reads to be in binary (no conversion)
RT	All future reads to be in text format (always do a conversion)
RI	All future reads to decide themselves whether binary or text
RS	Return to caller the status of the last read ("B" = binary, "T" = text)
WB	All future writes to be in binary (no conversion)
WT	All future writes to be in text format (always do a conversion)
WI	All future writes to decide themselves whether binary or text
WS	Return to caller the status of the last write ("B" = binary, "T" = text)
KB	All future reads/writes have the record key unaltered
KT	All future reads/writes have the record key modified
KI	All future reads/writes to decide if to do a conversion
KS	Return to caller the status of the last record key ("B" = binary, "T" = text)

EXAMPLE 1

The application wants to open a file, and to ensure that all reads and writes to that file are in binary, and that no translation such as new-lines to attribute marks is performed.

```
INCLUDE JBC.h
OPEN "FILE" TO DSCB ELSE STOP 201,"FILE"
IF IOCTL(DSCB,JIOCTL_COMMAND_CONVERT,"RB,WB") ELSE
  CRT "UNABLE TO IOCTL FILE 'FILE'" ; EXIT(2)
END
```

EXAMPLE 2

Read a record from a file, and find out if the last record read was in text format (were new-lines converted to attribute marks and the trailing new-line deleted), or in binary format (with no conversion at all).

```
INCLUDE JBC.h
OPEN "." TO DSCB ELSE STOP 201,"."
READ rec FROM DSCB,"prog.o" ELSE STOP 202,"prog.o"
status = "RS"
IF IOCTL(DSCB,JIOCTL_COMMAND_CONVERT,status) THEN
  IF status EQ "T" THEN CRT "TEXT" ELSE CRT "BINARY"
END ELSE
  CRT "The IOCTL failed !!"
END
```

JIOCTL_COMMAND_FILESTATUS COMMAND

The JIOCTL_COMMAND_FILESTATUS command will return an attribute delimited list of the status of the file to the caller.

Attribute	Description
<1>	File type, as a string
<2>	FileFlags, as decimal number, show LOG, BACKUP and TRANS
<3>	BucketQty, as decimal number, number of buckets in the file
<4>	BucketSize, as decimal number, size of each bucket in bytes
<5>	SecSize, as decimal number, size of secondary data space
<6>	Restore Spec, a string showing any restore re-size specification
<7>	Locking identifiers, separated by multi-values
<8>	FileFlags showing LOG, BACKUP and TRANSACTION permissions

<8,1>	Set to non-zero to suppress logging on this file
<8,2>	Set to non-zero to suppress transaction boundaries on this file
<8,3>	Set to no-zero to suppress backup of the file using jbackup
<9>	Hashing algorithm used

EXAMPLE

```
Open a file and see if the file type is a directory.

INCLUDE JBC.h
OPEN ".." TO DSCB ELSE STOP 201,".."
status = ""
IF IOCTL(DSCB,JIOCTL_COMMAND_FILESTATUS,status) ELSE
   CRT "IOCTL failed !!" ; EXIT(2)
END
IF status<1> EQ "UD" THEN
   PRINT "File is a directory"
END ELSE
   PRINT "File type is ":DQUOTE(status<1>)
   PRINT "This is not expected for .."
END
```

EXAMPLE 2

Open a file ready to perform file operations in a transaction against it. Make sure the file has not been removed as a transaction type file by a previous invocation of the command "jchmod

```
-T CUSTOMERS".

INCLUDE JBC.h
OPEN "CUSTOMERS" TO DSCB ELSE STOP 201,"CUSTOMERS"
IF IOCTL(DSCB,JIOCTL_COMMAND_FILESTATUS,status) ELSE
   CRT "IOCTL failed !!" ; EXIT(2)
END
IF status<8,2> THEN
   CRT "Error ! File CUSTOMERS is not"
   CRT "part of transaction boundaries !!"
   CRT "Use "jchmod +T CUSTOMERS" !!"
   EXIT(2)
END
```

JIOCTL_COMMAND_FINDRECORD COMMAND

This command will find out if a record exists on a file without the need to actually read in the record. This can provide large performance gains in certain circumstances.

EXAMPLE

Before writing out a control record, make sure it doesn't already exist. As the control record is quite large, it will provide performance gains to simply test if the output record already exists, rather than reading it in using the READ statement to see if it exists.

```
INCLUDE JBC.h
OPEN "outputfile" TO DSCB ELSE STOP 201,"outputfile"
... Make up the output record to write out in "output"
key = "output.out"
rc = IOCTL(DSCB,JIOCTL_COMMAND_FINDRECORD,key)
BEGIN CASE
  CASE rc EQ 0
    WRITE output ON DSCB,key
    CRT "Data written to key " : key
  CASE rc GT 0
    CRT "No further action, record already exists"
  CASE 1
    CRT "IOCTL not supported for file type"
END CASE
```

JIOCTL_COMMAND_FINDRECORD_EXTENDED COMMAND

This command to the IOCTL function returns the record size and the time and date the record was last updated. If the record does not exist, null is returned. The time/date stamp is returned in UTC format.

EXAMPLE

Print the time and data of last update for each record in filename.

```
INCLUDE JBC.h
OPEN "filename" TO DSCB ELSE STOP 201,"filename"
*
* Select each record in the newly opened file
*
SELECT DSCB
LOOP WHILE READNEXT record.key DO
*
* Get the details on the record and look for errors.
*
  record.info = record.key
```

```
  IF IOCTL(DSCB,JIOCTL_COMMAND_FINDRECORD_EXTENDED,record.info) ELSE
    CRT "Error! File driver does not support this"
    STOP
  END
*
* Extract and convert the returned data
*
  record.size = record.info<1>
  record.utc = record.info<2>
  record.time = OCONV(record.utc,"U0ff0")
  record.date = OCONV(record.utc,"U0ff1")
*
* Print the information.
*
  PRINT "Record key ":record.key:" last updated at ":
  PRINT OCONV(record.time,"MTS"):" ":
  PRINT OCONV(record.date,"D4")
REPEAT
```

JIOCTL_COMMAND_HASH_RECORD COMMAND

For jBASE hashed files such as j3 and j4 each record is pseudo-randomly written to one of the buckets (or groups) of the hashed file. The actual bucket it is written to depends upon two factors:

The actual record key (or item-id)
The number of buckets in the file (or modulo)

This IOCTL command shows which bucket number the record would be found in, given the input record key. The bucket number is in the range 0 to (b-1) where b is the number of buckets in the file specified when the file was created (probably using CREATE-FILE).

The command only returns the expected bucket number, as is no indication that the record actually exists in the file.

Two attributes are returned by this command. The first is the hash value that the record key has hashed to, and the second attribute is the bucket number.

EXAMPLE

Open a file, and find out what bucket number the record "PIPE&SLIPPER" would be found in.

```
INCLUDE JBC.h
OPEN "WEDDING-PRESENTS" TO DSCB ELSE STOP
key = "PIPE&SLIPPER"
parm = key
IF IOCTL(DSCB,JIOCTL_COMMAND_HASH_RECORD,parm) THEN
```

```
  PRINT "key ":key:" would be in bucket ":parm<2>
END ELSE
  CRT "IOCTL failed, command not supported"
END
```

JIOCTL_COMMAND_HASH_LOCK COMMAND

The jEDI locking mechanism for records in jEDI provided database drivers is not strictly a 100% record locking mechanism. Instead, it uses the hashed value of the record key to give a value from 0 to 230-1 to describe the record key. The IOCTL command can be used to determine how a record key would be converted into a hashed value for use by the locking mechanism.

EXAMPLE

Lock a record in a file and find out what the lock id of the record key is. The example then calls the jRLA locking demon and the display of locks taken should include the lock taken by this program.

```
INCLUDE JBC.h
DEFCE getpid()
OPEN "WEDDING-PRESENTS" TO DSCB ELSE STOP
key = "PIPE&SLIPPER"
parm = key
IF IOCTL(DSCB,JIOCTL_COMMAND_HASH_LOCK,parm) ELSE
  CRT "IOCTL failed, command not supported"
  EXIT(2)
END
PRINT "The lock ID for the key is ":parm
PRINT "Our process id is " : getpid()
```

ISALPHA

The ISALPHA function will check that the expression consists of entirely alphabetic characters.

COMMAND SYNTAX

ISALPHA(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISALPHA function will then return TRUE (1) if the expression consists of entirely alphabetic characters. The function will return FALSE (0) if any character in the expression is not alphabetic.

INTERNATIONAL MODE

When the ISALPHA function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISALNUM

The ISALNUM function will check that the expression consists of entirely alphanumeric characters.

COMMAND SYNTAX

ISALNUM(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISALNUM function will then return TRUE (1) if the expression consists of entirely alphanumeric characters. The function will return FALSE (0) if the expression contains any characters, which are not alphanumeric.

INTERNATIONAL MODE

When the ISALNUM function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISCNTRL

The ISCNTRL function will check that the expression consists entirely of control characters.

COMMAND SYNTAX

ISCNTRL(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISCNTRL function will then return TRUE (1) if the expression consists of entirely control characters. The function will return FALSE (0) if the expression contains any characters, which are not control characters.

INTERNATIONAL MODE

When the ISCNTRL function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISDIGIT

The ISDIGIT function will check that the expression consists of entirely numeric characters.

COMMAND SYNTAX

ISDIGIT(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISDIGIT function will then return TRUE (1) if the expression consists of entirely numeric characters. The function will return FALSE (0) if the expression contains any characters, which are not numeric.

INTERNATIONAL MODE

When the ISDIGIT function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISLOWER

The ISLOWER function will check that the expression consists of entirely lower case characters.

COMMAND SYNTAX

ISLOWER(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISLOWER function will then return TRUE (1) if the expression consists of entirely lower case characters. The function will return FALSE (0) if the expression contains any characters, which are not lower case characters.

INTERNATIONAL MODE

When the ISLOWER function is used in International Mode the properties of each character is determined according to the Unicode Standard

ISPRINT

The ISPRINT function will check that the expression consists of entirely printable characters.

COMMAND SYNTAX

ISPRINT(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISPRINT function will then return TRUE (1) if the expression consists of entirely printable characters. The function will return FALSE (0) if the expression contains any characters, which are not printable.

INTERNATIONAL MODE

When the ISPRINT function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISSPACE

The ISSPACE function will check that the expression consists of entirely space type characters.

COMMAND SYNTAX

ISSPACE(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISSPACE function will then return TRUE (1) if the expression consists of entirely spacing type characters. The function will return FALSE (0) if the expression contains any characters, which are not space characters.

INTERNATIONAL MODE

When the ISSPACE function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ISUPPER

The ISUPPER function will check that the expression consists of entirely upper case characters.

COMMAND SYNTAX

ISUPPER(expression)

SYNTAX ELEMENTS

The expression can return a result of any type. The ISUPPER function will then return TRUE (1) if the expression consists of entirely lower case characters. The function will return FALSE (0) if the expression contains any characters, which are not upper case characters.

INTERNATIONAL MODE

When the ISUPPER function is used in International Mode the properties of each character is determined according to the Unicode Standard.

ITYPE

Use the ITYPE function to return the value resulting from the evaluation of an I-type expression in a jBASE file dictionary.

COMMAND SYNTAX

ITYPE (i.type)

I.type is an expression evaluating to the contents of the compiled I-descriptor. You must compile the I-descriptor before the ITYPE function uses it; otherwise, you get a run-time error message.
Using several methods set the I.type to the evaluated I-descriptor in several ways. One way is to read the I-descriptor from a file dictionary into a variable, then use the variable as the argument to the ITYPE function. If the I-descriptor references a record ID, the current value of the system variable **@ID** is used. If the I-descriptor, references field values in a data record, the data is taken from the current value of the system variable @RECORD.
To assign field values to @RECORD, read a record from the data file into @RECORD before invoking the ITYPE function.
If i.type evaluates to null, the ITYPE function fails and the program terminates with a run-time error message.
NOTE: Set the @FILENAME to the name of the file before ITYPE execution.

EXAMPLE

This is the SLIPPER file content:

JIM	GREG	ALAN
001 8	001 10	001 5

This is the DICT SLIPPER content:

```
SIZE
001 D
002 1
003
004
005 10L
006 L
```

This is the program source code:

```
OPEN 'SLIPPERS' TO FILE ELSE STOP
OPEN 'DICT','SLIPPERS' TO D.FILE ELSE STOP
```

```
*
READ ITYPEDESC FROM D.FILE, 'SIZE' ELSE STOP
*
EXECUTE 'SELECT SLIPPERS'
@FILENAME = “SLIPPERS”
LOOP
READNEXT @ID DO
*
READ @RECORD FROM FILE, @ID THEN
*
PRINT @ID: "WEARS SLIPPERS SIZE " ITYPE(ITYPEDESC)
END
REPEAT
```

The output of this program is:

```
3 records selected
JIM WEARS SLIPPERS SIZE 8
GREG WEARS SLIPPERS SIZE 10
ALAN WEARS SLIPPERS SIZE 5
```

JBASECOREDUMP

Use as a diagnostic tool for applications and allows a snapshot of the application to be dumped to an external file for later analysis

COMMAND SYNTAX

JBASECOREDUMP(expression1, expression2)

SYNTAX ELEMENTS

For jBASE 4.1 upwards only, the program variables and CALL/GOSUB stack will be dumped.

The output is in free style text format.

The function is called such:

```
PRINT " fatal application error, outputting a core dump"
filename = "GLOBUSDUMP_":TIME():"_":DATE():"_":SYSTEM(18)
PRINT "Please send the file ":filename:" to your Temenos support"
dummy = JBASECOREDUMP(filename , 0 )
EXIT(99)
```

The first parameter shows the name of the operating system file to output the core dump to. You can supply "" instead of a file name and jBASE allocates a filename of:

/JBASECOREDUMP_nnnn_mmmmm
where
nnn is the port number and mmmmmm is the process id.
The second parameter is not used at present. Future versions will allow extra information to be selectively dumped.
A null string is always returned from the function.

EXAMPLE

```
jBASE Core dump created at Thu Apr 10 17:12:01 2003

Program test31 , port 0 , process id 21959
```

CALL/GOSUB stack

```
Line 0 , Source jmainfunction.b , Level 0

Source changed to ./test31.b

0007     GOSUB 100
```

```
0012     GOSUB 200

0016     CALL SUB1

Source changed to ./SUB1.b

0004     GOSUB 100

0009     GOSUB 200
```

All the defined VAR's in the program

COMMON variables

```
  0x8057dd0 : greg1[1,-1]               : (V) String               :
13       bytes at address 0x8057f60 : This is greg1

  0x8057e30 : greg2[1,-1]               : (V) String               :
13       bytes at address 0x8057fc0 : This is greg2

STANDARD Variables in SUBROUTINE main()

  0xbfffed54 : Var1[1,-1]                : (V) String                :
12       bytes at address 0x8057f00 : This is CAR1

STANDARD Variables in SUBROUTINE SUB1

  0xbfffe39c : I                         : (V) Integer               :
5

  0xbfffe3b8 : VM                        : (V) Uninitialised         :
(UNASSIGNED)

  0xbfffe3d4 : x1[1,-1]                  : (V) String                :
56       bytes at address 0x8059a60 :
2\3742\3749\374SUB1.b\3752\3741\3744\374SUB1.b\3751\3742\37412\374tes
t31.b\3751\3741\3747\374test31.b

  0xbfffe3f0 : rc[1,-1]                  : (V) String                :
0        bytes at address 0x40422b04 :

  0xbfffe40c : GGC2                      : (V) Integer               :
4

  0xbfffe428 : GGC3[1,-1]              : (V) String                :
56       bytes at address 0x8059e68 :
2\3742\3749\374SUB1.b\3752\3741\3744\374SUB1.b\3751\3742\37412\374tes
t31.b\3751\3741\3747\374test31.b
```

```
  0xbfffe444 : DSCB                    : (V) File descriptor      :
File './fb3'

  0xbfffe460 : rec                     : (V) Uninitialised        :
(UNASSIGNED)

0xbfffe47c : USERSTATS                 : (V) Uninitialised        :
(UNASSIGNED)
```

JBASETHREADCreate

Use the JBASETHREADCreate command to start a new thread.

COMMAND SYNTAX

JBASETHREADCreate(ProgramName, Arguments, User, Handle)

SYNTAX ELEMENTS

ProgramName Name of program to execute

Arguments Command line arguments

User Name of user in format "user{,account{,password}}" or "" to configuration as calling user id

JBASETHREADStatus

The JBASETHREADStatus command shows the status of all running threads.

COMMAND SYNTAX

JBASETHREADStatus(ThreadList)

SYNTAX ELEMENTS

ThreadList a list of all threads active in this process, with one attribute per thread.

The layout of the multi-values in each attribute is as follows:

< n,1 > port number
< n,2 > thread handle returned from JBASETHREADCreate

JQLCOMPILE

JQLCOMPILE compiles a jQL statement.

COMMAND SYNTAX

```
JQLCOMPILE (Statement, Command, Options, Messages)
```

SYNTAX ELEMENTS

Statement is the variable, which will receive the compiled statement, used by a majority of functions to execute and work on the result set etc.
Command is the actual jQL query that you want to compile (such as SELECT or something similar). Use RETRIEVE to obtain data records as the verb rather than an existing jQL verb. This will ensure that the right options are set internally. In addition, use any word that is not a jQL reserved word as the verb and it will work in the same way as RETRIEVE: implement a PLOT command that passes the entire command line into JQLCOMPILE and the results will be the same as if the first word were replaced with RETRIEVE.
Option: you must specify JQLOPT_USE_SELECT to supply a select list to the JQLEXECUTE function; the compile builds a different execution plan if using select lists.
Messages: If the statement fails to compile, this dynamic array is in the STOP format, therefore STOP messages can be programmed and printed. Provides a history of compilation for troubleshooting purposes; Returns -1 if there is a problem found in the statement and 0 for no problem

JQLEXECUTE

JQLEXECUTE starts executing a compiled jQL statement.

COMMAND SYNTAX

JQLEXECUTE (Statement, SelectVar)

SYNTAX ELEMENTS

Statement is the valid result of a call to a JQLCOMPILE(Statement, …)
SelectVar is a valid select list used to limit the statement to a predefined set of items. For example:

```
SELECT PROGRAMMERS WITH IQ_IN_PTS > 250

1 Item Selected

> LIST PROGRAMMERS NAME


PROGRAMMERS...    NAME

0123              COOPER, F B
```

This function returns -1 in the event of a problem, such as an incorrect statement variable. It will cause the statement to run against the database and produce a result set for use with JQLFETCH()

JQLFETCH

JQLFETCH fetches the next result in a compiled jQL statement.

COMMAND SYNTAX

JQLFETCH (Statement, ControlVar, DataVar)

SYNTAX ELEMENTS

Statement is the result of a valid call to JQLCOMPILE(), followed by a valid call to JQLEXECUTE().
ControlVar will receive the 'control break' elements of any query. FOR EXAMPLE, if there are BREAK values in the statement, described here are the totals:
The format of ControlVar is:

```
Attr 1            Level: 0 means detail line 1 – 25   for the control
breaks, the same as the A correlative NB.

Attr2             Item ID

Attr 3      Break control Value is 1 if a blank line should be output
first.

Attr 4            Pre-break value for 'B' option in header

Attr 5            Post-break value for 'B' option in header
```

DataVar will receive the actual screen data on a LIST statement for instance. The format is one attribute per column.
Applies Attribute 7 Conversions (or attribute 3 in Prime-style DICTS) to the data
If the property STMT_PROPERTY_FORMAT is set then each attribute is also formatted according to the width and justification of the attribute definition and any override caused by the use of FMT, of DISPLAY.LIKE on the command line –
NOTE that column headers may also affect the formatting for that column.
This function is called until there is no more output (multiple).

JQLGETPROPERTY

Gets the property of a compiled jQL statement

COMMAND SYNTAX

```
JQLGETPROPERTY (PropertyValue, Statement, Column, PropertyName)
```

SYNTAX ELEMENTS

PropertyValue Receives the requested property value from the system or “” if the property is not set
Statement The result of a valid JQLCOMPILE(Statement)
Column Specifies that you want the value of the property for a specific column (otherwise 0 for the whole statement).
PropertyName These are EQUATED values defined by INCLUDE’ing the file JQLINTERFACE.h.
This function returns -1 if there is a problem with the parameters or the programmer. These properties answer questions such as “Was LPTR mode asked for,” and “How many columns are there?”

Note: Properties are valid after the compile; this is the main reason for separating the compile and execute into two functions. After compiling, it is possible examine the properties and set properties before executing.

JQLPUTPROPERTY

JQLPUTPROPERTY sets a property in a compiled jQL statement.

COMMAND SYNTAX

```
JQLPUTPROPERTY (PropertyValue, Statement, Column, PropertyName)
```

SYNTAX ELEMENTS

PropertyValue is the value to which you want to set the specified property, such as one or "BLAH"
Statement is the result of a valid JQLCOMPILE() function.

NOTE: Some properties may require JQLEXECUTE()first.

Column Holds 0 for a general property of the statement, or a column number if it is something that can be set for a specific column.
PropertyName – These are EQUATED values defined by INCLUDE'ing the file JQLINTERFACE.h. There are lots of these and someone is going to have to document each one.

This function returns -1 if it locates a problem in the statement and zero for no problem.

NOTE: Properties are valid after the compile; this is the main reason for separating the compile and execute into two functions. After compiling, it is possible examine the properties and set properties before executing.

KEYIN

Use the KEYIN function to read a single character from the input buffer and return it.

COMMAND SYNTAX

KEYIN ( )

KEYIN uses raw keyboard input, therefore all special character handling (for example, backspace) is disabled. System special character handling (for example, processing of interrupts) is unchanged.

LATIN1

The LATIN1 function converts a UTF-8 byte sequence into the binary or latin1 equivalent.

COMMAND SYNTAX

LATIN1(expression)

SYNTAX ELEMENTS

The expression is to be a UTF-8 encoded byte sequence, which is the default format when executing in International Mode.

NOTES

Use this function for converting UTF-8 data into binary or the latin1 code page for external consumption. i.e. Tape devices.

LEFT

The LEFT function extracts a sub-string of a specified length from the beginning of a string.

COMMAND SYNTAX

LEFT(expression, length)

SYNTAX ELEMENTS

expression evaluates to the string from which the sub string is extracted.

length is the number of extracted characters if length is less than 1, LEFT() returns null.

NOTES

The LEFT() function is equivalent to sub-string extraction starting from the first character position, i.e. expression[1,length]

See also: RIGHT ()

EXAMPLE

```
S = "The world is my lobster"
CRT DQUOTE (LEFT(S,9))
CRT DQUOTE(LEFT(S,999))
CRT DQUOTE(LEFT(S,0))
```

This code displays:

"The world"

"The world is my lobster"

""

LEN

The LEN function returns the character length of the supplied expression.

COMMAND SYNTAX

LEN(expression)

INTERNATIONAL MODE

The LEN function when used in International Mode will return the number of characters in the specified expression rather than the number of bytes. If the expression consists of entirely of UTF-8 characters in the ASCII range 0 – 127 then the character length of the expression will equate to the byte length. However, when the expression contains characters outside the ASCII range 0 – 127 then byte length and character length will differ. If the byte is specifically required then use the BYTELEN function in place of the LEN function.

NOTE: Do not use programs manipulating byte counts in International Mode.

SYNTAX ELEMENTS

expression can evaluate to any type and the function will convert it to a string automatically.

EXAMPLES

```
Lengths = "
FOR I = 1 TO 50
   Lengths = LEN(Values)
NEXT I
```

LENS

Use the LENS function to return a dynamic array of the number of bytes in each element of the dynamic.array.

COMMAND SYNTAX

LENS (dynamic.array)
Each element of dynamic.array must be a string value. The characters in each element of dynamic.array are counted, with the counts returned.
The LENS function includes all blank spaces, including trailing blanks, in the calculation.
If dynamic.array evaluates to a null string, it returns zero (0). If any element of dynamic.array is null, returns zero (0) for that element.

INTERNATIONAL MODE

The LEN function when used in International Mode will return the number of characters in the specified expression rather than the number of bytes. If the expression consists of entirely of UTF-8 characters in the ASCII range 0 – 127 then the character length of the expression will equate to the byte length. However, when the expression contains characters outside the ASCII range 0 – 127 then byte length and character length will differ. If the byte is specifically required then use the BYTELEN function in place of the LEN function.

NOTE: Do not use programs to manipulate byte counts in International Mode.

LENDP

The LENDP function returns the display length of an expression

COMMAND SYNTAX

LENDP(expression)

SYNTAX ELEMENTS

The expression can evaluate to any type. The LENDP function will evaluate each character in the expression and return the calculated display length.

INTERNATIONAL MODE

The LENDP function when used in International Mode will return the display length for the characters in the specified expression rather than the number of bytes.

NOTE: Some characters, usually Japanese, Chinese, etc will return a display length of greater than one for some characters. Some characters, for instance control characters or null (char 0), will return a display length of 0.

LE - Less than or equal operator Ditto re GE and LES re INTERNATIONAL MODE

LES

Use the LES function to determine whether elements of one dynamic array are less than or equal to the elements of another dynamic array.

COMMAND SYNTAX

LES (array1, array2)

It compares each element of array1 with the corresponding element of array2. If the element from array1 is less than or equal to the element from array2, a 1 is returned in the corresponding element of a new dynamic array. If the element from array1 is greater than the element from array2, it returns a zero (0). If an element of one dynamic array has no corresponding element in the other dynamic array, it evaluates the undefined element as empty, and the comparison continues.

If either of a corresponding pair of elements is null, it returns null for that element. If you use the subroutine syntax, it returns the resulting dynamic array as return.array.

LN

The LN function returns the value of the natural logarithm of the supplied value.

COMMAND SYNTAX

LN(expression)

SYNTAX ELEMENTS

The **expression** should evaluate to a numeric value. The function will then return the natural logarithm of that value.

NOTES

The calculation of the natural logarithm is by using the mathematical constant e as a number base.

EXAMPLES

A = LN(22/7)

LOCALDATE

Return an internal date using the specified Timestamp and TimeZone combination.

COMMAND SYNTAX

LOCALDATE(Timestamp, TimeZone)

SYNTAX ELEMENTS

The LOCALDATE function uses the specified timestamp and adjusts the value by the specified time zone to return the date value in internal date format.

LOCALTIME

Return an internal time using the specified Timestamp and TimeZone combination.

COMMAND SYNTAX

LOCALTIME(Timestamp, TimeZone)

SYNTAX ELEMENTS

The LOCALTIME function uses the specified timestamp and adjusts the value by the specified time zone to return the time value in internal time format.

LOCATE

The LOCATE statement finds the position of an element within a specified dimension of a dynamic array.

COMMAND SYNTAX

LOCATE expression1 IN expression2{<expression3{,expression4}>}, {, expression5} {BY expression6} SETTING Var THEN|ELSE statement(s)

SYNTAX ELEMENTS

expression1 evaluates to the string that will be searched for in expression2.
expression2 evaluates to the dynamic array within which expression1 will be searched for.
expression3 and expression4, when specified, cause a value or subvalue search respectively.
expression5 indicates the field, value or subvalue from which the search will begin.
BY **expression6** causes the search to expect the elements to be arranged in a specific order, which can considerably improve the performance of some searches. The available string values for expression6 are:

AL	Values are in ascending alphanumeric order
AR	Values are in right justified, then ascending order
AN	Values are in ascending numeric order
DL	Values are in descending alphanumeric order
DR	Values are in right justified, then descending order
DN	Values are in descending numeric order

Var will be set to the position of the Field, Value or Sub-Value in which expression1 was found if indeed. If it was not found and expression6 was not specified then Var will be set to one position past the end of the searched dimension. If expression6 did specify the order of the elements then Var will be set to the position before which the element should be inserted to retain the specified order.
The statement must include one of or both of the THEN and ELSE clauses. If expression1 is found in an element of the dynamic array, it executes the statements defined by the THEN clause. If expression1 is not found in an element of the dynamic array, it executes the statements defined by the ELSE clause.

INTERNATIONAL MODE

When the LOCATE statement is used in International Mode, the statement will use the currently configured locale to determine the rules by which each string is considered less than or greater than the other will.

NOTES

See also: FIND, FINDSTR

EXAMPLES

```
Name = "Nelson"
LOCATE Name IN ForeNames BY "AL" SETTING Pos ELSE
    INS Name BEFORE ForeNames<Pos>
END
```

LOCK

The LOCK statement will attempt to set an execution lock thus preventing any other jBASE BASIC program that respects that lock to wait until this program has released it.

COMMAND SYNTAX

LOCK expression {THEN|ELSE statements}

SYNTAX ELEMENTS

The expression should evaluate to a numeric value between 0 and 255 (63 in R83 import mode).
The statement will execute the THEN clause (if defined) providing the lock could be taken. If another program holds the LOCK and an ELSE clause is provided then the statements defined by the ELSE clause are executed. If no ELSE clause was provided with the statement then it will block (hang) until the other program has released the lock.

NOTES

See also: UNLOCK.
If you used the environment variable JBASE BASICEMULATE set to r83, to compile the program the number of execution locks is limited to 64. If an execution lock greater than this number is specified, the actual lock taken is the specified number modulo 64.

EXAMPLES

```
LOCK 32 ELSE
   CRT "This program is already executing!"
STOP
END
```

LOOP

The LOOP construct allows the programmer to specify loops with multiple exit conditions.

COMMAND SYNTAX

LOOP statements1 WHILE|UNTIL expression DO statements2 REPEAT

SYNTAX ELEMENTS

statements1 and statements2 consist of any number of standard statements include the LOOP statement itself, thus allowing nested loops.

statements1 will always be executed at least once, after which the WHILE or UNTIL clause is evaluated.

expression is tested for Boolean TRUE/FALSE by either the WHILE clause or the UNTIL clause. When tested by the WHILE clause statements2 will only be executed if expression is Boolean TRUE. When tested by the UNTIL clause, statements2 will only be executed if the expression evaluates to Boolean FALSE.

REPEAT causes the loop to start again with the first statement following the LOOP statement.

NOTES

See also: BREAK, CONTINUE

EXAMPLES

```
LOOP WHILE B < Max DO
    Var<B> = B++ *6
REPEAT

LOOP
    CRT "+":
WHILE READNEXT KEY FROM List DO
    READ Record FROM FILE, KEY ELSE CONTINUE
    Record<1> *= 6
REPEAT
CRT
```

LOWER

The LOWER function lowers system delimiters in a string to the next lowest delimiter.

COMMAND SYNTAX

LOWER(expression)

SYNTAX ELEMENTS

The expression is a string containing one or more delimiters, lowered as follows:

ASCIICharacter	Lowered To
255	254
254	253
253	252
252	251
251	250
250	249
249	248

EXAMPLE

ValuemarkDelimitedVariable = LOWER(AttributeDelimitedVariable)

MAKETIMESTAMP

Generate a timestamp using combination of internal date, time and timezone.

COMMAND SYNTAX

```
MAKETIMESTAMP(InternalDate, InternalTime, TimeZone)
```

SYNTAX ELEMENTS

Use the MAKETIMESTAMP function to generate a timestamp using a specified time zone. The internal date and internal time values are combined together with the time zone specification to return a UTC timestamp as decimal seconds.

MAT

Use the MAT command to either assign every element in a specified array to a single value or to assign the entire contents of one array to another.

COMMAND SYNTAX

MAT Array = expression
MAT Array1 = MAT Array2

SYNTAX ELEMENTS

Array, **Array1** and **Array2** are all pre-dimensioned arrays declared with the DIM statement. Expression can evaluate to any data type.

NOTES

If any element of the array Array2 has not been assigned a value then a runtime error message will occur. This can be avoided by coding the statement MAT Array2 = " after the DIM statement.

EXAMPLES

```
001 DIM A(45), G(45)
002 MAT G = "Array value"
003 MAT A = MAT G
```

MATBUILD

Use the MATBUILD statement to create a dynamic array out of a dimensioned array.

COMMAND SYNTAX

MATBUILD variable FROM array{, expression1{, expression2}} {USING expression3}

SYNTAX ELEMENTS

variable is the jBASE BASIC variable into which the created dynamic array will be stored. Array is a previously dimensioned and assigned matrix from which the dynamic array will be created.
expression1 and expression2 should evaluate to numeric integers. expression1 specifies which element of the array the extraction will start with; expression2 specifies which element of the array the extraction will end with (inclusive).
By default, each array element is separated in the dynamic array by a field mark. By specifying **expression3**, the separator character can be changed. If expression3 evaluates to more than a single character, only the first character of the string is used.

NOTES

When specifying starts and end positions with multi-dimensional arrays, it is necessary to expand the matrix into its total number of variables to calculate the correct element number. See the information about dimensioned arrays earlier in this chapter for detailed instructions on calculating element numbers.

EXAMPLES

```
DIM A(40)
MATBUILD Dynamic FROM A,3,7 USING ":"
Builds a 5 element string separated by a : character.

MATBUILD Dynamic FROM A Builds a field mark separated dynamic array
from every element contained in the matrix A.
```

MATCHES

The MATCH or MATCHES function applies pattern matching to an expression.

INTERNATIONAL MODE

When using the MATCHES statement in International Mode, the statement will use the currently configured locale to determine the properties according to the Unicode Standard for each character in the expression. i.e., is the character alpha or numeric?

COMMAND SYNTAX

expression1 MATCHES **expression2**

SYNTAX ELEMENTS

expression1 may evaluate to any type. expression2 should evaluate to a valid pattern matching string as described below.
expression1 is then matched to the pattern supplied and a value of Boolean TRUE is returned if the pattern is matched. A value of Boolean FALSE is returned if the pattern is not matched.
expression2 can contain any number of patterns to match those separated by value marks. The value mark implies a logical OR of the specified patterns and the match will evaluate to Boolean TRUE if expression1 matches any of the specified patterns.

NOTES

The rule table shown below shows construction of pattern matching strings (n refers to any integer number).

Pattern	Explanation
nN	this construct matches a sequence of n digits
nA	this construct matches a sequence of n alpha characters
nC	this construct matches a sequence of n alpha characters or digits
nX	this construct matches a sequence of any characters
"string"	This construct matches the character sequence string exactly.

Applies the pattern to all characters in expression1 and it must match all characters in the expression to evaluate as Boolean TRUE.
Specify the integer value n as 0. This will cause the pattern to match any number of characters of the specified type.

EXAMPLES

IF Var MATCHES "0N" THEN CRT "A match!"
Matches if all characters in Var are numeric or Var is a null string.

IF Var MATCHES "0N".'2N"...

Matches if Var contains any number of numerics followed by the “.” character followed by 2 numeric characters. e.g. 345.65 or 9.99

Pattern = "4X':'6N';'2A"
Matched = Serno MATCHES Pattern
Matches if the variable Serno consists of a string of 4 arbitrary characters followed by the ":" character then 6 numerics then the ";" character and then 2 alphabetic characters. e.g. 1.2.:123456;AB or 17st:456789;FB

MATCHFIELD

COMMAND SYNTAX

MATCHFIELD (string, pattern, field)

DESCRIPTION

Use the MATCHFIELD function to check a string against a match pattern: See also: MATCH operator for information about pattern matching.

field is an expression that evaluates to the portion of the match string to be returned.

If string matches pattern, the MATCHFIELD function returns the portion of string that matches the specified field in pattern. If string does not match pattern, or if string or pattern evaluates to the null value, the MATCHFIELD function returns an empty string. If **field** evaluates to the null value, the MATCHFIELD function fails and the program terminates with a run-time error.

pattern must contain specifiers to cover all characters contained in string. For example, the following statement returns an empty string because not all parts of string are specified in the pattern:

MATCHFIELD ("XYZ123AB", "3X3N", 1)

To achieve a positive pattern match on the string above, use the following statement:

MATCHFIELD ("XYZ123AB", "3X3N0X", 1)

This statement returns a value of "XYZ".

EXAMPLES

In the following example, the string does not match the pattern:

In the following example, the entire string does not match the pattern:

Source Lines Program Output

```
Q=MATCHFIELD("AA123BBB9","2A0N3A0N",3)

PRINT "Q= ",Q

Q= BBB

ADDR='20 GREEN ST. NATICK, MA.,01234'

ZIP=MATCHFIELD(ADDR,"0N0X5N",3)

PRINT "ZIP= ",ZIP

ZIP= 01234

INV='PART12345 BLUE AU'
```

```
COL=MATCHFIELD(INV,"10X4A3X",2)

PRINT "COL= ",COL

COL= BLUE
```

Source Lines Program Output

```
XYZ=MATCHFIELD('ABCDE1234',"2N3A4N",1)

PRINT "XYZ= ",XYZ

XYZ=
```

Source Lines Program Output

```
ABC=MATCHFIELD('1234AB',"4N1A",2)

PRINT "ABC= ",ABC

ABC=
```

MATPARSE

Use the MATPARSE statement to assign the elements of a matrix from the elements of a dynamic array.

COMMAND SYNTAX

MATPARSE array{, expression1{, expression2}} FROM variable1 {USING expression3} SETTING variable2

SYNTAX ELEMENTS

array is a previously dimensioned matrix, which will be assigned to from each element of the dynamic array. variable1 is the jBASE BASIC variable from which the matrix array will be stored.
expression1 and expression2 should evaluate to numeric integers. expression1 specifies which element of the array the assignment will start with; expression2 specifies which element of the array the assignment will end with (inclusive).
By default, the dynamic array assumes the use of a field mark to separate each array element. By specifying expression3, the separator character can be changed. If expression3 evaluates to more than a single character, only the first character of the string is used.
As assignment will stop when the contents of the dynamic array have been exhausted, it can be useful to determine the number of matrix elements that were actually assigned to. If the **SETTING** clause is specified then variable2 will be set to the number of elements of the array that were assigned to.

NOTES

When specifying starts and end positions with multi-dimensional arrays, it is necessary to expand the matrix into its total number of variables to calculate the correct element number. See the information about dimensioned arrays earlier in this section for detailed instructions on calculating element numbers.

EXAMPLE

```
DIM A(40)

MATPARSE A,3,7 FROM Dynamic
```

Assign 5 elements of the array starting at element 3.

MATREAD

The MATREAD statement allows a record stored in a jBASE file to be read and mapped directly into a dimensioned array.

COMMAND SYNTAX

MATREAD array FROM {variable1,}expression {SETTING setvar} {ON ERROR statements} {LOCKED statements} {THEN|ELSE statements}

SYNTAX ELEMENTS

array should be a previously dimensioned array, which will be used to store the record to be read. If specified, **variable1** should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable1 is not specified then the default file is assumed. The expression should evaluate to a valid record key for the file.

If no record is found and can be read from the file then it is mapped into the array and executes the THEN statements (if any). If the record cannot be read from the file then array is unchanged and executes the ELSE statements (if any).

If the record could not be read because another process already had a lock on the record then one of two actions is taken. If the LOCKED clause was specified in the statement then the statements dependent on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the other process releases the lock. If a LOCKED clause is used and the read is successful, a lock will be set.

If the SETTING clause is specified, setvar will be set to the number of fields in the record on a successful read. If the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, it executes the statements following the ON ERROR clause for any of the above Incremental File Errors except error 128.

NOTES

The record is mapped into the array using a predefined algorithm. The record is expected to consist of a number of Field separated records, which are then assigned one at a time to each successive element of the matrix. See the notes on matrix organization earlier in this section for details of multi dimensional arrays.

If there were more fields in the record than elements in the array, then the final element of the array will be assigned all remaining fields. If there were fewer fields in the record than elements in the array then remaining array elements will be assigned a null value.

Note that if multi-values are read into an array element they will then be referenced individually as:

Array(n)<1,m>

not

Array(n)<m>

EXAMPLES

```
MATREAD Xref FROM CFile, "XREF" ELSE MAT Xref = "

MATREAD Ind FROM IFile, "INDEX" ELSE MAT Ind = 0

MATREAD record FROM filevar, id SETTING val ON ERROR

PRINT "Error number ":val:" occurred which prevented record from
being read."
    STOP

END THEN

    PRINT 'Record read successfully'

END ELSE

    PRINT 'Record not on file'

END
PRINT "Number of attributes in record = ": val
```

MATREADU

The MATREADU statement allows a record stored in a jBASE file to be read and mapped directly into a dimensioned array. The record will also be locked for update by the program.

COMMAND SYNTAX

MATREADU array FROM { variable1,}expression {SETTING setvar} {ON ERROR statements} {LOCKED statements} {THEN|ELSE statements}

SYNTAX ELEMENTS

array should be a previously dimensioned array, which will be used to store the record to be read. If specified, **variable1** should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If **variable1** is not specified then the default file is assumed. The expression should evaluate to a valid record key for the file.

If found, the record can be read from the file then it is mapped into array and executes the THEN statements (if any). If the record cannot be read from the file for some reason then array is unchanged and executes the ELSE statements (if any).

If the record could not be read because another process already had a lock on the record then one of two actions is taken. If the LOCKED clause was specified in the statement then the statements dependent on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the other process releases the lock.

If the SETTING clause is specified, setvar will be set to the number of fields in the record on a successful read. If the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

The record is mapped into the array using a predefined algorithm. The record is expected to consist of a number of Field separated records, which are then assigned one at a time to each successive element of the matrix. See the notes on matrix organization earlier in this section for details of the layout of multi dimensional arrays.

If there were more fields in the record than elements in the array, then the final element of the array will be assigned all remaining fields. If there were fewer fields in the record than elements in the array then remaining array elements will be assigned a null value.

NOTE: that if multi-values are read into an array element they will then be referenced individually as:

Array(n)<1,m>

not

Array(n)<m>

EXAMPLES

```
MATREADU Xref FROM CFile, "XREF" ELSE MAT Xref = "

MATREADU Ind FROM IFile, "INDEX" LOCKED

   GOSUBInformUserLock ;* Say it is locked

END THEN

   GOSUB InformUserOk ;* Say we got it

END ELSE

   MAT Ind = 0 ;* It was not there

END

MATREADU record FROM filevar, id SETTING val ON ERROR

    PRINT "Error number ":val:" occurred which prevented record from
being read."
    STOP

END LOCKED

    PRINT "Record is locked"

END THEN

    PRINT 'Record read successfully'

END ELSE

    PRINT 'Record not on file'

END
PRINT "Number of attributes in record = ": val
```

MATWRITE

The MATWRITE statement transfers the entire contents of a dimensioned array to a specified record on disc.

COMMAND SYNTAX

MATWRITE array ON { variable,}expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

array should be a previously dimensioned and initialized array. If specified, variable should be a previously opened file variable (i.e. the subject of an OPEN statement). If variable is not specified the default file variable is used. expression should evaluate to the name of the record in the file.
If the SETTING clause is specified and the write succeeds, setvar will be set to the number of attributes read into array.
If the SETTING clause is specified and the write fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

The compiler will check that the variable specified is a dimensioned array before its use in the statement.

EXAMPLES

```
DIM A(8)

MAT A = 99

....
MATWRITE A ON "NewArray" SETTING ErrorCode ON ERROR

    CRT "Error: ":ErrorCode:"  Record could not be written."

END
...
MATWRITE A ON RecFile, "OldArray"
```

MATWRITEU

The MATWRITEU statement transfers the entire contents of a dimensioned array to a specified record on file, in the same manner as the MATWRITE statement. An existing record lock will be preserved.

COMMAND SYNTAX

MATWRITEU array ON { variable,}expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

array should be a previously dimensioned and initialized array. If specified, variable should be a previously opened file variable (i.e. the subject of an OPEN statement). If variable is not specified the default file variable is used.

expression should evaluate to the name of the record in the file.

If the SETTING clause is specified and the write succeeds, setvar will be set to the number of attributes read into array.

If the SETTING clause is specified and the write fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

The compiler will check that the variable specified is indeed a dimensioned array before its use in the statement.

EXAMPLES

```
DIM A(8)

MAT A = 99

....
MATWRITEU A ON "NewArray"
```

MAXIMUM

The MAXIMUM function is used to return the element of a dynamic array with the highest numerical value.

COMMAND SYNTAX

MAXIMUM(DynArr)

SYNTAX ELEMENTS

DynArr should evaluate to a dynamic array.

NOTES

Null dynamic array elements are treat as zero.

Non-numeric dynamic array elements are ignored.

See also: MINIMUM.

EXAMPLE

If EResults is a variable containing the dynamic array:

```
1.45032:@AM:-3.60441:@VM:4.29445:@AM:2.00042:@SM:-3.90228
```

```
the code:
```

```
PRECISION 5
CRT = MAXIMUM(EResults)
```

```
displays 4.29445
```

MINIMUM

The MINIMUM function is used to return the element of a dynamic array with the lowest numerical value.

COMMAND SYNTAX

MINIMUM(DynArr)

SYNTAX ELEMENTS

DynArr should evaluate to a dynamic array.

NOTES

Null dynamic array elements are treat as zero.

Non-numeric dynamic array elements are ignored.

See also: MAXIMUM.

EXAMPLE

If EResults is a variable containing the dynamic array:

```
1.45032:@AM:-3.60851:@VM:4.29445:@AM:2.07042:@SVM:-3.90258
```

```
the code:
```

```
PRECISION 3
CRT = MINIMUM(EResults)
```

```
displays -3.903
```

MOD

The MOD function returns the arithmetic modulo of two numeric expressions.

COMMAND SYNTAX

MOD (expression1, expression2)

SYNTAX ELEMENTS

Both **expression1** and **expression2** should evaluate to numeric expressions or a runtime error will occur.

NOTES

The remainder of expression1 divided by expression2 calculates the modulo. If expression2 evaluates to 0, then the value of expression1 is returned.

EXAMPLES

```
FOR I = 1 TO 10000

    IF MOD (I, 1000) = 0 THEN CRT "+":

NEXT I
```

displays a "+" on the screen every 1000 iterations

MODS

Use the MODS function to create a dynamic array of the remainder after the integer division of corresponding elements of two dynamic arrays.

COMMAND SYNTAX

MODS (array1, array2)

The MODS function calculates each element according to the following formula:
XY.element = X ??(INT (X / Y) * Y)
X is an element of **array1** and Y is the corresponding element of array2. The resulting element is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If an element of array2 is 0, 0 is returned. If either of a corresponding pair of elements is null, null is returned for that element.

EXAMPLE

```
A=3:@VM:7
B=2:@SM:7:@VM:4
PRINT MODS (A,B)
```

The output of this program is: 1\0]3

MSLEEP

Allows the program to pause execution for a specified number of milliseconds

COMMAND SYNTAX

MSLEEP {milliseconds}

SYNTAX ELEMENTS

milliseconds must be an integer, which, specifies the number of milliseconds to sleep.

When there are no parameters assumes a default time of 1 millisecond.

NOTES

If the debugger is invoked while a program is sleeping and then execution continued, the user will be prompted:

Continue with SLEEP (Y/N) ?

If "N" is the response, the program will continue at the next statement after the MSLEEP

See also: SLEEP to sleep for a specified number of seconds or until a specified time.

EXAMPLES

```
Sleep for 1/10th of a second...

MSLEEP 100

*
* 40 winks...

MSLEEP 40000
```

MULS

See also: Floating point Operations

Use the MULS function to create a dynamic array of the element-by-element multiplication of two dynamic arrays.

COMMAND SYNTAX

MULS (array1, array2)

Each element of array1 is multiplied by the corresponding element of array2 with the result being returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If either of a corresponding pair of elements is null, null is returned for that element.

EXAMPLE

```
A=1:@VM:2:@VM:3:@SM:4
B=4:@VM:5:@VM:6:@VM:9
PRINT MULS (A,B)
```

The output of this program is: 4]10]18\0]0

NEGS

Use the NEGS function to return the negative values of all the elements in a dynamic array.

COMMAND SYNTAX

NEGS (dynamic.array)

If the value of an element is negative, the returned value is positive. If dynamic.array evaluates to null, null is returned. If any element is null, null is returned for that element.

NES

Use the NES function to determine whether elements of one dynamic array are equal to the elements of another dynamic array.

COMMAND SYNTAX

NES (array1, array2)

Each element of array1 is compared with the corresponding element of array2. If the two elements are equal, a 0 is returned in the corresponding element of a new dynamic array. If the two elements are not equal, a 1 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, a 1 is returned. If either of a corresponding pair of elements is null, null is returned for that element.

NOBUF

Use the NOBUF statement to turn off buffering for a file previously opened for sequential processing.

COMMAND SYNTAX

NOBUF file.variable {THEN statements [ELSE statements] | ELSE statements}

DESCRIPTION

jBASE can buffer for sequential input and output operations. The NOBUF statement turns off this behavior and causes all writes to the file to be performed immediately. The NOBUF statement should be used in conjunction with a successful OPENSEQ statement and before any input or output is performed on the record.

If the NOBUF operation is successful, it executes the THEN statements otherwise, executes the ELSE statements. If file.variable is not a valid file descriptor then NOBUF statement fails and the program enters the debugger.

EXAMPLE

In the following example, if RECORD in DIRFILE can be opened, output buffering is turned off:

```
OPENSEQ 'DIRFILE', 'RECORD' TO DATA THEN NOBUF DATA

ELSE ABORT
```

NOT

The NOT function is used to invert the Boolean value of an expression. It is useful for explicitly testing for a false condition.

COMMAND SYNTAX

```
NOT (expression)
```

SYNTAX ELEMENTS

expression may evaluate to any Boolean result.

NOTES

The NOT function will return Boolean TRUE if the expression returned a Boolean FALSE. It will return Boolean FALSE of the expression returned a Boolean TRUE.
The NOT function is useful for explicitly testing for the false condition of some test and can clarify the logic of such a test.

EXAMPLES

```
EQU Sunday TO NOT (MOD (DATE(), 7))

IF Sunday THEN

    CRT "It is Sunday!"

END
```

In this example, the expression MOD (DATE(),7) will return 0 (FALSE) if the day is Sunday and 1 to 6 (TRUE) for the other days. To explicitly test for the day Sunday we need to invert the result of the expression. BY using the NOT function we return a 1 (TRUE) if the day is Sunday and 0 (FALSE) for all other values of the expression.

NOTS

Use the NOTS function to return a dynamic array of the logical complements of each element of dynamic.array.

COMMAND SYNTAX

NOTS (dynamic.array)

If the value of the element is true, the NOTS function returns a value of false (0) in the corresponding element of the returned array. If the value of the element is false, the NOTS function returns a value of true (1) in the corresponding element of the returned array.

A numeric expression that evaluates to 0 has a logical value of false. A numeric expression that evaluates to anything else, other than the null value, is a logical true.

An empty string is logically false. All other string expressions, including strings, which consist of an empty string, spaces, or the number 0 and spaces, are logically true.

If any element in dynamic.array is null, it returns null for that element.

EXAMPLE

```
X=5; Y=5

PRINT NOTS X-Y:@VM:X+Y)
```

The output of this program is:

1]0

NULL

The NULL statement performs no function but can be useful in clarifying syntax and where the language requires a statement but the programmer does not wish to perform any actions.

COMMAND SYNTAX

NULL

SYNTAX ELEMENTS

None

EXAMPLES

```
LOCATE A IN B SETTING C ELSE NULL
```

NUM

Use the NUM function to test arguments for numeric values.

COMMAND SYNTAX

NUM (expression)

SYNTAX ELEMENTS

expression may evaluate to any data type.

NOTES

If found that every character in expression is numeric then NUM returns a value of Boolean TRUE If any character in expression is found not to be numeric then a value of Boolean FALSE is returned. Note that to execute user code migrated from older systems correctly, the NUM function will accept both a null string and the single characters ".", "+", and "-" as being numeric.
NOTE: if running jBASE BASIC in ros emulation the "." , "+" and "-" characters would not be considered numeric.

EXAMPLE

```
LOOP
    INPUT Answer,1

    IF NUM (Answer) THEN BREAK ;* Exit loop if numeric

REPEAT
```

NUMS

Use the NUMS function to determine whether the elements of a dynamic array are numeric or nonnumeric strings.

COMMAND SYNTAX

NUMS (dynamic.array)

If an element is numeric, a numeric string, or an empty string, it evaluates to true, and returns a value of 1 to the corresponding element in a new dynamic array. If the element is a nonnumeric string, it evaluates to false, and returns a value of 0.

The NUMS of a numeric element with a decimal point (.) evaluates to true; the NUMS of a numeric element with a comma (,) or dollar sign ($) evaluates to false.

If dynamic.array evaluates to null, it returns null. If an element of dynamic.array is null, it returns null for that element.

INTERNATIONAL MODE

When using the NUMS function in International Mode, the statement will use the Unicode Standard to determine whether an expression is numeric.

OBJEXCALLBACK

jBASE OBjEX provides the facility to call a subroutine from a front-end program written in a tool that supports OLE, such as Delphi or Visual Basic. The OBJEXCALLBACK statement allows communication between the subroutine and the calling OBjEX program.

COMMAND SYNTAX

OBJEXCALLBACK expression1, expression2 THEN|ELSE statements

SYNTAX ELEMENTS

expression1 and expression2 can contain any data. They are returned to the OBjEX program where they are defined as variants.
If the subroutine containing the OBJEXCALLBACK statement is not called from an OBjEX program (using the Call Method) then the ELSE clause will be taken.

NOTES

The OBJEXCALLBACK statement is designed to allow jBASE BASIC subroutines to temporarily return to the calling environment to handle exception conditions or prompt for additional information. After servicing this event, the code should return control to the jBASE BASIC program to ensure that the proper clean up operations are eventually made. The two parameters can be used to pass data between the jBASE BASIC and OBjEX environments in both directions. They are defined as Variants in the OBjEX environment and as normal variables in the jBASE BASIC environment.
See the OBjEX documentation for more information.

EXAMPLE

```
param1 = "SomeActionCode"

param2 = ProblemItem

OBJEXCALLBACK param1, param2 THEN

* this routine was called from ObjEX

END ELSE

* this routine was not called from ObjEX

END
```

OCONV

Use the OCONV statement to convert internal representations of data to their external form.

COMMAND SYNTAX

OCONV (expression1, expression2)

SYNTAX ELEMENTS

expression1 may evaluate to any data type but must be relevant to the conversion code.
expression2 should evaluate to a conversion code from the list below. Alternatively, expression2 may evaluate to a user exit known to the jBASE BASIC language or supplied by the user.

INTERNATIONAL MODE

Description of date, time, number and currency conversions when used in ICONV and International Mode

NOTES

OCONV will return the result of the conversion of expression1 by expression2. Shown below are valid conversion codes:

Conversion	Action
D{n{c}}	Converts an internal date to an external date format. The numeric argument n specifies the field width allowed for the year and can be 0 to 4 (default 4). The character c causes the date to be return in the form ddcmmcyyyy. If it is not specified the month name is return in abbreviated form.
DI	Allow the conversion of an external date to the internal format even though an output conversion is expected.
DD	Returns the day in the current month.
DM	Returns the number of the month in the year.
DMA	Returns the name of the current month.
DJ	Returns the number of the day in the year (0-366).
DQ	Returns the quarter of the year as a number 1 to 4
DW	Returns the day of the week as a number 1 to 7 (Monday is 1).
DWA	Returns the name of the day of the week.
DY{n}	Returns the year in a field of n characters.
F	Given a prospective filename for a command such as CREATE-FILE this conversion will return a filename that is acceptable to the version of UNIX jBASE is running on.
MCA	Removes all but alphabetic characters from the input string.
MC/A	Removes all but the NON-alphabetic characters in the input string.
MCN	Removes all but numeric characters in the input string
MC/N	Removes all but NON numeric characters in the input string

Conversion	Action
MCB	Returns just the alphabetic and numeric characters from the input string
MC/B	Remove the alphabetic and numeric characters from their input string.
MCC;s1;s2	Replaces all occurrences of string s1 with string s2
MCL	Converts all upper case characters in the string to lower case characters
MCU	Converts all lower case characters in the string to upper case characters.
MCT	Capitalizes each word in the input string; e.g. JIM converts to Jim
MCP{c}	Converts all non-printable characters to a period character "." in the input string. When supplied use the character "c" in place of the period.
MCPN{n}	In the same manner as the MCP conversion, it replaces all non-printable characters. The ASCII hexadecimal value follows the replacing character.
MCNP{n}	Performs the opposite conversion to MCPN. The ASCII hexadecimal value following the tilde character converts to its original binary character value.
MCDX	Converts the decimal value in the input string to its hexadecimal equivalent.
MCXD	Converts the hexadecimal value in the input string to its decimal equivalent.
Gncx	Extracts x groups separated by character c skipping n groups, from the input string.
MT{HS}	Performs time conversions.
MD	Converts the supplied integer value to a decimal value.
MP	Converts a packed decimal number to an integer value.
MX	Converts ASCII input to hexadecimal characters.
T	Performs file translations given a cross-reference table in a record in a file.

OCONVS

Use the OCONVS function to convert the elements of dynamic.array to a specified format for external output.

COMMAND SYNTAX

OCONVS (dynamic.array, conversion)
Converts the elements to the external output format specified by conversion and returned in a dynamic array conversion must evaluate to one or more conversion codes separated by value marks (ASCII 253).
If multiple codes are used, they are applied from left to right as follows: the left-most conversion code is applied to the element, the next conversion code to the right is then applied to the result of the first conversion, and so on.
If dynamic.array evaluates to null, it returns null. If any element of dynamic.array is null, it returns null for that element. If conversion evaluates to null, the OCONVS function fails and the program terminates with a run-time error message.
The STATUS function reflects the result of the conversion:

0 The conversion is successful.
1 Passes an invalid element to the OCONVS function; the original element is returned. If the invalid element is null, it returns null for that element.
2 The conversion code is invalid.

For information about converting elements in a dynamic array to an internal format
See also: ICONVS function.

INTERNATIONAL MODE

Description of date, time, number and currency conversions when used in ICONV and International Mode

ONGOTO

The ON...GOSUB and ON...GOTO statements are used to transfer program execution to a label based upon a calculation.

COMMAND SYNTAX

ON expression GOTO label{, label...}
ON expression GOSUB label{, label...}

SYNTAX ELEMENTS

expression should evaluate to an integer numeric value. Labels should be defined somewhere in the current source file.
ON GOTO will transfer execution to the labeled source code line in the program.
ON GOSUB will transfer execution to the labeled subroutine within the source code.

NOTES

Use the value of expression as an index to the list of labels supplied. If the expression evaluates to 1 then the first label will be jumped to, 2 then the second label will be used and so on.
If the program was compiled when the emulation included the setting generic_pick = true, then no validations are performed on the index to see if it is valid. Therefore, if the index is out of range this instruction will take no action and report no error.
If the program was compiled for other emulations then the index will be range checked. If found that the index is less than 1, it is assumed to be 1 and a warning message is issued If the index is found to be too big, then the last label in the list will be used to transfer execution and a warning message issued.

EXAMPLE

```
INPUT Ans,1_

ON SEQ (Ans)-SEQ(A)+1 GOSUB RoutineA, RoutineB...
```

OPEN

Use the OPEN statement to open a file or device to a descriptor variable within jBASE BASIC.

COMMAND SYNTAX

OPEN {expression1,}expression2 TO {variable} {SETTING setvar} THEN|ELSE statements

SYNTAX ELEMENTS

The combination of expression1 and expression2 should evaluate to a valid file name of a file type that already installed on the jBASE system. If the file has a dictionary section to be opened by the statement then specify by the literal string "DICT" being specified in expression1. If specified, the variable will be used to hold the descriptor for the file. It should then be to access the file using READ and WRITE. If no file descriptor variable is supplied, then the file will be opened to the default file descriptor.
Specific data sections of a multi level file may specified by separating the section name from the file name by a "," char in expression2.
If the OPEN statement fails it will execute any statements associated with an ELSE clause. If the OPEN is successful, it will execute any statements associated with a THEN clause. Note that the syntax requires either one or both of the THEN and ELSE clauses.
If specifying the SETTING clause and the open fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES: The OPEN statement uses the environment variable JEDIFILEPATH to search for the named file. If there is no defined named file, it will search the current working directory followed by the home directory of the current process.
The file that is the subject of the OPEN statement can be of any type known to the jBASE system. Its type will be determined and correctly opened transparently to the application, which need not be aware of the file type.
A jBASE BASIC program can open an unlimited amount of files.

EXAMPLES

```
OPEN "DICT", "CUSTOMERS" TO F.Dict.Customers ELSE

     ABORT 201, "DICT CUSTOMERS"

END
```

opens the dictionary section of file CUSTOMERS to its own file descriptor F.Dict.Customers.

```
OPEN "CUSTOMERS" ELSE ABORT 201, "CUSTOMERS"
```

opens the CUSTOMERS file to the default file variable.

OPENDEV

Opens a device (or file) for sequential writing and/or reading

COMMAND SYNTAX

OPENDEV Device TO FileVar { LOCKED statements } THEN | ELSE statements

SYNTAX ELEMENTS

Device specifies the target device or file

FileVar contains the file descriptor of the file when the open was successful

Statements conditional jBASE BASIC statements

NOTES

If the device does not exist or cannot be opened it executes the ELSE clause. Once open it takes a lock on the device. If the lock cannot be taken then the LOCKED clause is executed if it exists otherwise the ELSE clause is executed. The specified device can be a regular file, pipe or special device file. Regular file types only take locks. Once open the file pointer is set to the first line of sequential data.

EXAMPLE

```
OPENDEV "\\.\TAPE0" TO tape.drive ELSE STOP
```

Opens the Windows default tape drive and prepares it for sequential processing.

For more information on sequential processing, see READSEQ, WRITESEQ the sequential processing example.

OPENINDEX

The OPENINDEX statement is used to open a particular index definition for a particular file. This index file variable can later be used with the SELECT statement.

COMMAND SYNTAX

OPENINDEX filename,indexname TO indexvar {SETTING setvar} THEN|ELSE statements

SYNTAX ELEMENTS

filename should correspond to a valid file which has at least one index.
indexname should correspond to an index created for the filename.
indexvar is the variable that holds the descriptor for the index.
If the **OPEN** statement fails it will execute any statements associated with an ELSE clause. If the OPEN is successful it will execute any statements associated with a THEN clause. Note that the syntax requires either one or both of the THEN and ELSE clauses.
If the **SETTING** clause is specified and the open fails, setvar will be set to one of the following values:

EXAMPLES

```
OPENINDEX "CUSTOMER","IXLASTNAME" TO custlastname.ix SETTING errval ELSE
  CRT "OPENINDEX failed for file CUSTOMER, index IXLASTNAME"
  ABORT
END
```

OPENPATH

Use the OPENPATH statement to open a file (given an absolute or relative path) to a descriptor variable within jBASE BASIC.

See also: the OPEN statement.

COMMAND SYNTAX

OPENPATH expression1 TO {variable} {SETTING setvar} THEN|ELSE statements

SYNTAX ELEMENTS

Expression1 should be an absolute or relative path to the file including the name of the file to be opened. If specified, variable will be used to hold the descriptor for the file. It should then be to access the file using READ and WRITE. If no file descriptor variable is supplied, then the file will be opened to the default file descriptor.

If the **OPENPATH** statement fails it will execute any statements associated with an ELSE clause. If successful, the OPENPATH will execute any statements associated with a THEN clause. Note that the syntax requires either one or both of the THEN and ELSE clauses.

If the **SETTING** clause is specified and the open fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

The path specified may be either a relative or an absolute path and must include the name of the jBASE file being opened.

The file that is the subject of the OPENPATH statement can be of any type known to the jBASE system. Its type will be determined and correctly opened transparently to the application, which need not be aware of the file type.

A jBASE BASIC program can open an unlimited amount of files.

EXAMPLES

```
OPENPATH "C:\Home\CUSTOMERS" TO F.Customers ELSE

     ABORT 201, "CUSTOMERS"

END
```

opens the file CUSTOMERS (located in C:\Home) to its own file descriptor F.Customers

```
OPEN "F:\Users\data\CUSTOMERS" ELSE ABORT 201, "CUSTOMERS"
```

opens the CUSTOMERS file (located in F:\Users\data) to the default file variable.

OPENSEQ

Opens a file for sequential writing and/or reading

COMMAND SYNTAX

OPENSEQ Path{,File} {READONLY} TO FileVar { LOCKED statements } THEN | ELSE statements

SYNTAX ELEMENTS

Path specifies the relative or absolute path of the target directory or file

File specifies additional path information of the target file

FileVar contains the file descriptor of the file when the open was successful

Statements conditional jBASE BASIC statements

NOTES

If the file does not exist or cannot be opened it then executes the ELSE clause. However, if JBASICEMULATE is set for Sequoia (use value "seq") emulation then OPENSEQ will create the file if it does not exist. This behavior can also be achieved by specifying "openseq_creates = true" in Config_EMULATE for the emulation being used. Once open a lock is taken on the file. If the lock cannot be taken then the LOCKED clause is executed if it exists otherwise the ELSE clause is executed. If specified the READONLY process takes a read lock on the file, otherwise it takes a write lock. The specified file can be a regular, pipe or special device file. Locks are only taken on regular file types. Once open the file pointer is set to the first line of sequential data.

SEQUENTIAL FILE PROCESSING EXAMPLES

EXAMPLE 1

```
This program uses sequential processing to create (write to)an ASCII
text file


* from a jBASE hashed file. It illustrates the use of the commands:

*      OPENSEQ, WRITESEQ, WEOFSEQ, CLOSESEQ

*
* First, let's set the destination directory and file path

  Path = "d:\temp\textfile"

*
* Open the destination file path. If it does not exist it will be
created.
```

```
* Note that "openseq_creates=true" must be set for the emulation in
config_EMULATE
  OPENSEQ Path TO MyPath THEN
      CRT "The file already exists and we don't want to overwrite
it."
  END ELSE
      CRT "File is being created..."
  END
*
* Open the jBASE file
  OPEN "FileName" TO jBaseFile ELSE STOP
  SELECT
 jBaseFile          ;* Process all records
*
* Now, let's loop thru each item and build the ASCII text file.
  LOOP WHILE READNEXT
 ID DO
      READ
 MyRec FROM jBaseFile, ID THEN
          Line = ""
*
* Process MyRec and build the Line variable with the information to
be
* written to the ASCII text file. jBASE automatically takes care of
the
* end-of-line delimiters in this case a cr/lf is appended to the end
* of each line However, this can be changed with the IOCTL() function
.*
          WRITESEQ Line TO MyPath ELSE
              CRT "What happened to the file?"
```

```
            STOP
        END
    END
  REPEAT
*
* Wrapup

  WEOFSEQ MyPath

  CLOSESEQ MyPath
```

EXAMPLE 2

```
This program uses sequential processing to read from an ASCII text
file

* and write to a jBASE hashed file. It illustrates the use of the
commands:

*      OPENSEQ, READSEQ, CLOSESEQ

*
* First, let's define the path where the sequential file resides.

  Path = "d:\temp\textfile"

*
* Open the file. If it does not exist an error will be produced.

  OPENSEQ Path TO MyPath ELSE

      CRT "Can't find the specified directory or file."

      ABORT
  END
*
* Open the jBASE hashed file

  OPEN "FileName" TO jBaseFile ELSE STOP


*
* Now, let's read and process each line of the ASCII (sequential)
file.

  LOOP
      READSEQ Line FROM MyPath THEN

Initialize the record that will be written to the jBASE hashed file.
```

```
          MyRec = ""
*
* Process the Line variable. This involves extracting the information
 which
define the key and data of the record to be written to the base
 hashed
* file. This will be left up to the application developer since a


 "line"
```

could either be fixed length or delimited by some character such as a tab or a comma. We will assume that Key & MyRec are assembled here.

```
*
* All that's left to do is to write to the jBASE-hashed file
.          WRITE MyRec on jBaseFile, Key
      END
  REPEAT
*
* Wrapup
  CLOSESEQ MyPath
```

OPENSER

Use the OPENSER statement to handle the Serial IO. However, the OPENSER statement has also been provided.

Serial IO to the COM ports on NT and to device files, achieves this on UNIX by using the sequential file statements. In addition, you can perform certain control operations using the IOCTL function.

COMMAND SYNTAX

OPENSER Path,DevInfo| PIPE TO FileVar THEN | ELSE Statements

SYNTAX ELEMENTS

Path is the pathname of the required device.

DevInfo consists of the following:

Baud		baud rate required
Flow	y	X-ON X-OFF flow control (default)
	n	no flow control
	i	input flow control
	o	output flow control
Parity	e	7 bit even parity
	o	7 bit odd parity
	n	8 bit no parity, (Default)
	s	8 bit no parity, strip top bit

PIPE specifies the file is to be opened to a PIPE for reading.

NOTES

The PIPE functionality allows a process to open a PIPE, once opened then the process can execute a command via the WRITESEQ/SEND statement and then received the result back via the GET/READSEQ statements.

EXAMPLE

```
FileName = "/dev/tty01s"

OPENSER FileName TO File ELSE STOP 201,FileName

WRITESEQ "ls -ail" ON File,"" ;* ONLY for PIPEs

LOOP
    Terminator = CHAR (10)

    WaitTime = 4

    GET Input SETTING Count FROM File UNTIL Terminator RETURNING

 TermChar
```

```
 WAITING WaitTime THEN

        CRT "Get Ok, Input ":Input:" Count ":Count:"TermChar

 ":TermChar

    END ELSE

        CRT "Get Timed out Input ":Input:" Count ":Count:" TermChar

 ":TermChar
    END
WHILE Input NE "" DO

REPEAT
```

ORS

Use the ORS function to create a dynamic array of the logical OR of corresponding elements of two dynamic arrays.

COMMAND SYNTAX

ORS (array1, array2)

Each element of the new dynamic array is the logical OR of the corresponding elements of array1 and array2. If an element of one dynamic array has no corresponding element in the other dynamic array, it assumes a false for the missing element.

If both corresponding elements of array1 and array2 are null, it returns null for those elements. If one element is the null value and the other is 0 or an empty string, it returns null. If one element is the null value and the other is any value other than 0 or an empty string, it returns true.

EXAMPLE

```
A="A":@SM:0:@VM:4:@SM:1
B=0:@SM:1-1:@VM:2
PRINT ORS (A,B)
```

The output of this program is: 1\0]1\1

OSBREAD

The OSBREAD command reads data from a file starting at a specified byte location for a certain length of bytes, and assigns the data to a variable.

COMMAND SYNTAX

OSBREAD var FROM file.var [AT byte.expr] LENGTH length.expr [ON ERROR statements]
OSBREAD performs an operating system block read on a UNIX or Windows file.
REMINDER:
Before you use OSBREAD, you must open the file by using the OSOPEN or OPENSEQ command.
NOTE: jBASE uses the ASCII 0 character [CHAR (0)] as a string-end delimiter. Therefore, ASCII 0 cannot be used in any string variable within jBASE. OSBREAD converts CHAR(0) to CHAR(128) when reading a block of data.

SYNTAX ELEMENTS

var specifies a variable to which to assign the data read.
FROM file.var specifies a file from which to read the data.
AT byte.expr specifies a location in the file from which to begin reading data. If byte.expr is 0, the read begins at the beginning of the file.
LENGTH length.expr specifies a length of data to read from the file, starting at byte.expr. length.expr cannot be longer than the maximum string length determined by your system configuration.
ON ERROR statements specifies statements to execute if a fatal error occurs (if the file is not open, or if the file is a read-only file). If you do not specify the ON ERROR clause, the program terminates under such fatal error conditions.

STATUS Function Return Values

After you execute OSBREAD, the STATUS function returns either 0 or a failure code.

EXAMPLES

In the following example, the program statement reads 10,000 bytes of the file MYPIPE starting from the beginning of the file. The program assigns the data it reads to the variable TEST.

```
OSBREAD Data FROM MYPIPE AT 0 LENGTH 10000
```

OSBWRITE

The OSBWRITE command writes an expression to a sequential file starting at a specified byte location.

COMMAND SYNTAX

OSBWRITE expr {ON | TO} file.var [AT byte.expr] [NODELAY] [ON ERROR statements]
OSBWRITE immediately writes a file segment out to the UNIX, Windows NT, or Windows 2000 file. You do not have to specify a length expression because the number of bytes in expr is written to the file.
REMINDER: Before you use OSBWRITE, you must open the file by using the OSOPEN or OPENSEQ command.
NOTE: jBASE uses the ASCII 0 character [CHAR (0)] as a string-end delimiter. Therefore, ASCII 0 cannot be used in any string variable within jBASE. If jBASE reads a string that contains CHAR(0) characters by using OSBREAD, those characters are converted to CHAR(128).
OSBWRITE converts CHAR (128) back to CHAR(0) when writing a block of characters.

SYNTAX ELEMENTS

expr specifies the expression to write to the file.
ON | TO file.var specifies the file on which to write the expression
AT byte.expr If byte.expr is 0, the write begins at the beginning of the file.
NODELAY forces an immediate write.
ON ERROR statements specifies statements to execute if the OSBWRITE statement fails with a fatal error because the file is not open, an I/O error occurs, or jBASE cannot find the file. If you do not specify the ON ERROR clause and a fatal error occurs, the program terminates.

STATUS Function Return Values

After you execute OSBWRITE, the STATUS function returns either 0 or a failure code.

0	The write was successful.
1	The write failed.

EXAMPLE

In the following example, the program statement writes the data in MYPIPE to the opened file starting from the beginning of the file:

```
OSBWRITE Data ON MYPIPE AT 0
```

OSCLOSE

The OSCLOSE command closes a sequential file that you opened with the OSOPEN or OPENSEQ command.

COMMAND SYNTAX

OSCLOSE file.var [ON ERROR statements]

SYNTAX ELEMENTS

file.var Specifies the file to close.

ON ERROR statements Specifies statements to execute if the OSCLOSE statement fails with a fatal error because the file is not open, an I/O error occurs, or jBASE cannot find the file.

If you do not specify the ON ERROR clause and a fatal error occurs, the program will enter the debugger.

STATUS Function Return Values

After you execute OSCLOSE, the STATUS function returns either 0 or a failure code.

0	it closes the file successfully.
1	Close failed.

EXAMPLE

In the following example, the program statement closes the file opened to MYPIPE file variable.

```
OSCLOSE MYPIPE
```

OSDELETE

The OSDELETE command deletes a NT or UNIX file.

COMMAND SYNTAX

OSDELETE filename [ON ERROR statements]

SYNTAX ELEMENTS

filename Specifies the file to delete. filename must include the file path. If you do not specify a path, jBASE searches the current directory.

ON ERROR statements Specifies statements to execute if the OSDELETE statement fails with a fatal error because the file is not open, an I/O error occurs, or jBASE cannot find the file.

If you do not specify the ON ERROR clause and a fatal error occurs, the program terminates.

STATUS Function Return Values

After you execute OSDELETE, the STATUS function returns either 0 or a failure code.

0 It deletes the file

1 Delete failed.

EXAMPLES

In the following example, the program statement deletes the file ‘MYPIPE’ in the current directory:

```
OSDELETE "MYPIPE"
```

OSOPEN

The OSOPEN command opens a sequential file that does not use CHAR (10) as the line delimiter.

COMMAND SYNTAX

OSOPEN filename TO file.var
[ON ERROR statements] {THEN | ELSE} statements [END]
Read/write access mode is the default. Specify this access mode by omitting READONLY and WRITEONLY.
TIP: After opening a sequential file with OSOPEN, use OSBREAD to read a block of data from the file, or OSBWRITE to write a block of data to the file. You also can use READSEQ to read a record from the file, or WRITESEQ or WRITESEQF to write a record to the file, if the file is not a named pipe. (READSEQ, WRITESEQ, WRITESEQF are line-oriented commands that use CHAR (10) as the line delimiter.)

SYNTAX ELEMENTS

filename Specifies the file to open. filename must include the entire path name unless the file resides in the current directory.
TO file.var Specifies a variable to contain a pointer to the file.
ON ERROR statements specifies statements to execute if the OSOPEN statement fails with a fatal error because the file is not open, an I/O error occurs, or JBASE cannot find the file. If you do not specify the ON ERROR clause and a fatal error occurs, the program enters the debugger.
THEN statements Executes if the read is successful.
ELSE statements Executes if the read is not successful or the record (or ID) does not exist

EXAMPLE

In the following example, the program statement opens the file ‘MYSLIPPERS’ as SLIPPERS.

```
OSOPEN 'MYSLIPPERS' TO SLIPPERS ELSE STOP
```

OSREAD

Reads an OS file.

COMMAND SYNTAX

OSREAD Variable FROM expression {ON ERROR Statements} {THEN | ELSE} Statements {END}

SYNTAX ELEMENTS

Variable - Specifies the variable to contain the data from the read.
Expression - Specifies the full file path. If the file resides in the JEDIFILEPATH then just the file name is required.
ON ERROR Statements - Conditional jBASE BASIC statements to execute if the OSREAD statement fails with a fatal error because the file is not open, an I/O error occurs, or jBASE cannot find the file. If you do not specify the ON ERROR clause and a fatal error occurs, the program terminates.
THEN | ELSE If the OSREAD statement fails it will execute any statements associated with an ELSE clause. If the OSREAD is successful, it will execute any statements associated with a THEN clause. Note that the syntax requires either one or both of the THEN and ELSE clauses.

WARNING

Do not use OSREAD on large files. The jBASE BASIC OSREAD command reads an entire sequential file and assigns the contents of the file to a variable. If the file is too large for the program memory, the program aborts and generates a runtime error message. On large files, use OSBREAD or READSEQ.
jBASE uses the ASCII 0 character (CHAR (0)) as a string-end delimiter. ASCII 0 is not useable within string variable in jBASE BASIC. This command converts CHAR(0) to CHAR(128) when reading a block of data.

```
OSREAD MyFile FROM "C:\MyDirectory\MyFile" ELSE PRINT "FILE NOT FOUND"
```

OSWRITE

The OSWRITE command writes the contents of an expression to a sequential file.

COMMAND SYNTAX

OSWRITE expr {ON | TO} filename [ON ERROR statements]

NOTE:

JBASE uses the ASCII 0 character [CHAR(0)] as a string-end delimiter. For this reason, you cannot use ASCII 0 in any string variable in jBASE. If jBASE reads a string with a CHAR(0) character, and then the character is converted to CHAR(128), OSWRITE converts CHAR(128) to CHAR(0) when writing a block of characters.

SYNTAX ELEMENTS

expr Specifies the expression to write to filename.
ON | TO filename specifies the name of a sequential file to which to write.
ON ERROR statements Specifies statements to execute if the OSWRITE statement fails with a fatal error because the file is not open, an I/O error occurs, or jBASE cannot find the file. If you do not specify the ON ERROR clause and a fatal error occurs, the program enters the debugger.

EXAMPLE

In the following example, the program segment writes the contents of FOOTWEAR to the file called "PINK" in the directory '/usr/local/myslippers'
OSWRITE FOOTWEAR ON "/usr/local/myslippers"

OUT

The OUT statement is used to send raw characters to the current output device (normally the terminal).

COMMAND SYNTAX

OUT expression

SYNTAX ELEMENTS

expression should evaluate to a numeric integer in the range 0 to 255, being the entire range of ASCII characters.

NOTES

The numeric expression is first converted to the raw ASCII character specified and then sent directly to the output advice.

EXAMPLES

```
EQUATE BELL TO OUT 7

BELL ;* Sound terminal bell

FOR I = 32 TO 127; OUT I; NEXT I ;* Printable chars

BELL
```

PAGE

Prints any FOOTING statement, throws a PAGE and prints any heading statement on the current output device.

COMMAND SYNTAX

PAGE {expression}

SYNTAX ELEMENTS

If expression is specified it should evaluate to a numeric integer, which will cause the page number after the page throw to be set to this value.

EXAMPLES

```
HEADING "10 PAGE REPORT"

FOR I = 1 TO 10

    PAGE
    GOSUB PrintPage

NEXT I
```

PAUSE

The PAUSE statement allows processing to be suspended until an external event triggered by a WAKE statement from another process or a timeout occurs.

COMMAND SYNTAX

PAUSE {expression}

SYNTAX ELEMENTS

expression may evaluate to a timeout value, which is the maximum number of seconds to suspend the process. If expression is omitted then the PAUSE statement will cause the process to suspend until woken by the WAKE statement.

I f a timeout value is specified and the suspended process is not woken by a WAKE statement then the process will continue once the timeout period has expired.

If executing a WAKE statement for the process before the process executes the PAUSE statement then the PAUSE will be ignored and processing will continue until a subsequent PAUSE statement.

PCPERFORM

PCPERFORM is synonymous with and PERFORM.

PERFORM

PERFORM is synonymous with and PERFORM.

PRECISION

The PRECISION statement informs jBASE as to the number of digits of precision it uses after the decimal point in numbers.

COMMAND SYNTAX

PRECISION integer

SYNTAX ELEMENTS

integer should be in the range 0 to 9.

NOTES

A PRECISION statement can be specified any number of times in a source file. Only the most recently defined precision will be active at any one time.
Calling programs and external subroutines do not have to be compiled at the same degree of precision, however, any changes to precision in a subroutine will not persist when control returns to the calling program.
jBASE uses the maximum degree of precision allowed on the host machine in all mathematical calculations to ensure maximum accuracy. It then uses the defined precision to format the number.

EXAMPLES

```
PRECISION 6

CRT 2/3
```

will print the value 0.666666 (note: truncation not rounding!).

PRINT

The PRINT statement sends data directly to the current output device, which will be either the terminal or the printer.

COMMAND SYNTAX

PRINT expression {, expression...} {:}

SYNTAX ELEMENTS

An expression can evaluate to any data type. The PRINT statement will convert the result to a string type for printing. Expressions separated by commas will be sent to the output device separated by a tab character.
The PRINT statement will append a newline sequence to the final expression unless it is terminated with a colon ":" character.

NOTES

As the expression can be any valid expression, it may have output formatting applied to it.
If a PRINTER ON statement is currently active then output will be sent to the currently assigned printer form queue.
See also: SP-ASSIGN command and CRT.

EXAMPLES

```
PRINT A "L#5"


PRINT @ (8,20):"Patrick":
```

PRINTER

Use the PRINTER statement to control the destination of output from the PRINT statement.

COMMAND SYNTAX

PRINTER ON|OFF|CLOSE

NOTES

PRINTER ON redirects all subsequent output from the PRINT statement to the print spooler.
PRINTER OFF redirects all subsequent output from the PRINT statement to the terminal device.
PRINTER CLOSE will act as PRINTER OFF but in addition closes the currently active spool job created by the active PRINTER ON statement.

EXAMPLES

```
PRINTER ON;* Open a spool job

FOR I =1 TO 60

    PRINT "Line ":I ;* Send to printer

    PRINTER OFF

    PRINT "+": ;* Send to terminal

    PRINTER ON ;* Back to printer

NEXT I

PRINTER CLOSE ;* Allow spooler to print it
```

PRINTERR

Use PRINTERR to print standard jBASE error messages

COMMAND SYNTAX

PRINTERR expression

SYNTAX ELEMENTS

Field 1 of the expression should evaluate to the numeric or string name of a valid error message in the jBASE error message file. If the error message requires parameters then these can be passed to the message as subsequent fields of the expression.

INTERNATIONAL MODE

When the PRINTERR statement is used in International Mode, the error message file to be used, i.e. the default “jBASICmessages” or other as configured via the error message environment variable, will be suffixed with the current locale. For example, if the currently configured locale is “fr_FR” then the statement will attempt to find the specified error message record id in the “jBASICmessages_fr_FR” error message file. If the file cannot be found then the country code will be discarded and just the language code used. i.e. the file “jBASICmessages_fr” will be used. If this file is also not found then the error message file “jBASICmessages” will be used.

NOTES

The PRINTERR statement is most useful for user-defined messages that have been added to the standard set.
You should be very careful when typing this statement it is very similar to the PRINTER statement. Although this is not ideal, the PRINTERR statement must be supported for compatibility with older systems.

EXAMPLES

```
PRINTERR 201:CHAR (254):"CUSTOMERS"
```

PROCREAD

Use PROCREAD to retrieve data passed to programs from a jCL program.

COMMAND SYNTAX

PROCREAD variable THEN|ELSE statements

SYNTAX ELEMENTS

variable is a valid jBASE BASIC identifier, which will be used to store the contents of the primary input buffer of the last jCL program called.
If a jCL program did not initiate the program the PROCREAD will fail and executes any statements associated with an ELSE clause. If the program was initiated by a jCL program then the PROCREAD will succeed, the jCL primary input buffer will be assigned to variable and any statements associated with a THEN clause will be executed.

NOTES

It is recommended that the use of jCL and therefore the PROCREAD statement should be not be expanded within your application and gradually replaced with more sophisticated methods such as UNIX scripts or jBASE BASIC programs.

EXAMPLE

```
PROCREAD Primary ELSE

 CRT "Unable to read the jCL buffer"

 STOP
END
```

PROCWRITE

Use PROCWRITE to pass data back to the primary input buffer of a calling jCL program.

COMMAND SYNTAX

PROCWRITE expression

SYNTAX ELEMENTS

expression may evaluate to any valid data type.

NOTES

See also: PROCREAD

EXAMPLES

PROCWRITE "Success":CHAR (254):"0"

PROGRAM

PROGRAM performs no function other than to document the source code

COMMAND SYNTAX

PROGRAM progname

SYNTAX ELEMENTS

Progname can be any string of characters.

EXAMPLES

```
PROGRAM HelpUser

!
! Start of program
```

PROMPT

Used to change the PROMPT character used by terminal input commands

COMMAND SYNTAX

PROMPT expression

SYNTAX ELEMENTS

expression can evaluate to any printable string.

NOTES

The entire string is used as the prompt.

The default prompt character is the question mark "?" character.

EXAMPLE

```
PROMPT "Next answer : "


INPUT Answer
```

PUTENV

Use PUTENV to set environment variables for the current process.

COMMAND SYNTAX

PUTENV (expression)

SYNTAX ELEMENTS

expression should evaluate to a string of the form:
EnvVarName=value

where

EnvVarName is the name of a valid environment variable and value is any string that makes sense to variable being set.
If PUTENV function succeeds it returns a Boolean TRUE value, if it fails it will return a Boolean FALSE value.

NOTES

PUTENV only sets environment variables for the current process and processes spawned (say by EXECUTE) by this process. These variables are known as export only variables.
See also: GETENV

EXAMPLE

```
IF PUTENV("JBASICLOGNAME=":UserName) THEN

 CRT "Environment configured"

END
```

PWR

The PWR function raises a number to the n'th power.

COMMAND SYNTAX

PWR (expression1, expression2)

or

expression1 ^ expression2

SYNTAX ELEMENTS

Both expression1 and expression2 should evaluate to numeric arguments. The function will return the value of expression1 raised to the value of expression2.

NOTES

If expression1 is negative and expression2 is not an integer then a maths library error is displayed and the function returns the value 0. The error message displayed is:

pow: DOMAIN error

All calculations are performed at the maximum precision supported on the host machine and truncated to the compiled precision on completion.

EXAMPLES

```
A = 2

B = 31

CRT "2 GB is ":A^B
```

or

```
CRT "2 GB is": PWR (A, B)
```

QUOTE / DQUOTE / SQUOTE

These three functions will put a single or double quotation mark and the beginning and end of a string.

COMMAND SYNTAX

QUOTE(expression)

DQUOTE(expression)

SQUOTE(expression)

SYNTAX ELEMENTS

expression may be any expression that is valid in the JBASE BASIC language.

NOTES

The QUOTE and DQUOTE functions will enclose the value in double quotation marks. The SQUOTE function will enclose the value in single quotation marks.

RAISE

The RAISE function raises system delimiters in a string to the next highest delimiter.

COMMAND SYNTAX

RAISE (expression)

SYNTAX ELEMENTS

The expression is a string containing one or more delimiters, which are raised as follows:

ASCII Character	Raised To
248	249
249	250
250	251
251	252
252	253
253	254
254	255

EXAMPLE

AttributeDelimitedVariable = RAISE(ValuemarkDelimitedVariable)

READ

The READ statement allows a program to read a record from a previously opened file into a variable.

COMMAND SYNTAX

READ variable1 FROM { variable2,} expression {SETTING setvar} {ON ERROR statements} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the identifier into which the record will be read.
variable2, if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then the default file is assumed.
The expression should evaluate to a valid record key for the file.
If the SETTING clause is specified and the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

If you wish to set a lock on a record, you should do so explicitly with the READU statement.

EXAMPLE 1

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"

END
READ Rec FROM DCusts, "Xref" THEN

    READ DataRec FROM Rec<7> ELSE

        ABORT 202, Rec<7>

    END
END ELSE
```

```
    ABORT 202, "Xref"
END
```

EXAMPLE 2

```
READ record FROM filevar, id SETTING errorNumber ON ERROR

    PRINT errorNumber
END THEN
    PRINT 'Record read successfully'
END ELSE
    PRINT 'Record not on file'
END
```

READBLK

Use the READBLK statement to read a block of data of a specified length from a file opened for sequential processing and assigns it to a variable.

COMMAND SYNTAX

READBLK variable FROM file.variable, blocksize
{ THEN statements [ELSE statements] | ELSE statements }
The READBLK statement reads a block of data beginning at the current position in the file and continuing for blocksize bytes and assigns it to variable. The current position is reset to just beyond the last readable byte.
file.variable specifies a file previously opened for sequential processing.
If the data can be read from the file, the THEN statements are executed; any ELSE statements are ignored. If the file is not readable or if the end of file is encountered, the ELSE statements are executed and the THEN statements are ignored. If the ELSE statements are executed, variable is set to an empty string. If either file.variable or blocksize evaluates to null, the READBLK statement fails and the program enters the debugger.
NOTE: A new line in UNIX files is one byte long, whereas in Windows NT it is two bytes long. This means that for a file with newlines, the same READBLK statement may return a different set of data depending on the operating system the file is stored under.
The difference between the READSEQ statement and the READBLK statement is that the READBLK statement reads a block of data of a specified length, whereas the READSEQ statement reads a single line of data.

EXAMPLE

```
OPENSEQ 'MYSLIPPERS', 'PINK' TO FILE ELSE ABORT

READBLK VAR1 FROM FILE, 50 THEN PRINT VAR1

PRINT
READBLK VAR2 FROM FILE, 100 THEN PRINT VAR2
```

INTERNATIONAL MODE

When using the READBLK statement in International Mode, care must be taken to ensure that the input variable is handled properly subsequent to the READBLK statement. The READBLK statement requires that a “bytecount” be specified, however when manipulating variables in International Mode character length rather than byte lengths are usually used and hence possible confusion or program malfunction can occur.
If requiring character data convert the input variable from ‘binary/latin1’ to UTF-8 byte sequence via the UTF8 function.

It is recommended that the READBLK/WRITEBLK statements not be used when executing in International Mode. Similar functionality can be obtained via the READSEQ/WRITESEQ statement, which can be used to read/writecharacters a line at a time from a file.

READL

The READL statement allows a process to read a record from a previously opened file into a variable and takes a read-only shared lock on the record. It respects all records locked with the READU statement but allows other processes using READL to share the same lock.

COMMAND SYNTAX

READL variable1 FROM {variable2,} expression {SETTING setvar} {ON ERROR statements} {LOCKED statements} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the identifier into which the record will be read.
variable2, if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement if variable2 is not specified then the default file is assumed.
The expression should evaluate to a valid record key for the file.
If the **SETTING** clause is specified and the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

READL takes a read-only shared record lock whereas READU takes an exclusive lock. This means that any record, which is read using READL, can also be read by another process using a READL. In other words, the lock on the record is 'shared' in that no READU lock against the same record can be taken. Similarly, if a READU takes a lock then READL will respect that lock. By comparison, a READU takes an exclusive lock in that the one process retains control over the record.
The usage of READU is already well documented and understood. The usage of READL allows for an application to present a record to one or more users such that its integrity is ensured, i.e. the user(s) viewing the record can be assured that wysiwyg and that no updates to that record have been made whilst viewing the record.
While it is permissible to WRITE a record that has a READL lock, the intent of READL is to permit a 'read-only' shared lock and the act of WRITEing this record would not be considered good programming practice.

READ takes no lock at all and does not respect any lock taken with READU or READL. In other words, a READ can be performed at any time and on any record regardless of any existing locks.
Due to limitations on Windows platforms, the READL statement behaves the same as the READU statement, in other words they both take exclusive locks.
If the record could not be read because another process already had a READU lock on the record then one of two actions is taken. If the LOCKED clause was specified in the statement then the statements dependent on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the other process releases the lock. The SYSTEM (43) function can be used to determine which port has the lock.
If the statement fails to read the record then any statements associated with the ELSE clause will be executed. If the statement successfully reads the record then the statements associated with any THEN clause are executed. Either or both of THEN and ELSE clauses must be specified with the statement.
The lock taken by the READL statement will be released by any of the following events however, be aware that the record will not be fully released until all shared locks have been released:
The same program with WRITE, WRITEV or MATWRITE statements writes to the record.
The same program with the DELETE statement deletes the record.
The record lock is released explicitly using the RELEASE statement.
The program stops normally or abnormally.
When a file is OPENed to a local file variable in a subroutine then the file is closed when the subroutine RETURNS so all locks taken on that file are released, including locks taken in a calling program. Files that are opened to COMMON variables are not closed so the locks remain intact.
See also: WRITE, WRITEU, MATWRITE, MATWRITEU, RELEASE, and DELETE

READLIST

READLIST allows the program to retrieve a previously stored list (perhaps created with the SAVE-LIST command), into a jBASE BASIC variable.

COMMAND SYNTAX

READLIST variable1 FROM expression {SETTING variable2} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the variable into which the list will be read.
expression should evaluate to the name of a previously stored list to retrieve. If specified, variable2 will be set to the number of elements in the list.
If the statement succeeds in retrieving the list, then the statements associated with any THEN clause will be executed. If the statement fails to find the list, then the statements associated with any ELSE clause will be executed.

NOTES

The READLIST statement is identical in function to the GETLIST statement.
See also: DELETELIST, FORMLIST, WRITELIST

EXAMPLES

Find the list first

```
READLIST MyList FROM "MyList" ELSE STOP

LOOP
* Loop until there are no more elements

WHILE READNEXT Key FROM MyList DO

......
REPEAT
```

READNEXT

READNEXT retrieves the next element in a list variable.

COMMAND SYNTAX

READNEXT variable1, variable2 {FROM variable3} {SETTING setvar} {THEN|ELSE statements}

SYNTAX ELEMENTS

variable1 is the variable into which the next element of the list will be read.
variable2 is used when the list has been retrieved externally from a SSELECT or similar jBASE command that has used an exploding sort directive. When specified, this variable will be set to the multi-value reference of the current element. For example, if the SSELECT used a BY-EXP directive on field 3 of the records in a file, the list will contain each record key in the file as many times as there are multi-values in the field. Each READNEXT instance will set variable2 to the multi-value in field 3 to which the element refers. This allows the multi-values in field 3 to be retrieved in sorted order.
If variable3 is specified with the FROM clause, the READNEXT operates on the list contained in variable3. If variable3 is not specified, the default select list variable will be assumed.
If the SETTING clause is specified and the read (to build the next portion of the list) fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

READNEXT can be used as an expression returning a Boolean TRUE or FALSE value. If an element is successfully read from the list, TRUE is returned. If the list was empty, FALSE is returned.
See also: SELECT, extensions for secondary indexes.

EXAMPLE

```
LOOP
WHILE READNEXT Key FROM RecordList DO

......
REPEAT
```

READPREV

This statement is syntactically similar to the READNEXT but it works in reverse order. There are some considerations when the direction is changed from a forward search to a backward search or vice-versa. When a SELECT statement is first executed a forward direction is assumed. Therefore if a SELECT is immediately followed by a READPREV, then a change of direction is assumed.
During the READNEXT or READPREV sequence a next-key pointer is kept up to date. This is the record key, or index key to use should a READNEXT be executed.
During a change of direction from forward (READNEXT) to backward (READPREV) then the next record key or index key read in by the READPREV will be the one preceding the next-key pointer. When the select list is exhausted it will either point one before the start of the select list (if READPREVs have been executed) or one past the end of the select list (if READNEXTs have been executed). Thus in the event of a change of direction the very first or very last index key or record key will be used.

EXAMPLE

Consider the following jBASE BASIC code

```
list = "DAVE" : : "GREG" : : "JIM"
SELECT list
```

The following table shows what happens if you do READNEXTs and READPREVs on the above code and the reasons for it.

Statement executed	Result of operation	Comment
READNEXT key ELSE	key becomes "DAVE"	First key in list
READNEXT key ELSE	key becomes "GREG"	Second key in list
READPREV key ELSE	key becomes "DAVE"	Reversed so take preceding key
READPREV key ELSE	Take ELSE clause	The next key ptr exhausted at start.
READNEXT key ELSE	key becomes "DAVE"	First key in list
READNEXT key ELSE	key becomes "GREG"	Second key in list
READNEXT key ELSE	key becomes "JIM"	Final key. Next key ptr exhausted.
READPREV key ELSE	key becomes "JIM"	Reversed but list exhausted.
READPREV key ELSE	key becomes "GREG"	Second key in list

READPREV key ELSE key becomes "DAVE" First key in list

READSELECT

See also:READLIST.

READSEQ

Read from a file opened for sequential access.

COMMAND SYNTAX

READSEQ Variable FROM FileVar THEN | ELSE statements

SYNTAX ELEMENTS

Variable specifies the variable to contain next record from sequential file.

FileVar specifies the file descriptor of the file opened for sequential access.

Statements Conditional jBASE BASIC statements

NOTES

Each READSEQ reads a line of data from the sequentially opened file. After each READSEQ, the file pointer moves forward to the next line of data. The variable contains the line of data less the new line character from the sequential file.

The default buffer size for a READSEQ is 1024 bytes. This can be changed using the IOCTL () function with the JIOCTL_COMMAND_SEQ_CHANGE_RECORDSIZE Sequential File Extensions.

EXAMPLES

See also: Sequential File Examples

READT

The READT statement is used to read a range of tape devices 0-9.

COMMAND SYNTAX

READT variable {FROM expression} THEN|ELSE statements

SYNTAX ELEMENTS

variable is the variable that will receive any data read from the tape device.
expression should evaluate to an integer value in the range 0-9 and specifies from which tape channel to read data. If the FROM clause is not specified the READT will assume channel 0.
If the READT fails then the statements associated with any ELSE clause will be executed. SYSTEM (0) will return the reason for the failure as follows:

1 There is no media attached to the channel
2 An end of file mark was found.

NOTES

A "tape" does not only refer to magnetic tape devices, but also any device that has been described to jBASE. Writing device descriptors for jBASE is beyond the scope of this manual.
If no tape device has been assigned to the specified channel the jBASE debugger is entered with an appropriate message.
Each instance of the READT statement will read the next record available on the device. The record size is not limited to a single tape block and the entire record will be returned whatever block size has been allocated by the T-ATT command.

EXAMPLE

```
LOOP
    READT TapeRec FROM 5 ELSE

        Reason = SYSTEM(0)

        IF Reason = 2 THEN BREAK ;* done

        CRT "ERROR"; STOP

    END
REPEAT
```

READU

The READU statement allows a program to read a record from a previously opened file into a variable. It respects record locking and locks the specified record for update.

COMMAND SYNTAX

READU variable1 FROM {variable2,} expression {SETTING setvar} {ON ERROR statements} {LOCKED statements} THEN|ELSE statements

SYNTAX ELEMENTS

Variable1 is the identifier into which the record will be read.
variable2 if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then the default file is assumed.
The expression should evaluate to a valid record key for the file.
If the SETTING clause is specified and the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

If the record could not be read because another process already had a lock on the record then one of two actions is taken. If the LOCKED clause was specified in the statement then the statements dependent on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the other process releases the lock. Use the SYSTEM (43) function to determine which port has the lock.
If the statement fails to read the record then any statements associated with the ELSE clause will be executed. If the statement successfully reads the record then the statements associated with any THEN clause are executed. Either or both of THEN and ELSE clauses must be specified with the statement.

The lock taken by the READU statement will be released by any of the following events:
The same program with WRITE, WRITEV or MATWRITE statements writes to the record.
The same program with the DELETE statement deletes the record.
The record lock is released explicitly using the RELEASE statement.
The program stops normally or abnormally.

When a file is OPENed to a local file variable in a subroutine then the file is closed when the subroutine RETURNS so all locks taken on that file are released, including locks taken in a calling program. Files that are opened to COMMON variables are not closed so the locks remain intact.

See also: WRITE, WRITEU, MATWRITE, MATWRITEU, RELEASE, and DELETE

EXAMPLES

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"

END
LOOP
    READU Rec FROM DCusts, "Xref" LOCKED

        CRT "Xref locked by port ":SYSTEM(43):" - retrying"

        SLEEP 1; CONTINUE ;* Restart LOOP

    END THEN

        READ DataRec FROM Rec ELSE

            ABORT 202, Rec

        END
        BREAK ;* Leave the LOOP

    END ELSE

        ABORT 202, "Xref"

    END
REPEAT
```

READV

The READV statement allows a program to read a specific field from a record in a previously opened file into a variable.

COMMAND SYNTAX

READV variable1 FROM { variable2,} expression1, expression2 {SETTING setvar} {ON ERROR statements} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the identifier into which the record will be read.
variable2 if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified, the default file is assumed.
expression1 should evaluate to a valid record key for the file.
expression2 should evaluate to a positive integer. If the number is invalid or greater than the number of fields in the record, a NULL string will be assigned to variable1. If the number is 0, then the readv0 emulation setting controls the value returned in variable1. If a non-numeric argument is evaluated, a run time error will occur.
If the SETTING clause is specified and the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

If you wish to set a lock on a record, do so explicitly with the READU or READVU statement. To read a field from a previously opened file into a variable and take a read-only shared lock on the field, use READVL.

EXAMPLE

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"
```

```
END
READV Rec FROM DCusts, "Xref",7 THEN

    READ DataRec FROM Rec<7> ELSE

        ABORT 202, Rec<7>

    END
END ELSE

    ABORT 202, "Xref"

END
```

READVL

Use the READVL statement to acquire a shared record lock and then read a field from the record.

The READVL statement conforms to all the specifications of the READL and READV statements.

READVU

The READVU statement allows a program to read a specific field in a record in a previously opened file into a variable. It also respects record locking and locks the specified record for update.

COMMAND SYNTAX

READVU variable1 FROM { variable2,} expression1, expression2 {SETTING setvar} {ON ERROR statements} {LOCKED statements} THEN|ELSE statements

SYNTAX ELEMENTS

variable1 is the identifier into which the record will be read.
variable2 if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then the default file is assumed.
expression1 should evaluate to a valid record key for the file.
expression2 should evaluate to a positive integer number. If the number is invalid or greater than the number of fields in the record, then a NULL string will be assigned to variable1. If the number is 0, then the readv0 emulation setting controls the value returned in variable1. If a non-numeric argument is evaluated a run time error will occur.
If the SETTING clause is specified and the read fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

If ON ERROR is specified, the statements following the ON ERROR clause will be executed for any of the above Incremental File Errors except error 128.

NOTES

If the record could not be read because another process already had a lock on the record then one of two actions is taken. If the LOCKED clause was specified in the statement then the statements dependent on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the other process releases the lock.
If the statement fails to read the record then any statements associated with the ELSE clause are executed. If the statement successfully reads the record then the statements associated with any THEN clause are executed. Either or both of the THEN and ELSE clauses must be specified with the statement.
The lock taken by the READVU statement will be released by any of the following events:
The same program with WRITE, WRITEV, MATWRITE or DELETE statements writes to the record.
The record lock is released explicitly using the RELEASE statement.

The program stops normally or abnormally.

When a file is OPENed to a local file variable in a subroutine then the file is closed when the subroutine RETURNS so all locks taken on that file are released, including locks taken in a calling program. Files that are opened to COMMON variables are not closed so the locks remain intact.

See also: WRITE, WRITEU, MATWRITE, MATWRITEU, RELEASE, and DELETE

EXAMPLE

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"

END
LOOP
    READVU Rec FROM DCusts, "Xref",7 LOCKED

        CRT "Locked - retrying"

        SLEEP 1; CONTINUE ;* Restart LOOP

    END THEN

        READ DataRec FROM Rec ELSE

            ABORT 202, Rec

        END
        BREAK ;*leave the LOOP

    END ELSE

        ABORT 202, "Xref"

    END
REPEAT
```

READXML

READXML rec FROM file, id ELSE STOP 202,id

Reads a record from a file using the style sheet held in DICT->@READXML to transform the data into xml format

EXAMPLE

```
READ rec FROM file,id THEN
      CRT rec
   END
   READXML xml FROM file,id THEN
      CRT xml
   END
```

Screen output

```
CLIVE^PIPENSLIPPERS^999 LETSBE AVENUE
…
<?xml version="1.0" encoding="UTF-8"?>

<mycustomer>
<firstname>CLIVE</firstname>
<lastname>PIPENSLIPPERS</lastname>
<address>999 LETSBE AVENUE</address>
```

RECORDLOCKED

Call the RECORDLOCKED function to ascertain the status of a record lock.

COMMAND SYNTAX

RECORDLOCKED (filevar, recordkey)

SYNTAX ELEMENTS

filevar is a file variable from a previously executed OPEN statement.
recordkey is an expression for the record id that will be checked.

NOTES

RECORDLOCKED returns an integer value to indicate the record lock status of the specified record id.

3	Locked by this process by a FILELOCK
2	Locked by this process by a READU
1	Locked by this process by a READL
0	Not locked
-1	Locked by another process by a READL
-2	Locked by another process by a READU
-3	Locked by another process by a FILELOCK

If the return value is negative, then the SYSTEM(43) and STATUS function calls can be used to determine the port number of the program that holds the lock. If -1 is returned, more than 1 port could hold the lock and so the port number returned will be the first port number found.]

EXAMPLE

```
OPEN "INVENTORY" TO invFvar ELSE ABORT 201,"Cannot open the INVENTORY
file"
...
...
IF RECORDLOCKED (invFvar,invId) = -2 THEN

    CRT "Inventory record ":invId:" is locked by port ":SYSTEM(43)

END
```

REGEXP

The REGEXP function is a powerful function that allows pattern matching using UNIX regular expressions. REGEXP is not supported on Windows.

COMMAND SYNTAX

REGEXP(variable, expression)

SYNTAX ELEMENTS

variable can be any type of jBASE BASIC variable and is the variable upon which pattern matching will be performed.
expression should evaluate to a standard UNIX regular expression as defined in the UNIX documentation.

NOTES

The function returns a numeric integer value being the first character in variable that failed to match the specified regular expression. If a match is not found or the regular expression was invalid then the function returns 0.

EXAMPLE

String = "jBASE Software Inc."
CRT REGEXP(String, "S[^t]*")

displays the value 4 being the position of the character "t" in the word Software

RELEASE

The RELEASE statement enables a program to explicitly release record locks without updating the records using WRITE.

COMMAND SYNTAX

RELEASE {{variable,} expression}

SYNTAX ELEMENTS

If variable is specified it should be a valid file descriptor variable (i.e. It should have been the subject of an OPEN statement)
If an expression is supplied it should evaluate to the record key of a record whose lock the program wishes to free. If variable was specified the record lock in the file described by it is released. If variable was not specified the record lock in it releases the file described by the default file variable
If RELEASE is issued without arguments then all record locks in all files that were set by the current program will be released.

NOTES

Where possible the program should avoid the use of RELEASE without arguments; this is less efficient and can be dangerous - especially in subroutines.

EXAMPLE

```
READU Rec FROM File, "Record" ELSE ABORT 203, "Record"

IF Rec<1> = "X" THEN

RELEASE File, "Record"

END
......
```

REMOVE

REMOVE will successively extract delimited strings from a dynamic array.

COMMAND SYNTAX

REMOVE variable FROM array SETTING setvar

SYNTAX ELEMENTS

variable is the variable, which is to receive the extracted string.
array is the dynamic array from which the string is to be extracted.
setvar is set by the system during the extraction to indicate the type of delimiter found:

0	end of the array	
1	xFF ASCII 255	
2	xFE ASCII 254	Field marker
3	xFD ASCII 253	Value marker
4	xFC ASCII 252	Subvalue marker
5	xFB ASCII 251	
6	xFA ASCII 250	
7	xF9 ASCII 249	

NOTES

The first time the REMOVE statement is used with a particular array, it will extract the first delimited string it and set the special "remove pointer" to the start of the next string (if any). The next time REMOVE is used on the same array, the pointer will be used to retrieve the next string and so on. The array is not altered.
The variable named in the SETTING clause is used to record the type of delimiter that was found - so that you can tell whether the REMOVE statement extracted a field, a value or a subvalue for example. Delimiters are defined as characters between xF9 and xFF only. Once the end of the array has been reached, the string variable will not be updated and the SETTING clause will always return 0. You can reset the "remove pointer" by assigning the variable to itself - for example REC = REC.

EXAMPLE

```
EQU FM TO CHAR (254), VM to CHAR(253), SVM to CHAR(252)

REC = "Field 1":FM:"Value 1":VM:" Value 2":FM:"Field 3"
```

```
REMOVE EXSTRING FROM REC SETTING DELIM

REMOVE EXSTRING FROM REC SETTING DELIM
```

The first time REMOVE is used, EXSTRING will contain "Field 1" and DELIM will contain xFE. The second time REMOVE is used, EXSTRING will contain "Value 1" and DELIM will contain xFD.

REPLACE

REPLACE is an obsolete way to assign to dynamic arrays via a function.

COMMAND SYNTAX

REPLACE (var, expression1{, expression2{, expression3}}; expression4)

SYNTAX ELEMENTS

var is the dynamic array that the REPLACE function will use to assign expression4. Unless the same var is assigned the result of the function remains unchanged.
expression1 specifies into which field assignment will be made and should evaluate to a numeric.
expression2 is only specified when multi-value assignment is to be done and should evaluate to a numeric.
expression3 is only specified when sub-value assignment is to be done and should evaluate to a numeric.
expression4 can evaluate to any data type and is the actual data that will be assigned to the array.

NOTES

The function returns a copy of var with the specified replacement carried out. This value may be assigned to the original var in which case the jBASE BASIC compiler will optimize the assignment.

EXAMPLES

```
X = "JBASE":MV:"is Great"

X = REPLACE (X,1,1;"jBASE")
```

RETURN

The RETURN statement transfers program execution to the caller of a subroutine/function or to a specific label in the program.

COMMAND SYNTAX

RETURN {TO label}
or
RETURN (expression)

SYNTAX ELEMENTS

label must reference an existing label within the source of the program.
expression evaluates to the value that is returned by a user-written function.

NOTES

The RETURN statement will transfer program execution to the statement after the GOSUB that called the current internal subroutine.
If the RETURN statement is executed in an external SUBROUTINE and there are no outstanding GOSUBs, then the program will transfer execution back to the program that called it via CALL.
The program will enter the debugger with an appropriate message should a RETURN be executed with no GOSUB or CALL outstanding.
The second form of the RETURN statement is used to return a value from a user-written function. This form can only be used in a user-written function.

REWIND

The REWIND statement will issue a rewind command to the device attached to the specified channel.

COMMAND SYNTAX

REWIND {ON expression} THEN|ELSE statements

SYNTAX ELEMENTS

expression, if specified, should evaluate to an integer in the range 0 to 9. Default is 0.

NOTES

If the statement fails to issue the rewind then any statements associated with the ELSE clause are executed. If the statement successfully issues the rewind command then the statements associated with any THEN clause are executed. Either or both of the THEN and ELSE clauses must be specified with the statement.
If the statement fails then the reason for failure can be determined via the value of SYSTEM(0) as follows:

Value	Meaning
1	there is no media attached to the channel
2	an end of file mark was found

RIGHT

The RIGHT function returns a sub-string composed of the last n characters of a specified string.

COMMAND SYNTAX

RIGHT (expression, length)

SYNTAX ELEMENTS

expression evaluates to the string from, which the sub string is extracted.

length is the number of characters that are extracted. If length is less than 1, RIGHT () returns null.

NOTES

The RIGHT () function is equivalent to sub-string extraction for the last n characters, i.e. expression[n]

See also: LEFT().

EXAMPLE

S = "The world is my lobster"

```
CRT DQUOTE (RIGHT (S,7))

CRT DQUOTE(RIGHT(S,99))

CRT DQUOTE(RIGHT(S,0))
```

This code displays:

"lobster"

"The world is my lobster"

""

RND

The RND function allows the generation of random numbers by a program.

COMMAND SYNTAX

RND (expression)

SYNTAX ELEMENTS

expression should evaluate to a numeric integer value or a runtime error will occur. The absolute value of expression is used by the function. The highest number expression can be on Windows is PWR(2,15) - 1. The highest number on UNIX is PWR(2,31) - 1.
See also:ABS

NOTES

The function will return a random integer number between 0 and the value of expression-1.

EXAMPLE

```
FOR I=1 TO 20

   CRT RND (100):", ":

NEXT I
```

prints 20 random numbers in the inclusive range 0 to 99.

RQM

RQM is synonymous with SLEEP.

RTNDATA

The RTNDATA statement allows a jBASE BASIC program to return specific data to the RTNDATA clause of another program's EXECUTE statement.

COMMAND SYNTAX

RTNDATA expression

SYNTAX ELEMENTS

expression may evaluate to any data type.

NOTES

When a jBASE BASIC program executes another jBASE BASIC program using the EXECUTE statement it may specify a variable to pick up data in using the RTNDATA clause. The data picked up will be that specified by the executed program using the RTNDATA statement.
The data will be discarded if the program is not executed by an EXECUTE statement in another program.

SADD

See also: Floating point Operations

The SADD function performs string addition of two base 10-string numbers.

COMMAND SYNTAX

SADD (expr1, expr2)

SYNTAX ELEMENTS

expr1 and expr2 are strings consisting of numeric characters, optionally including a decimal part.

NOTES

The SADD function can be used with numbers that may exceed a valid range with standard arithmetic operators.

The PRECISION declaration has no effect on the value returned by SADD.

EXAMPLE

```
A = 4000000000000000000000000000000

B = 7

CRT SADD (A,B)
```

Displays 4000000000000000000000000000007 to the screen

```
CRT SADD (4.33333333333333333,1.8)
```

Displays 6.13333333333333333 to the screen

SDIV

See also: Floating point Operations

The SDIV function performs a string division of two base 10-string numbers and rounds the result to 14 decimal places.

COMMAND SYNTAX

SDIV (expr1, expr2)

SYNTAX ELEMENTS

expr1 and expr2 are strings consisting of numeric characters, with either optionally including a decimal part.

NOTES

Use the SDIV function with numbers that may exceed a valid range with standard arithmetic operators. The PRECISION declaration has no effect on the value returned by SDIV.

EXAMPLE

```
A = 2

B = 3

CRT SDIV (A,B)
```

Displays 0.66666666666666 to the screen

```
CRT SDIV (355,113)
```

Displays 3.14159292035398 to the screen

SEEK

Use the SEEK statement to move the file pointer by an offset specified in bytes, relative to the current position, the beginning of the file, or the end of the file.

COMMAND SYNTAX

SEEK file.variable [, offset [, relto]]
{THEN statements [ELSE statements] | ELSE statements}

file.variable specifies a file previously opened for sequential access.
offset is the number of bytes before or after the reference position. A negative offset results in the pointer being moved before the position specified by relto. If offset is not specified, 0 is assumed.

NOTE: On Windows NT systems, line endings in files are denoted by the character sequence RETURN + LINEFEED rather than the single LINEFEED used in UNIX files. The value of offset should take into account this extra byte on each line in Windows NT file systems.
The permissible values of relto and their meanings follow:

0 Relative to the beginning of the file
1 Relative to the current position
2 Relative to the end of the file
If relto is not specified, 0 is assumed.

If the pointer is moved, the THEN statements are executed and the ELSE statements are ignored. If the THEN statements are not specified, program execution continues with the next statement.
If the file cannot be accessed or does not exist the ELSE statements are executed; any THEN statements are ignored.
If file.variable, offset, or relto evaluates to null, the SEEK statement fails and the program terminates with a run-time error message.

Note: On Windows NT systems, if you use the OPENDEV statement to open a 1/4-inch cartridge tape (60 MB or 150 MB) for sequential processing, you can move the file pointer only to the beginning or the end of the data. For diskette drives, you can move the file pointer only to the start of the data. Seeking beyond the end of the file and then writing creates a gap, or hole, in the file. This hole occupies no physical space, and reads from this part of the file return as ASCII CHAR 0 (neither the number nor the character 0).
For more information about sequential file processing, See also: OPENSEQ, READSEQ, and WRITESEQ statements.

EXAMPLE

The following example reads and prints the first line of RECORD4. Then the SEEK statement moves the pointer five bytes from the front of the file, then reads and prints the rest of the current line.

```
OPENSEQ '.', 'MYSEQFILE' TO FILE ELSE ABORT

READSEQ B FROM FILE THEN PRINT B

SEEK FILE,5, 0 THEN

READSEQ A FROM FILE THEN PRINT A ELSE ABORT

END
```

The output of this program is:

```
FIRST LINE


LINE
```

SELECT

The SELECT statement creates a select list of elements in a specified variable.

COMMAND SYNTAX

SELECT {variable1} {TO variable2 | listnum} {SETTING setvar}

SYNTAX ELEMENTS

variable1 can be an OPENed file descriptor, in which case the record keys in the specified file will be selected, or an ordinary variable in which case each field in the variable will become a list element. **variable1** may also be an existing list in which case the elements in the list will be selected.
If variable1 is not specified in the statement then it assumes the default file variable.
If variable2 is specified then the newly created list will be placed in the variable. Alternatively, specify a select list number in the range 0 to 10 with listnum. If neither variable2 nor listnum is specified then it assumes the default list variable.
If specifying the SETTING clause and the select fails, it sets setvar to one of the following values:

128	no such file or directory
4096	network error
24576	permission denied
32768	physical I/O error or unknown error

NOTES

When constructing a list from record keys in a file, it does so by extracting only the first few keys, which when removed from the list obtains the next few keys and so on. Therefore, the creation of the list is not immediate. This means that the list could contain records, written to the file after starting the SELECT command.
Consider the situation where you open a file, SELECT it and then, because of the keys obtained, write new records to the same file. It would be easy to assume that these new keys would not show up in the list because you created the list before the new records existed. This is not the case. Any records written beyond the current position in the file will eventually show up in the list. In situations where this might cause a problem, or to ensure that you obtain a complete, qualified list of keys, you should use a slower external command like jQL SELECT or SSELECT and then READNEXT to parse the file.
If using a variable to hold the select list, then it should be unassigned or null before the SELECT. If it contains a number in the range 0 to 10 then it will use the corresponding select list number to hold the list, although you can still reference the list with the variable name. This "feature" is for compatibility with older platforms. See also example 3.
Lists can be selected as many times as required.
See also: the extensions for secondary indexes.

EXAMPLE 1

```
OPEN "Customers" ELSE ABORT 201, "Customers"

SELECT TO CustList1

SELECT TO CustList2
```

EXAMPLE 2

```
OPEN "Customers" TO CustFvar ELSE ABORT 201, "Customers"

SELECT CustFvar TO 2

DONE = 0

LOOP
   READNEXT CustId FROM 2 ELSE Done = 1

UNTIL DONE DO

   GOSUB ProcessCust

REPEAT
```

EXAMPLE 3

```
CLEAR
OPEN "Customers" TO CustFvar ELSE ABORT 201, "Customers"

OPEN "Products" TO ProdFvar ELSE ABORT 201, "Products"

SELECT CustFvar TO Listvar1

SELECT ProdFvar TO Listvar2
```

This example demonstrates a coding error. The CLEAR statement is used to initialize all variables to zero. Since Listvar1 has the value 0, select list number 0 is used to hold the list. However, the CLEAR statement also initializes Listvar2 to zero, so the second SELECT overwrites the first list.

SEND

The SEND statement sends a block of data directly to a device.

COMMAND SYNTAX

SEND output {:} TO FileVar THEN | ELSE statements

SYNTAX ELEMENTS

The output is an expression evaluating to a string that will be sent to the output device (specified by FileVar). It is expected that the device has already been opened with OPENSER or OPENSEQ.
The SEND statement will append a newline sequence to the final output expression unless it is terminated with a colon ":" character.

NOTES

As the expression can be any valid expression, it may have output formatting applied to it.
The SEND syntax requires you specify either a THEN or ELSE clause, or both. It executes the THEN clause if the data is without error. Else executes, the ELSE clause if the data cannot be sent.
See also: SENDX

EXAMPLES

See also: Sequential File Processing.

SENDX

The SENDX statement sends a block of data (in hexidecimal) directly to a device.

COMMAND SYNTAX

SENDX output {:} TO FileVar THEN | ELSE statements

SYNTAX ELEMENTS

The output is an expression evaluating to a string that will be sent to the output device (specified by FileVar). It is expected that OPENSER or OPENSEQ has already opened the device .
The SENDX statement will append a newline sequence to the final output expression unless it is terminated with a colon ":" character.

NOTES

As the expression can be any valid expression, it may have output formatting applied to it.
The SENDX syntax requires a specified THEN or ELSE clause, or both. If the data is send without error, it executes the THEN clause. If the data cannot be sent, it executes the ELSE clause.
See also: SEND

EXAMPLES

See also: Sequential File Processing Examples.

SENTENCE

The SENTENCE function allows a program to locate the command used to invoke it and the arguments it was given.

COMMAND SYNTAX

SENTENCE ({expression})

SYNTAX ELEMENTS

If expression is specified it should evaluate to a positive integer value. A negative value will return a null string. A value of null will return the entire command line.
An integer value of expression will return a specific element of the command line with the command itself being returned by SENTENCE (0), the first parameter being returned by SENTENCE(1) and so on.

NOTES

It is assumed the command line arguments are space separated and when returning the entire command line they are returned as such. The SYSTEM(1000) function will return the command line attribute mark delimited.

EXAMPLES

```
DIM Parm(4)

ProgName = SENTENCE (0) ;* program is?

FOR I = 1 TO 4

   Parm(I) = SENTENCE(I) ;* get parameters

NEXT I
```

SEQ

The SEQ function returns numeric ASCII value of a character.

COMMAND SYNTAX

SEQ (expression)

INTERNATIONAL MODE

The SEQ function will return numeric values beyond 255 for UTF-8 byte sequences representing any Unicode values above 0x000000ff.

SYNTAX ELEMENTS

expression may evaluate to any data type. However, the SEQ function will convert the expression to a string and operate on the first character of that string.

NOTES

SEQ operates on any character in the integer range 0 to 255

EXAMPLES

```
EQU ENQ TO 5

* Get next comms code

* Time-out after 20 seconds

INPUT A, 1 FOR 200 ELSE BREAK

IF SEQ (A) = ENQ THEN

* Respond to ENQ char
```

SEQS

Use the SEQS function to convert a dynamic array of ASCII characters to their numeric string equivalents.

COMMAND SYNTAX

SEQS (dynamic.array)

dynamic.array specifies the ASCII characters to be converted. If dynamic.array evaluates to null, it returns null. If any element of dynamic.array is null, it returns null for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as return.array.

By using the SEQS function to convert a character outside its range results in a run-time message, and the return of an empty string.

EXAMPLE

```
G="T":@VM:"G"
A=SEQS (G)
PRINT A
PRINT SEQS("G")
```

The output of this program is: 84]71 71

INTERNATIONAL MODE

The SEQ function will return numeric values beyond 255 for UTF-8 byte sequences representing any Unicode values above 0x000000ff.

SIN

The SIN function returns the mathematical sine value of a numeric expression.

COMMAND SYNTAX

SIN (expression)

SYNTAX ELEMENTS

expression should evaluate to a numeric value and is interpreted as a number of degrees between 0 and 360.

NOTES

The function will calculate the sine of the angle specified by the expression as accurately as the host system will allow. It will then truncate the value according to the PRECISION of the program.

EXAMPLE

```
CRT @ (-1):

FOR I = 0 TO 79

    CRT @ (I,12+INT(SIN (360/80*(I+1))*10)):"*":

NEXT I
```

SLEEP

Sleep allows the program to pause execution for a specified period.

COMMAND SYNTAX

SLEEP {expression}

SYNTAX ELEMENTS

expression may evaluate to one of two forms:
Numeric in which case the statement will sleep for the specified number of seconds or fractions of a second
"nn:nn{:nn}" in which case the statement will sleep until the time specified.
If expression is not supplied then a default period of 1 second is assumed.

NOTES

Sleeping until a specified time works by calculating the time between the current time and the time supplied and sleeping for that many seconds. If in the meantime the host clock is changed the program will not wake up at the desired time;
If invoking the debugger while a program is sleeping and the execution continued, the user will be prompted:

Continue with SLEEP (Y/N)?

If "N" is the response, the program will continue at the next statement after the SLEEP.
See also: MSLEEP to sleep for a specified number of milliseconds.

EXAMPLES

```
Sleep until the end of the working day for anyone who doesn't program
computers
SLEEP "17:30”

* 40 winks...

SLEEP 40

* Sleep for two and a half seconds...

SLEEP 2.5
```

SMUL

See also: Floating Point Operations

The SMUL function performs string multiplication of two base 10-string numbers.

COMMAND SYNTAX

SMUL (expr1, expr2)

SYNTAX ELEMENTS

expr1 and **expr2** are strings consisting of numeric characters, with either optionally including a decimal part.

NOTES

Use the SMUL function with numbers that may exceed a valid range with standard arithmetic operators.

The PRECISION declaration does not affect the value returned by SMUL.

EXAMPLES

```
A = 243603310027840922

B = 3760

CRT SMUL (A,B)

Displays 915948445704681866720 to the screen

CRT SMUL (0.0000000000000475,3.61)

Displays 0.0000000000001714 to the screen
```

SORT

See also: Floating point Operations

The SORT function sorts all elements of a dynamic array in ascending left-justified order.

COMMAND SYNTAX

SORT (expression)

SYNTAX ELEMENTS

expression may evaluate to any data type but will only be useful if it evaluates to a dynamic array.

NOTES

The dynamic array can contain any number and combination of system delimiters.

The SORT () function will return an attribute-delimited array of the sorted elements.

Note: that all system delimiters in expression will be converted to an attribute mark '0xFE' in the sorted result. For example, the following code

```
MyArray = 'GEORGE':@VM:'FRED':@AM:'JOHN':@SVM:'ANDY'

 CRT SORT (MyArray)
```

will return

```
ANDY^FRED^GEORGE^JOHN
```

where '^' is an attribute mark, '0xFE'. MyArray remains unchanged.

The SORT is achieved by the quick sort algorithm, which sorts in situ and is very fast.

EXAMPLE

Read a list, sort it and write it back

```
*READ List FROM "Unsorted" ELSE List = ''

List = SORT (List)

WRITE List ON "Sorted"
```

INTERNATIONAL MODE

When using the SORT function in International Mode, the function will use the currently configured locale to determine the rules by which each string is considered less than or greater than the other for sort purposes.

SOUNDEX

The SOUNDEX function allows phonetic conversions of strings.

COMMAND SYNTAX

SOUNDEX (expression)

SYNTAX ELEMENTS

expression may evaluate to any data type but the function will only give meaningful results for English words.

NOTES

The phonetic equivalent of a string is calculated as the first alphabetic character in the string followed by a 1 to 3-digit representation of the rest of the word.
The digit string is calculated from the following table:

Characters	Value code
B F P V	1
C G J K Q S X Z	2
D T	3
L	4
M N	5
R	6

All characters not contained in the above table are ignored. The function is case insensitive and identical sequences of a character are interpreted as a single instance of the character.
The idea is to provide a crude method of identifying words such as last names even if they are not spelt correctly. The function is not foolproof should not be the sole method of identifying a word.

EXAMPLE

```
INPUT Lastname
```

Lastname = SOUNDEX (Lastname)
search the databases

SPACE

The SPACE function generates a specific number of ASCII space characters.

COMMAND SYNTAX

SPACE (expression)

SYNTAX ELEMENTS

expression should evaluate to a positive integer value.

NOTES

The SPACE function will return the specified number of ASCII space characters and is useful for padding strings. It should not be used to position output on the terminal screen as this is inefficient, accomplish this by using the @( ) function.

EXAMPLES

```
TenSpaces = SPACE (10)
```

SPACES

Use the SPACES function to return a dynamic array with elements composed of blank spaces.

COMMAND SYNTAX

SPACES (dynamic.array)

dynamic.array specifies the number of spaces in each element. If dynamic.array or any element of dynamic.array evaluates to null, the SPACES function will enter the debugger.

SPLICE

Use the SPLICE function to create a dynamic array of the element-by-element concatenation of two dynamic arrays, separating concatenated elements by the value of expression.

COMMAND SYNTAX

SPLICE (array1, expression, array2)
Each element of array1 is concatenated with expression and with the corresponding element of array2. The result is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, the element is returned properly concatenated with expression. If either element of a corresponding pair is null, null is returned for that element. If expression evaluates to null, null is returned for the entire dynamic array.

EXAMPLE

```
A="A":@VM:"B":@SM:"C"
B="D":@SM:"E":@VM:"F"
C='-'
PRINT SPLICE (A,C,B)
```

```
The output of this program is:
```

```
A-D\-E]B-F\C-
```

SPOOLER

The SPOOLER function returns information from the jBASE spooler.

COMMAND SYNTAX

SPOOLER (n{, Port|User})

SYNTAX ELEMENTS

n	Description
1	returns formqueue information
2	returns job information
3	formqueue assignment
4	returns status information

Port limits the information returned to the specified port
User limits the information returned to the specified user.

NOTES

SPOOLER(1) returns information about formqueues. The information is returned in a dynamic array, which contains an attribute for each formqueue. Each formqueue is structured as follows:

MultiValue	Description
1	Formqueue name
2	Form type
3	Device
4	Device type
5	Status
6	Number of jobs on the formqueue
7	Page skip

SPOOLER(2) returns information about print jobs. The information is returned in a dynamic array, which contains an attribute for each print job.

MultiValue	Description
1	Formqueue name
2	Print job number
3	Effective user id
4	Port number job was generated on
5	Creation date in internal format
6	Creation time in internal format
7	Job Status

MultiValue	Description
8	Options
9	Print job size (pages)
10	Copies
11	Reserved
12	Reserved
13	Reserved
14	Effective user id
15	Real user id
16	Application id as set by @APPLICATION.ID
17	JBASICLOGNAME id

SPOOLER(3) returns information about current formqueue assignments. The information is returned in a dynamic array, which contains an attribute for each assignment. Each attribute is structured as follows:

MultiValue	Description
1	Report (channel) number
2	Formqueue name
3	Options
4	Copies

SPOOLER(4) returns information about current print jobs. The information is returned in a dynamic array, which contains an attribute for each job being generated. Each attribute is structured as follows:

MultiValue	Description
1	Report (channel) number
2	Print job number
3	Print job size (pages)
4	Creation date in internal format
5	Creation time in internal format
6	Job Status
7	Effective User id
8	Real user id
9	JBASICLOGNAME id
10	Banner test from SETPTR BANNER text command

The values for Job Status are:

Status	Description
1	Queued

2	Printing
3	Finished
4	Open
5	Hold
6	Edited

SQRT

See also: Floating point Operations

The SQRT function returns the mathematical square root of a value.

COMMAND SYNTAX

SQRT (expression)

SYNTAX ELEMENTS

The expression should evaluate to a positive numeric value as the authors do not want to introduce a complex number type within the language. Negative values will cause a math error.

NOTES

The function calculates the result at the highest precision available and then truncates the answer to the required PRECISION.

EXAMPLE

```
FOR I = 1 TO 1000000

    J=SQRT (I)
NEXT I
```

SSELECT

Use the SSELECT statement to create:

A numbered select list of record IDs in sorted order from a jBASE hashed file
A numbered select list of record IDs from a dynamic array
A select list of record IDs from a dynamic array is not in sorted order.

You can then access this select list by a subsequent READNEXT statement, which removes one record ID at a time from the list.

COMMAND SYNTAX

SSELECT [variable] [TO list.number] [ON ERROR statements]
SSELECTN [variable] [TO list.number] [ON ERROR statements]
SSELECTV [variable] TO list.variable [ON ERROR statements]

variable can specify a dynamic array or a file variable. If it specifies a dynamic array, the record IDs must be separated by field marks (ASCII 254). If variable specifies a file variable, the file variable must have previously been opened. If variable is not specified, the default file is assumed. If the file is neither accessible nor open, or if variable evaluates to null, the SSELECT statement fails and the program enters the debugger with a run-time error message.
The TO clause specifies the select list that is to be used. list.number is an integer from 0 through 10. If no list.number is specified, select list 0 is used.
The record IDs of all the records in the file forms the list. The record IDs are listed in ascending order. Each record ID is one entry in the list.
Use the SSELECTV statement to store the select list in a named list variable instead of to a numbered select list. list.variable is an expression that evaluates to a valid variable name.

The ON ERROR Clause

The ON ERROR clause is optional in SSELECT statements. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of a SSELECT statement.

EXAMPLE

The following example opens the file SLIPPERS to the file variable DSCB, then creates an active sorted select list of record IDs. The READNEXT statement assigns the first record ID in the select list to the variable @ID, then prints it.

```
OPEN '','SLIPPERS' ELSE PRINT "NOT OPEN"
```

```
SSELECT
READNEXT @ID THEN PRINT @ID
```

```
The output of this program is:
```

```
0001
```

INTERNATIONAL MODE

When using the SSELECT statement in International Mode, the statement will use the currently configured locale to determine the rules by which each string is considered less than or greater than the other for sort purposes.

SSELECTN

See also: SSELECT.

SSELECTV

See also: SSELECT.

SSUB

See also: Floating Point Operations

The SSUB function performs string subtraction of two base 10-string numbers.

COMMAND SYNTAX

SSUB (expr1, expr2)

SYNTAX ELEMENTS

expr1 and expr2 are strings consisting of numeric characters, optionally including a decimal part.

NOTES

Use the SSUB function with numbers that may exceed a valid range with standard arithmetic operators.

The PRECISION declaration has no effect on the value returned by SSUB.

EXAMPLE

```
A = 2.3000000123456789

B = 5.0000000000000001

CRT SSUB (A,B)
```

Displays -2.6999999876543212 to the screen

STATUS Function

Use the STATUS function after an OPENPATH statement to find the cause of a file open failure (that is, for an tatement in which the ELSE clause is used). The following values can be returned if the statement is unsuccessful:

For File access commands

READ, WRITE, OPEN

Previous Operation

Value = 0 if successful

Value = Operating System error code if previous command failed

13 – permission denied on UNIX systems

OCONV Conversions

0 = successful

1 = invalid conversion requested

3 = conversion of possible invalid date

STATUS function

COMMAND SYNTAX

STATUS ( )

DESCRIPTION

Arguments are required for the STATUS function.

Values of STATUS after CLOSE, DELETE, MATREAD, MATWRITE, OPEN, READ and WRITE

After a DELETE statement: After a DELETE statement with an ON ERROR clause, the value returned is the error number.

Returns 0 if successful else returns ERROR number

STATUS function

After an OPEN, OPENPATH, or OPENSEQ statement: The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

After a READ statement: If the file is a distributed file, the STATUS function returns the following:

STATUS function

After a READL, READU, READVL, or READVU statement: If the statement includes the LOCKED clause, the returned value is the terminal number, as returned by the WHO command, of the user who set the lock.

After a READSEQ statement:

After a READT, REWIND, WEOF, or WRITET statement: The returned value is hardware-dependent (that is, it varies according to the characteristics of the specific tape drive unit). Consult the documentation that accompanied your tape drive unit for information about interpreting the values returned by the STATUS function.

STATUS statement

SYNTAX ELEMENTS

STATUS array FROM variable

```
THEN statements ELSE statements•ELSE statements
```

DESCRIPTION

Use the STATUS statement to determine the status of an open file. The STATUS statement returns the file status as a dynamic array and assigns it to an array.
The STATUS statement returns the following values in the following attributes:

STATUS Statement Values
Attribute Description

1 Current position in the file Offset in bytes from beginning of file
2 End of file reached 1 if EOF, 0 if not.
3 Error accessing file 1 if error, 0 if not.
4 Number of bytes available to read
5 File mode Permissions (in octal) 6 File size in bytes.
7 Number of hard links 0 if no links. Where applicable else 0
8 O/S User ID. ID based on the user name and domain of the user a jBASE pseudo user.
9 O/S Group ID.

10 I-node number; Unique ID of file on file system
11 Device on which i-node resides Number of device. The value is an internally calculated value on Windows NT.
12 Device for special character or block Number of device.
13 Time of last access in internal format
14 Date of last access in internal format.
15 Time of last modification in internal format
16 Date of last modification in internal format.
17 Time and date of last status change in internal format.
18 Date of last status change in internal format.
19 Number of bytes left in output queue (applicable to terminals only)
20 { }
21 jBASE File types j3, j4, jPLUS
22 jBASE File types j3, j4, jPLUS

23 jBASE File types j3, j4, jPLUS
24 Part numbers of part files belonging to a distributed file multivalued list

STATUS statement

variable specifies an open file. If variable evaluates to the null value, the STATUS statement fails and the program terminates with a run-time error message.
If the STATUS array is assigned to an array, the THEN statements are executed and the ELSE statements are ignored. If no THEN statements are present, program execution continues with the next statement. If the attempt to assign the array fails, the ELSE statements are executed; any THEN statements are ignored.

EXAMPLE

```
OPENSEQ '/Fred' TO test THEN PRINT "File Opened" ELSE STOP
STATUS info FROM filevar
filename= stat<20>
inode= info<10>
```

STOP

The STOP statement is virtually identical in function to the ABORT statement except that it does not terminate a calling jCL program.

STR

The STR function allows the duplication of a string a number of times.

COMMAND SYNTAX

STR (expression1, expression2)

SYNTAX ELEMENTS

expression1 will evaluate to the string to duplicate and may be of any length.

expression2 should evaluate to a numeric integer, which specifies the number of times the string will be duplicated.

EXAMPLE

LongString = STR ("long string ", 999)

STRS

Use the STRS function to produce a dynamic array containing the specified number of repetitions of each element of dynamic.array.

COMMAND SYNTAX

STRS (dynamic.array, repeat)

dynamic.array is an expression that evaluates to the strings to be generated.
repeat is an expression that evaluates to the number of times the elements are to be repeated. If it does not evaluate to a value that can be truncated to a positive integer, an empty string is returned for dynamic.array.
If dynamic.array evaluates to null, it returns null. If any element of dynamic.array is null, null is returned for that element. If repeat evaluates to null, the STRS function fails and the program enters the debugger.

EXAMPLE

```
ABC="A":@VM:"B":@VM:"C"
PRINT STRS (ABC,3)
```

```
The output of this program is:
```

```
AAA]BBB]CCC
```

SUBROUTINE

The SUBROUTINE statement is used at the start of any program that will be called externally by the CALL statement. It also declares any parameters to the compiler.

COMMAND SYNTAX

SUB{ROUTINE} Name {({MAT} variable{,{MAT} variable...})}

SYNTAX ELEMENTS

Name is the identifier by which the subroutine will be known to the compilation process. It should always be present as this name (not the source file name), will be used to call it by. However, if the name is left out, the compiler will name subroutine as the source file name (without suffixes). Default naming is not encouraged as it can cause problems if source files are renamed.
Each comma separated variable in the optional parenthesized list is used to identify parameters to the compiler. These variables will be assigned the values passed to the subroutine by a CALL statement.

NOTES

The SUBROUTINE statement must be the first code line in a subroutine.
A subroutine will inherit all the variables declared using the COMMON statement providing an equivalent COMMON area is declared within the SUBROUTINE source file. The program will fail to compile if the number of common variables used in each common area exceeds the number defined in the equivalent area in the main program.
Subroutines can only be called via the jBASE BASIC CALL statement
A subroutine can redefine PRECISION but the new precision will not persist when the subroutine returns to the calling program.
A subroutine will return to the CALLing program if it reaches the logical end of the program or a RETURN is executed with no outstanding GOSUB statement.
A SUBROUTINE will not return to the calling program if a STOP or ABORT statement is executed.
See also: CALL, CATALOG, COMMON, RETURN

EXAMPLES

```
SUBROUTINE DialUp(Number, MAT Results)
  DIM Results(8)
....
```

SUBS

The SUBS function returns a dynamic array, the content of which is derived by subtracting each element of the second dynamic array argument from the corresponding element of the first dynamic array argument.

COMMAND SYNTAX

SUBS(DynArr1, DynArr2)

SYNTAX ELEMENTS

DynArr1 and **DynArr2** represent dynamic arrays.

NOTES

Null elements of argument arrays are treated as zero. Otherwise, a non-numeric element in an argument array will cause a run-time error.

EXAMPLE

```
X = 1 : @VM : @VM : 5 : @VM : 8 : @SVM : 27 : @VM : 4
Y = 1 : @VM : 5 : @VM : 8 : @VM : 70: @VM : 19
S = SUBS(X, Y)
```

```
The variable S is assigned the value:
```

```
0 : @VM : -5 : @VM : -3 : @VM : -62 : @SVM : 27 : @VM : -15
```

SUBSTRINGS

The SUBSTRINGS function returns a dynamic array of elements, which are sub-strings of the corresponding elements in a supplied dynamic array.

COMMAND SYNTAX

SUBSTRINGS (DynArr, Start, Length)

SYNTAX ELEMENTS

DynArr should evaluate to a dynamic array.
Start specifies the position from which characters are extracted from each array element. It should evaluate to an integer greater than zero.
Length specifies the number of characters to extract from each dynamic array element. If the length specified exceeds the number of characters remaining in an array element then all characters from the Start position are extracted.

INTERNATIONAL MODE

When using the SUBSTRINGS function in International Mode, the function will use the 'start' and length' parameters to the function as character count values, rather than bytecount

EXAMPLES

The following program shows how each element of a dynamic array can be changed with the FIELDS function.

```
t = ""

t<1> = "AAAAA"

t<2> = "BBBBB" : @VM: "CCCCC" : @SVM: "DDDDD"

t<3> = "EEEEE":@VM:@SVM

r1 = SUBSTRINGS (t,3,2)

r2 = SUBSTRINGS(t,4,20)

r3 = SUBSTRINGS(t,0,1)
```

The above program creates 3 dynamic arrays. v represents a value mark. s represents a sub-value mark.

r1 <1>AA
<2>BB **v** CC **s** DD
<3>EE **v s**

r2 <1>AA
<2>BB **v** CC **s** DD

<3>EE **v s**

r3

<1>A

<2>B v C s D

<3>E v s

SUM

The SUM function sums numeric elements in a dynamic array.

COMMAND SYNTAX

SUM (expr)

SYNTAX ELEMENTS

expr is a dynamic array.

NOTES

Non-numeric sub-values, values and attributes are ignored.

EXAMPLES

```
s = CHAR (252)

v = CHAR(253)

a = CHAR(254)

a0 = 1:s:2:v:3:a:4:s:5:v:6:a:7:s:8:v:'NINE'

a1 = SUM (A)

a2 = SUM(a1)

a3 = SUM(a2)

CRT a0

CRT a1

CRT a2

CRT a3

The above code displays:

12²345²678²NINE
3²39²615²0
61515
36
```

SWAP

The SWAP function operates on a variable and replaces all occurrences of one string with another.

COMMAND SYNTAX

SWAP (variable, expression1, expression2)

SYNTAX ELEMENTS

expression1 may evaluate to any result and is the string of characters that will be replaced.
expression2 may also evaluate to any result and is the string of characters that will replace expression1. The variable may be any previously assigned variable in the program.

NOTES: Either string can be of any length and is not required to be the same length. This function is provided for compatibility with older systems.
See also: CHANGE function.

EXAMPLE

```
String1 = "Jim"

String2 = "James"

Variable = "Pick up the tab Jim"

CRT SWAP ( Variable, String1, String2)

CRT SWAP( Variable, "tab", "check")
```

System Functions

The following system functions are supported by jBASE:

SYSTEM(0)	Return the last error code
SYSTEM(1)	Return 1 if output directed to printer
SYSTEM(2)	Return page width
SYSTEM(3)	Return page depth
SYSTEM(4)	Return no of lines to print in current page. (HEADING statement)
SYSTEM(5)	Return current page number (HEADING statement)
SYSTEM(6)	Return current line number (HEADING statement)
SYSTEM(7)	Return terminal type
SYSTEM(8)	Return record length for tape channel 0
SYSTEM(9)	Return CPU milliseconds
SYSTEM(10)	Return 1 if stacked input available
SYSTEM(11)	Returns the number of items in an active select list or 0 if no list is active
SYSTEM(12)	Return 1/1000, (or 1/10 for ROS), seconds past midnight
SYSTEM(13)	Release time slice
SYSTEM(14)	Returns the number of characters available in input buffer. Invoking SYSTEM(14) can cause a slight delay in program execution.
SYSTEM(15)	Return bracket options used to invoke command
SYSTEM(16)	Return current PERFORM/EXECUTE level
SYSTEM(17)	Return stop code of child process
SYSTEM(18)	Return port number or JBCPORTNO
SYSTEM(19)	Return login name or JBASICLOGNAME. If the system_19_timedate emulation option is set then returns the number of seconds since midnight December 31, 1967.
SYSTEM(20)	Returns last spooler file number created
SYSTEM(21)	Returns port number or JBCPORTNO
SYSTEM(22)	Reserved
SYSTEM(23)	Returns status of the break key Enabled 0 Enabled 1 Disabled by BASIC 2 Disabled by Command 3 Disabled by Command and BASIC
SYSTEM(24)	Returns 1 if echo enabled, 0 if echo disabled
SYSTEM(25)	Returns 1 if background process
SYSTEM(26)	Returns current prompt character
SYSTEM(27)	Returns 1 if executed by PROC
SYSTEM(28)	Reserved.
SYSTEM(29)	Reserved.

SYSTEM(30)	Returns 1 if paging is in effect (HEADING statement)
SYSTEM(31)	Reserved
SYSTEM(32)	Reserved
SYSTEM(33)	Reserved
SYSTEM(34)	Reserved
SYSTEM(35)	Returns language in use as a name or number (ROS)
SYSTEM(36)	Reserved
SYSTEM(37)	Returns thousands separator
SYSTEM(38)	Returns decimal separator
SYSTEM(39)	Returns money symbol
SYSTEM(40)	Returns program name
SYSTEM(41)	Returns release number
SYSTEM(42)	Reserved
SYSTEM(43)	Returns port number of item lock
SYSTEM(44)	Returns 99 for jBASE system type
SYSTEM(45)	Reserved
SYSTEM(46)	Reserved
SYSTEM(47)	Returns 1 if currently in a transaction
SYSTEM(48)	Reserved
SYSTEM(49)	Returns PLID environment variable
SYSTEM(50)	Returns login user id
SYSTEM(51)	Reserved
SYSTEM(52)	Returns system node name
SYSTEM(53)	Reserved
SYSTEM(100)	Returns program create information
SYSTEM(101)	Returns port number or JBCPORTNO
SYSTEM(102)	Reserved
SYSTEM(1000)	Returns command line separated by attribute marks
SYSTEM(1001)	Returns command line and options
SYSTEM(1002)	Returns temporary scratch file name
SYSTEM(1003)	Returns terminfo Binary definitions
SYSTEM(1004)	Returns terminfo Integer definitions
SYSTEM(1005)	Returns terminfo String definitions
SYSTEM(1006)	Reserved
SYSTEM(1007)	Returns system time
SYSTEM(1008)	Returns SYSTEM file path
SYSTEM(1009)	Returns MD file path
SYSTEM(1010)	Returns Print Report information
SYSTEM(1011)	Returns jBASE release directory path. JBASICRELEASEDIR

SYSTEM(1012)	Returns jBASE global directory path. JBASICGLOBALDIR
SYSTEM(1013)	Returns memory usage (UNIX only): <1> Free memory small blocks <2> Free memory large blocks <3> Used memory small blocks <4> Used memory large blocks
SYSTEM(1014)	Returns relative PROC level
SYSTEM(1015)	Returns effective user name. LOGNAME
SYSTEM(1016)	Returns tape assignment information
SYSTEM(1017)	Returns platform. UNIX, WINNT or WIN95
SYSTEM(1018)	Returns configured processors
SYSTEM(1019)	Returns system information (uname -a)
SYSTEM(1020)	Returns login user name
SYSTEM(1021)	JBASE release information: <1> Major release number <2> Minor release number <3> Patch level <4> Copyright information
SYSTEM(1022)	Returns the status of jBASE profiling: 0 no profiling is active 1 full profiling is active 2 short profiling is active 3 jCOVER profiling is active
SYSTEM (1023)	Used by STATUS() function
SYSTEM(1024)	Retrieves details about last signals
SYSTEM(1025)	Returns value of International mode for thread
SYSTEM(1026)	Total amount of memory in use formatted with commas

SYSTEM(1027) Returns directory PROC; Used by WHERE, LISTU

Information about running processes can be obtained via the PROC jedi....

This JEDI enables retrieval of information from executing processes and is the interface now used by the WHERE command...

```
OPEN SYSTEM(1027) TO PROC ELSE STOP 201, "PROC"

SELECT PROC TO Sel

LOOP

WHILE READNEXT key FROM Sel DO

READ ProcessRecord FROM PROC, key ELSE CRT "Read Error"; STOP

REPEAT
```

Info for current user can be returned from the @USERSTATS variable.
Attribute descriptions for Process Records returned from the PROC Jedi READ interface.
<1> Port number
<2> Number of programs running
<3> Connect time
<4> Process ID
<5> Account name
<6> User name
<7> Terminal name in jBASE format
<8> Terminal name in UNIX format
<9> Database name
<10> Name of the tty device
<11> Language name
<12> Time listening thread executed
<13> Mallinfo memory free
<14> Mallinfo memory used
<15>Type of thread as a number
<16> Type of thread as a string WHERE

```
thread_type_string = "Normal" =1

thread_type_string = "javaOBjEX" = 2

thread_type_string = "vbOBjEX" = 3

thread_type_string = "jrfs" = 4

thread_type_string = "Compiler" = 5
```

<17> Number of instructions executed and licenses allocated to work around a bug in Windows. Need to build the buffer in separate sprintf's

<18> Number of OPEN's

<19> Number of READ's

<20> Number of WRITE's

<21> Number of DELETE's

<22> Number of CLEARFILE's

<23> Number of EXECUTE's

<24> Number of INPUT's

<25> UNUSED

<26> Number of files the application thinks is open

<27> Number of files that in reality are opened by the OS

<28> Application data set by @USER.ROOT

<29> Text String to identify process

<41> Command line arguments < threadnext >

<42> Current Line Number < threadnext >

<43> Name of source <threadnext >

<44> Status as a text string < threadnext >

status = "Program running normally"

status = "Program is SLEEPING"

status = "Program in DEBUGGER"

status = "Program at keyboard INPUT"

status = "Program blocked on record LOCK"

status = "Program performing EXECUTE/PERFORM"

status = "Error!! Status unknown"

<47> Status as an integer <threadnext >

<48> User CPU time <threadnext >

<49> System CPU time <threadnext >

<50> Child User CPU time <threadnext >

<51> Child System CPU time <threadnext >

<52> User defined thread data <threadnext >

SYSTEM(1028) Logged in database name

SYSTEM(1029) Shows the CALL stack history so that in error conditions the application, such as database I/O statistics, programs being performed and so on. Can be used with @USERDATA.

SYSTEM(1030) This new entry into the SYSTEM() function returns the current perform level in the range 1 to 32. This is similar to SYSTEM(16), which returns the nested execute level. The difference is that SYSTEM(16) does not include any procs, paragraphs or shells and returns the relative application program level. SYSTEM(1030) returns the relative program level including all the proc interpreters, paragraph interpreters and shells.

SYSTEM(1031) Number of free bytes on the current file system

SYSTEM(1032) Returns default frame size

SYSTEM(1034) Returns handle of the current thread

SYSTEM(1035) Returns the product ID of the license currently in use by this process;

1. Enterprise
13. Server

Entries above 2000 are for system use only.

TAN

The TAN function returns the mathematical tangent of an angle.

COMMAND SYNTAX

TAN (expression)

SYNTAX ELEMENTS

expression should evaluate to a numeric type.

NOTES

The function calculates the result at the highest precision available on the host system; it truncates the result to the current PRECISION after calculation.

EXAMPLES

```
Adjacent = 42

Angle = 34

CRT "Opposite length = ":TAN (Angle)*Adjacent
```

TIME

The TIME() function returns the current system time.

COMMAND SYNTAX

TIME ()

NOTES

Returns the time as the number of seconds past midnight

EXAMPLES

CRT "Time is ":OCONV(TIME(), "MTS")

TIMEDATE

The TIMEDATE() function returns the current time and date as a printable string.

COMMAND SYNTAX

TIMEDATE ()

NOTES

The function returns a string of the form: hh:mm:ss dd mmm yyyy or in the appropriate format for your international date setting.

EXAMPLES

CRT "The time and date is ":TIMEDATE ()

TIMEDIFF

Returns the interval between two timestamp values as a dynamic array

COMMAND SYNTAX

```
Time Diff(Timestamp1, Timestamp2,Mask)
```

SYNTAX ELEMENTS

The TIMEDIFF function returns the interval between two timestamp values by subtracting the value of Timestamp2 from Timestamp1. The interval is returned as an attribute delimited array of the time difference.

The Mask is an integer from 0 to 7 and selects one of the following output formats:

Mask		Array
0	-	Days^Hours^Minutes^Seconds^Milliseconds (Default)
1	-	Weeks^Days^Hours^Minutes^Seconds^Milliseconds
2	-	Months^Days^Hours^Minutes^Seconds^Milliseconds
3	-	Months^Weeks^Days^Hours^Minutes^Seconds^Milliseconds
4	-	Years^Days^Hours^Minutes^Seconds^Milliseconds
5	-	Years^Weeks^Days^Hours^Minutes^Seconds^Milliseconds
6	-	Years^Months^Days^Hours^Minutes^Seconds^Milliseconds
7	-	Years^Months^Weeks^Days^Hours^Minutes^Seconds^Milliseconds

TIMEOUT

If no data is read in the specified time, use the TIMEOUT statement to terminate a READSEQ or READBLK statement.

COMMAND SYNTAX

```
TIMEOUT file.variable, time
```

file.variable specifies a file opened for sequential access.

Time: is an expression that evaluates to the number of seconds the program should wait before terminating the
READSEQ statement.

TIMEOUT causes subsequent READSEQ and READBLK statements to terminate and execute ELSE statements if the number of seconds specified by time elapses while waiting for data.

If either file.variable or time evaluates to null, the TIMEOUT statement fails and the program enters the debugger.

EXAMPLES

```
TIMEOUT SLIPPERS, 10

READBLK VAR1 FROM SLIPPERS, 15 THEN PRINT VAR1 ELSE
PRINT "TIMEOUT OCCURRED"
END
```

TIMESTAMP

Returns a UTC timestamp value as decimal seconds

COMMAND SYNTAX

```
TIMESTAMP ()
```

SYNTAX ELEMENTS

The TIMESTAMP function returns a Universal Coordinated Time (UTC) value as decimal seconds, i.e. Seconds with tenths and hundredths specified after the decimal point.
"The value is returned as a variable with as many decimal places as the current precision allows. However, successive calls may return the same value many times before the operating system updates the underlying timer. For example, Windows updates the low level timer every 1/50 second even though it stores the time in billionths of a second."

TRANS

The TRANS function will return the data value of a field, given the name of the file, the record key, the field number, and an action code.

COMMAND SYNTAX

TRANS ([DICT] filename, key, field#, action.code)

SYNTAX ELEMENTS

DICT is the literal string to be placed before the file name in the event it is desired to open the dictionary portion of the file, rather than the data portion.
filename is a string containing the name of the file to be accessed. Note that it is the actual name of the file, and not a file unit variable. This function requires the file name, regardless of whether or not the file has been opened to a file unit variable.
key is an expression that evaluates to the record key, or item ID, of the record from which data is to be accessed.
field# is the field number to be retrieved from the record.
action.code indicates what should happen if the field is null, or the if record is not found. This is a literal. The valid codes are:

X	Returns a null string. This is the default action
V	Prints an error message.
C	Returns the value of key

NOTES

If the field being accessed is a dynamic array, TRANS will return the array with the delimiter characters lowered by 1. For example, multivalue marks (ASCII-253) are returned as subvalue marks (ASCII-252), and subvalue marks are returned as text marks (ASCII-251).
If you supply -1 for field#, the entire record will be returned.
The TRANS function is the same as the XLATE function.

EXAMPLES

Retrieval of a simple field: Given a file called "VENDORS" containing a record with the record key of "12345" and which contains the value of "ABC Company" in field 1,

```
VENDOR.ID = "12345"

VENDOR.NAME = TRANS ("VENDORS",VENDOR.ID,1,"X")

CRT VENDOR.NAME
```

will display: ABC Company

Retrieval of an array: Suppose field 6 of the VENDORS file contains a multivalued list of purchase order numbers, such as

10011]10062]10079
use the TRANS function to retrieve it:

```
PO.LIST = TRANS ("VENDORS",VENDOR.ID,6,"X")

CRT PO.LIST
```

will display: 10011\10062\10079
Notice that the backslashes (\) were substituted for brackets (]), indicating that the delimiter is now CHAR(252).
Retrieval of an entire dictionary item: Given a dictionary item called "VENDOR.NAME" with the following content

```
001 A

002 1

003 Vendor Name

004
005
006
007
008
009 L

010 30
```

these statements

```
DICT.ID = "VENDOR.NAME"

DICT.REC = TRANS ("DICT VENDORS",VENDOR.ID,-1,"C")

PRINT DICT.REC

will display

A]1]Vendor Name]]]]]L]30
```

TRANSABORT

The TRANSABORT statement is used to abort the current transaction and reverse any updates to the database.

COMMAND SYNTAX

TRANSABORT {abort-text} [THEN statement | ELSE statement]

SYNTAX ELEMENTS

abort-text specifies an optional text string to save in the transaction abort record.
A **THEN** or **ELSE** (or both) statement is required. The **THEN** clause will be executed if the transaction is successfully aborted. The **ELSE** clause will be executed if the transaction abort fails for any reason.

NOTES

Any record locks set during the transaction will be released upon successful completion.

TRANSQUERY

The TRANSQUERY function is used to detect whether or not a transaction is active on the current process.

COMMAND SYNTAX

TRANSQUERY()

NOTES

TRANSQUERY will return 1 (true) if the process is within a transaction boundary, and 0 (false) if it is not. In other words, TRANSQUERY will return true if the TRANSTART statement has been issued but a TRANSEND or TRANSABORT statement has not yet been processed.
By default, all hashed files are marked for inclusion in a transaction however this can be modified by the jchmod utility.

TRANSTART

In transaction processing, the TRANSTART statement is used to mark the beginning of a transaction.

COMMAND SYNTAX

TRANSTART {SYNC}{start-text} [THEN statement | ELSE statement]

SYNTAX ELEMENTS

SYNC is an option to force the updates to be flushed at transaction end or abort. start-text specifies an optional text string to save with the transaction start record.
A **THEN** or **ELSE** (or both) statement is required. The THEN clause will be executed if the transaction is successfully started. The ELSE clause will be executed if the transaction start fails for any reason.

NOTES

Record locks set during the transaction will not be released until a TRANSEND or TRANSABORT statement is processed.
A program (or series of programs) can only have one active transaction at one time. If another TRANSTART statement is encountered whilst a transaction is active, a run-time error will be generated.

TRANSEND

The TRANSEND statement is used to mark the end of a successfully completed transaction.

COMMAND SYNTAX

TRANSEND {end-text} [THEN statement | ELSE statement]

SYNTAX ELEMENTS

end-text specifies an optional text string to save with the transaction end record.
A **THEN** or **ELSE** (or both) statement is required. The **THEN** clause will be executed if the transaction is successfully ended. The **ELSE** clause will be executed if the transaction end fails for any reason.

NOTES

Any record locks set during the transaction will be released upon successful completion.

TRIM

The TRIM statement allows characters to be removed from a string in a number of ways.

COMMAND SYNTAX

TRIM (expression1 {, expression2{, expression3}})

SYNTAX ELEMENTS

expression1 specifies the string from which to trim characters.

expression2 may optionally specify the character to remove from the string. If not specified then the space character is assumed.

expression3 evaluates to a single character specifies the type of trim to perform.

NOTES

The trim types available for expression3 are:

Type	Operation
L	removes leading characters only
T	removes trailing characters only
B	removes leading and trailing characters
A	removes all occurrences of the character
R	removes leading, trailing and redundant characters
F	removes leading spaces and tabs
E	removes trailing spaces and tabs
D	removes leading, trailing and redundant spaces and tabs.

EXAMPLE

```
INPUT Answer

* Remove spaces and tabs (second parameter ignored)

Answer = TRIM (Answer, '', "D")

INPUT Joker

* Remove all dots

Thief = TRIM(Joker, ".", "A")
```

TRIMB

The TRIMB() function is equivalent to TRIM(expression, " ", "T")

TRIMBS

Use the TRIMBS function to remove all trailing spaces and tabs from each element of dynamic.array.

COMMAND SYNTAX

TRIMBS (dynamic.array)

TRIMBS removes all trailing spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If dynamic.array evaluates to null, null is returned. If any element of dynamic.array is null, null is returned for that value.

TRIMF

The TRIMF() function is equivalent to TRIM(expression, " ", "L")

TRIMFS

Use the TRIMFS function to remove all leading spaces and tabs from each element of dynamic.array.

COMMAND SYNTAX

TRIMFS (dynamic.array)

TRIMFS removes all leading spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If dynamic.array evaluates to null, it returns null. If any element of dynamic.array is null, it returns null for that value.

UNASSIGNED

The UNASSIGNED function allows a program to determine whether a variable has been assigned a value.

COMMAND SYNTAX

UNASSIGNED (variable)

SYNTAX ELEMENTS

variable is the name of variable used elsewhere in the program.

NOTES

The function returns Boolean TRUE if variable has not yet been assigned a value. The function returns Boolean FALSE if variable has already been assigned a value.
See also: ASSIGNED

EXAMPLES

```
IF UNASSIGNED(Var1) THEN

    Var1 = "Assigned now!"

END
```

UNIQUEKEY

Returns a unique 16-byte character key

COMMAND SYNTAX

UNIQUEKEY ()

SYNTAX ELEMENTS

The UNIQUEKEY() function will generate a unique 16-byte character key on each call to the function.
The key contains characters from the set A-Z a-z 0-9 + and / (base64)
Based on the current UTC time and the process number, the key is unique on a single computer system providing that the system clock is not turned back.
If the system administrator adjusts the system clock backwards, then there is a slight possibility of generating duplicate keys during the period until the clock has caught back up to time that the adjustment was made.
Any process that continues to execute throughout this period will continue to produce unique keys.
A process that starts up during this period and is given the process ID of a process that terminated during the period, may possibly generate a duplicate key until the period ends.

UNLOCK

The UNLOCK statement releases a previously LOCKed execution lock.

COMMAND SYNTAX

UNLOCK {expression}

SYNTAX ELEMENTS

If specifying **expression** it should evaluate to the number of a held execution lock, for release.

If omitting **expression** then it releases all execution locks held by the current program

NOTES

There is no action if the program attempts to release an execution lock that it had not taken.

See also: LOCK.

EXAMPLE

```
LOCK 23 ; LOCK 32

......
UNLOCK
```

UDTEXECUTE

See also:EXECUTE.

UPCASE

See also:DOWNCASE/UPCASE.

UTF8

The UTF8 function converts a latin1or binary string into the UTF-8 equivalent byte sequence.

COMMAND SYNTAX

UTF8 (expression)

SYNTAX ELEMENTS

The **expression** is expected to be a binary/latin1code page string, which converts the binary string into a **UTF-8** encoded byte sequence, used to represent the Unicode values for each byte in the expression.

NOTES

This function is useful for converting binary or latin1 code page data into internal format when in International Mode.

WAKE

Use the WAKE statement to wake a suspended process, which has executed a PAUSE statement.

COMMAND SYNTAX

WAKE PortNumber

SYNTAX ELEMENTS

PortNumber is a reference to awaken the target port. The WAKE statement has no effect on processes, which do not execute the PAUSE statement.

WEOF

The WEOF statement allows the program to write an EOF mark on an attached tape device.

COMMAND SYNTAX

WEOF {ON expression}

SYNTAX ELEMENTS

expression specifies the device channel to use. Should evaluate to a numeric integer argument in the range 0-9, the default value is zero.

NOTES

If the WEOF fails it then executes the statements associated with any ELSE clause. SYSTEM(0) will return the reason for the failure as follows:

1 there is no media attached to the channel

2 end of media found

NOTES

A "tape" does not refer to magnetic tape devices only but to any device described previously to jBASE. If the specified channel has no assigned tape device, it enters the jBASE debugger with an appropriate message.

EXAMPLE

```
WEOF ON 5 ELSE
    CRT "No tape device exists for channel 5"
END
```

WEOFSEQ

Write end of file on file opened for sequential access.

COMMAND SYNTAX

WEOFSEQ FileVar { THEN | ELSE Statements}

SYNTAX ELEMENTS

FileVar specifies the file descriptor of the file opened for sequential access.
Statements conditional jBASE BASIC statements

NOTES

WEOFSEQ forces truncation of the file at the current file pointer nothing is actually 'written' to the sequential file.

EXAMPLES

See also: Sequential File Examples

WRITE

The WRITE statement allows a program to write a record into a previously opened file.

COMMAND SYNTAX

WRITE variable1 ON|TO { variable2,} expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

variable1 is the identifier containing the record to write.

variable2, if specified, should be a previous opened jBASE BASIC variable to a file using the OPEN statement. If not specifying variable2 then it assumes the default file.

The expression should evaluate to a valid record key for the file.

If specifying the SETTING clause and the write fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

If holding a lock on the record by this process, it is released by the WRITE.

If you wish to retain a lock on a record, you should do so explicitly with the WRITEU statement.

EXAMPLE

```
OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"

END
WRITE Rec ON DCusts, "Xref" ON ERROR

CRT "Xref not written to DICT Customers"

END
```

WRITEBLK

Use the WRITEBLK statement to write a block of data to a file opened for sequential processing.

COMMAND SYNTAX

WRITEBLK expression ON file.variable
{THEN statements [ELSE statements] | ELSE statements}
Each WRITEBLK statement writes the value of expression starting at the current position in the file. The current position is incremented to beyond the last byte written. WRITEBLK does not add a new line at the end of the data.
file.variable specifies a file opened for sequential processing.
The value of expression is written to the file, and the THEN statements are executed. If no THEN statements are specified, program execution continues with the next statement. If the file is neither accessible or does not exist, it executes the ELSE statements; and ignores any THEN statements.
If either expression or file.variable evaluates to null, the WRITEBLK statement fails and the program enters the debugger with a run-time error message.

INTERNATIONAL MODE

When using the WRITEBLK statement in International Mode, care must be taken to ensure that the write variable is handled properly before the WRITEBLK statement. The WRITEBLK statement expects the output variable to be in “bytes”, however when manipulating variables in International Mode character length rather than byte lengths are usually used and hence possible confusion or program malfunction can occur. If requiring byte count data the output variable can be converted from the UTF-8 byte sequence to ‘binary/latin1’ via the LATIN1 function.

It is not recommended that you use the READBLK/WRITEBLK statements when executing in International Mode. You can obtain similar functionality via the READSEQ/WRITESEQ statement, which can be used to read/write, characters a line at a time from a file.

WRITELIST

WRITELIST allows the program to store a list held in a jBASE BASIC variable to the global list file.

COMMAND SYNTAX

WRITELIST variable ON|TO expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

variable is the variable in which the list is held.

expression should evaluate to the required list name. If expression is null, it writes the list to the default external list.

If the SETTING clause is specified and the write fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTE

See also: DELETELIST, READLIST, FORMLIST

EXAMPLE

```
* Create the list first
WRITELIST MyList ON "MyList"
```

WRITESEQ

Write to a file opened for sequential access.

COMMAND SYNTAX

WRITESEQ Expression {APPEND} ON|TO FileVar THEN | ELSE statements

or

WRITESEQF Expression {APPEND} TO FileVar THEN | ELSE statements

SYNTAX ELEMENTS

Variable specifies the variable to contain next record from sequential file.

FileVar specifies the file descriptor of the file opened for sequential access.

Statements conditional jBASE BASIC statements

NOTES

Each WRITESEQ writes the data on a line of the sequentially opened file. Each data is suffixed with a new line character. After each WRITESEQ, the file pointer moves forward to the end of line. The WRITESEQF statement forces each data line to be flushed to the file when it is written. The APPEND option forces each WRITESEQ to advance to the end of the file before writing the next data line.

EXAMPLES

See also: Sequential File Examples

WRITESEQF

SYNTAX

WRITESEQF expression {ON | TO} file.variable [ON ERROR statements]
{THEN statements [ELSE statements] | ELSE statements}

DESCRIPTION

Use the WRITESEQF statement to write new lines to a file opened for sequential processing, and to ensure that data is physically written to disk (that is, not buffered) before the next statement in the program is executed. The sequential file must be open, and the end-of-file marker must be reached before you can write to the file. You can use the FILEINFO function to determine the number of the line about to be written.

Normally, when you write a record using the WRITESEQ statement, the record is moved to a buffer that is periodically written to disk. If a system failure occurs, you could lose all the updated records in the buffer. The WRITESEQF statement forces the buffer contents to be written to disk; the program does not execute the statement following the WRITESEQF statement until the buffer is successfully written to disk.

A WRITESEQF statement following several WRITESEQ statements ensures that all buffered records are written to disk. WRITESEQF is intended for logging applications and should not be used for general programming. It increases the disk I/O of your program and therefore degrades performance.

file.variable specifies a file opened for sequential access.

The value of expression is written to the file as the next line, and the THEN statements are executed. If THEN statements are not specified, program execution continues with the next statement; if the specified file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If expression or file.variable evaluates to the null value, the WRITESEQF statement fails and the program terminates with a run-time error message.

The ON ERROR Clause

The ON ERROR clause is optional in the WRITESEQF statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the WRITESEQF statement is being processed.

WRITET

The WRITET statement enables data to be written to a range of tape devices between 0-9.

COMMAND SYNTAX

WRITET variable {ON|TO expression} THEN|ELSE statements

SYNTAX ELEMENTS

variable is the variable that holds the data for writing to the tape device.
expression should evaluate to an integer value in the range 0-9 and specifies from which tape channel to read the data. If the ON clause is not specified the WRITET will assume channel 0.
If the WRITET fails then the statements associated with any ELSE clause will be executed.
SYSTEM(0) will return the reason for the failure as follows:

1	there is no media attached to the channel
2	end of media found

NOTES

A "tape" does not refer to magnetic tape devices only but any device that has been described to jBASE. Writing device descriptors for jBASE is beyond the scope of this documentation.
If no tape device has been assigned to the specified channel the jBASE debugger is entered with an appropriate message.
Where possible the record size is not limited to a single tape block and the entire record will be written blocked to whatever block size has been allocated by the T-ATT command. However, certain devices do not allow jBASE to accomplish this (SCSI tape devices for instance).

EXAMPLE

```
LOOP
    WRITET TapeRec ON 5 ELSE

        Reason = SYSTEM(0)

        IF Reason = 2 THEN BREAK ;* done

        CRT "ERROR"; STOP

END
REPEAT
```

WRITEU

The WRITEU statement allows a program to write a record into a previously opened file. An existing record lock will be preserved.

COMMAND SYNTAX

WRITEU variable1 ON|TO { variable2,} expression {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

variable1 is the identifier holding the record to be written.
variable2, if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then the default file is assumed.
The expression should evaluate to a valid record key for the file.
If the SETTING clause is specified and the write fails, setvar will be set to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

If the statement fails to write the record then any statements associated with the ON ERROR clause is executed.

The lock maintained by the WRITEU statement will be released by any of the following events:
the same program with WRITE, WRITEV or MATWRITE statements writes to the record.

the record lock is released explicitly using the RELEASE statement.

the program stops normally or abnormally.

See also: READU, MATREADU, RELEASE

EXAMPLES

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"
```

```
END
WRITEU Rec FROM DCusts, "Xref" Setting Err ON ERROR

    CRT "I/O Error[":Err:"]"

    ABORT
END
```

WRITEV

The WRITEV statement allows a program to write a specific field of a record in a previously opened file.

COMMAND SYNTAX

WRITEV variable1 ON|TO {variable2,} expression1, expression2 {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

variable1 is the identifier holding the record to be written.
variable2, if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then it assumes the default file.
expression1 should evaluate to a valid record key for the file.
expression2 should evaluate to a positive integer number. If the number is greater than the number of fields in the record, it will add null fields to variable1. If expression2 evaluates to a non-numeric argument, it will generate a run time error.
If the SETTING clause is specified and the write fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

The WRITEV statement will cause the release of any lock held on the record by this program. If you wish to retain a lock on the record, do so explicitly with the WRITEVU statement.

EXAMPLE

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE

    ABORT 201, "DICT Customers"

END
WRITEV Rec ON DCusts, "Xref",7 Setting Err ON ERROR

    CRT "I/O Error[":Err:"]"
```

```
    ABORT
END
```

WRITEXML

WRITEXML rec ON file,id ELSE STOP 210,id

Write a dynamic array in xml format using a style sheet from the DICT

Use WRITEXML to write an XML record to a hash file

Transforms the XML into a dynamic array before being written to the file

The transform takes place using the style sheet in DICT->@WRITEXML

EXAMPLE

```
WRITEXML rec ON file,id ON ERROR CRT "Broken! " : rec
```

WRITEVU

The WRITEVU statement allows a program to write a specific field on a record in a previously opened file. An existing record lock will be preserved.

COMMAND SYNTAX

WRITEVU variable1 ON|TO { variable2,} expression1, expression2 {SETTING setvar} {ON ERROR statements}

SYNTAX ELEMENTS

variable1 is the identifier holding the record to be written.
variable2, if specified, should be a jBASE BASIC variable that has previously been opened to a file using the OPEN statement. If variable2 is not specified then the default file is assumed.
expression1 should evaluate to a valid record key for the file.
expression2 should evaluate to a positive integer number; if the number is greater than the number of fields in the record, null fields will be added to variable1. If expression2 evaluates to a non-numeric argument, a run time error will be generated.
If the SETTING clause is specified and the write fails, it sets setvar to one of the following values:

INCREMENTAL FILE ERRORS

128	No such file or directory
4096	Network error
24576	Permission denied
32768	Physical I/O error or unknown error

NOTES

If the statement fails to write the record, it executes any statements associated with the ON ERROR clause.
Any of the following events will release the lock taken by the WRITEVU statement:

The same program with WRITE, WRITEV or MATWRITE statements writes to the record.
By explicitly using the RELEASE statement, it releases the record lock.
The program stops normally or abnormally.
See also: MATWRITEU, RELEASE, WRITE, WRITEU.

EXAMPLE

```
OPEN "Customers" ELSE ABORT 201, "Customers"

OPEN "DICT Customers" TO DCusts ELSE
```

```
    ABORT 201, "DICT Customers"

END
WRITEVU Rec ON DCusts, "Xref",1 SETTING Err ON ERROR

    CRT "I/O Error[":Err:"]

ABORT
END
```

XLATE

The XLATE function will return the data value of a field, given the name of the file, the record key, the field number, and an action code.

COMMAND SYNTAX

XLATE ([DICT] filename, key, field#, action.code)

SYNTAX ELEMENTS

DICT is the literal string to be placed before the file name in the event it is desired to open the dictionary portion of the file, rather than the data portion.
filename is a string containing the name of the file to be accessed. Note that it is the actual name of the file, and not a file unit variable. This function requires the file name, regardless of whether or not the file has been opened to a file unit variable.
key is an expression that evaluates to the record key, or item ID, of the record from which data is to be accessed.
field# is the field number to be retrieved from the record.
action.code indicates the procedure if the field is null, or cannot find the if record. This is a literal. The valid codes are:

X Returns a null string. This is the default action
V Prints an error message.
C Returns the value of key

NOTES

If the field being accessed is a dynamic array, XLATE will return the array with the delimiter characters lowered by 1. For example, multivalue marks (ASCII-253) are returned as subvalue marks (ASCII-252), and subvalue marks are returned as text marks (ASCII-251).
If you supply -1 for field#, it returns the entire record.
The XLATE function is the same as the TRANS function.

EXAMPLE

1. Retrieval of a simple field: Given a file called "VENDORS" containing a record with the record key of "12345" and which contains the value of "ABC Company" in field 1,

```
VENDOR.ID = "12345"

VENDOR.NAME = XLATE("VENDORS",VENDOR.ID,1,"X")

CRT VENDOR.NAME
```

will display: ABC Company
2. Retrieval of an array: Suppose field 6 of the VENDORS file contains a multivalued list of purchase order numbers, such as

10011]10062]10079
use the XLATE function to retrieve it:

```
PO.LIST = XLATE("VENDORS",VENDOR.ID,6,"X")

CRT PO.LIST
```

will display: 10011\10062\10079
Notice that the backslashes (\) were substituted for brackets (]), indicating that the delimiter is now CHAR(252).
3. Retrieval of an entire dictionary item: Given a dictionary item called "VENDOR.NAME" with the following content

```
001 A

002 1

003 Vendor Name

004
005
006
007
008
009 L


010 30
```

these statements

```
DICT.ID = "VENDOR.NAME"
DICT.REC = XLATE("DICT VENDORS",VENDOR.ID,-1,"C")


PRINT DICT.REC
```

will display
A]1]Vendor Name]]]]]L]30

XMLTODYN

COMMAND SYNTAX

```
XMLTODYN(XML,XSL,result)
```

SYNTAX ELEMENTS

Converts the XML to a dynamic array using the optional XSL to transform

Array = XMLTODYN(XML,XSL,result)

If result = 0 Array will contain a dynamic array built from the xml / xsl

If result <> 0, Array will contain an error message

There is no requirement for xsl if you are reconverting from generic xml to dynarray

```
a = "Tom" : @AM : "Dick" : @AM : "Harry"
      xml = DYNTOXML(a,"",result)
      b = XMLTODYN(xml,"",result
      CRT CHANGE(b      ,@AM," ")
```

SCREEN OUTPUT

```
Tom Dick Harry
```

If passing a stylesheet in the second parameter, it performs a transform to give a different format of the array.

XML CONTENTS

```
<?xml version="1.0" encoding="UTF-8"?>

<mycustomer>
  <firstname>Tom</firstname>
  <lastname>Dick</lastname>
  <address>Harry</address>
</mycustomer>
```

EXAMPLE

```
a = XMLTODYN(xml,xsl,rc)
   CRT CHANGE(a,@AM," ")
```

XSL CONTENTS

```
<xsl:template match="mycustomer">
<array>
<xsl:apply-templates/>
```

```
</array>
</xsl:template>

<xsl:template match="firstname">
<data>
<xsl:attribute name="attribute">1</xsl:attribute>
<xsl:attribute name="value">
      <xsl:number level="single"/>
</xsl:attribute>
<xsl:attribute name="subvalue">1</xsl:attribute>
<xsl:value-of select="."/>
</data>
</xsl:template>

Etc
```

XMLTOXML

COMMAND SYNTAX

```
XMLTOXML(xml,xsl,result
```

SYNTAX ELEMENTS

Transform the XML using the XSL

If result=0, newxml will contain a transformed version of xml using xsl

If result=1, newxml will hold an error message

XSL CONTENTS

```
<?xml version="1.0" ?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="person">
<p><xsl:value-of select="name" /></p>
</xsl:template>

</xsl:stylesheet>
```

XML CONTENTS

```
<list>
<person>
      <name>Bob</name>
</person>
<person>
      <name>Amy</name>
</person>
<list>
```

EXAMPLE

```
newxml = XMLTOXML(xml,xsl,rc)
   CRT newxml
```

SCREEN OUTPUT

```
<p>Bob</p><p>Amy</p>
```

XTD

The XTD() function converts hexadecimal numbers into its decimal equivalent.

COMMAND SYNTAX

XTD(expression)

SYNTAX ELEMENTS

expression should evaluate to a valid hexadecimal string.

NOTES

The conversion process will halt at the first character that is not a valid base 16 character in the set [0-9, A-F or a-f].

See also: DTX.

EXAMPLES

```
A = "FF"

CRT XTD(A)
```

Embedded SQL for jBASE BASIC

The name "SQL" is an abbreviation for "Structured Query Language". The SQL language enables the defining, manipulating and controlling of data in a relational database. A relational database is a database that appears to the user as a collection of tables. A table is defined to be an unordered collection of rows. Finally the SQL terminology tends to refer to records as rows and fields within a record as a columns within a row.

Embedded SQL is a version of SQL designed for direct incorporation into hosts programs or specifically in the case of jBASE, into jBASE BASIC programs.

An Embedded SQL jBASE BASIC program contains normal jBASE BASIC code statements plus an Embedded SQL declare section, zero or more embedded cursor definitions, zero or more embedded exception declarations and one or more Embedded SQL statements.

Embedded SQL declarations, definitions and statements are prefixed by the reserved words **EXEC SQL**. This part of the Embedded SQL standard also enables the jBASE BASIC preprocessor to recognize and distinguish SQL statements from the normal jBASE BASIC code statements. The Embedded SQL statements are terminated by a semicolon.

Embedded SQL statements can include references to jBASE BASIC variables. The jBASE BASIC variables must be prefixed with a colon to distinguish them from SQL column names. The jBASE BASIC variables cannot be qualified or subscripted and must refer to scalars, i.e. character strings or numbers, not arrays or expressions.

All jBASE BASIC variables that will be referenced in Embedded SQL statements must be defined within an Embedded SQL declare section, the jBASE BASIC variable definitions are limited to simple forms. i.e. no expressions or arrays.

An Embedded SQL cursor must not appear in an Embedded SQL statement before it has been defined by an Embedded SQL cursor definition.

Any jBASE BASIC variables that will be referenced in Embedded SQL statements must have a data type that is compatible with the SQL data type of the column with which they are to be compared or assigned. However this requirement does not prevent jBASE BASIC variables from using the same name as Embedded SQL column references.

Embedded SQL statement exceptions can be handled either by utilizing the SYSTEM(0) function or predetermined by the SQL WHENEVER statement.

The following jBASE BASIC code provides an example of using Embedded SQL for Oracle.

PartEntry.b listing (Oracle)

```
*
* Declare jBASE BASIC vars to use in Embedded SQL statements ( A )
*

EXEC SQL BEGIN DECLARE SECTION;
INT PartNo;
STRING(20) PartName;
STRING(16) User;
```

```
STRING(16) Passwd;
EXEC SQL END DECLARE SECTION;
*
* Predetermine action on SQLERROR ( B )
*
EXEC SQL WHENEVER SQLERROR DO SQL_ERROR() ;
*
* Connect to database supplying user and password ( C )
*
User = "demo" ; Passwd = "demo99"
EXEC SQL CONNECT :User IDENTIFIED BY :Passwd;
*
* Create Parts table ( D )
*
EXEC SQL CREATE TABLE Parts
(
PartNo INTEGER NOT NULL PRIMARY KEY,
PartName CHAR(20)
);
*
* Loop until no more PartNos
*
LOOP
*
* Prompt for PartNo
*
    CRT "Part Number :":
    INPUT PartNo
WHILE PartNo NE '' DO

*
* Prompt for PartName
*
    CRT "Part Name :":
    INPUT PartName
*
* Add PartNo and PartName into Parts table ( E )
*
    EXEC SQL INSERT INTO Parts VALUES (:PartNo, :PartName );
REPEAT

*
* Commit updates to database ( F )
*
```

```
EXEC SQL COMMIT ;
```

(A) Declare jBASE BASIC variables to use within Embedded SQL statements

This section declares jBASE BASIC variables so that they can be used within Embedded SQL statements. All references to jBASE BASIC within the Embedded SQL statement must be prefixed by a colon. This feature of the Embedded SQL standard is used by the jBASE BASIC preprocessor to identify jBASE BASIC variables when parsing the Embedded SQL statement. The jBASE BASIC variables must be the same data type as the source or target Embedded SQL columns.

(B) Predetermine action on SQLERROR

This section configures the action to take on detecting an error with the previous executed Embedded SQL statement. Every SQL statement should in principle be followed by a test of the returned SQLCODE value. This can be achieved by utilizing the SYSTEM(0) function, which returns the result of the last SQL statement, or alternatively using the Embedded SQL WHENEVER statement to predetermine the action for all subsequent Embedded SQL statements. The SYSTEM(0) function will return three different possible values.

< 0	Embedded SQL statement failed.
0	Embedded SQL statement successful.
100	NOT FOUND. No rows where found.

The format of the Embedded SQL WHENEVER statement is as follows:

EXEC SQL WHENEVER Condition Action ;

where

Condition	NOT FOUND SQLERROR
Action	DO Function - Oracle implementation. CALL Function - Ingres and Informix implementations. GOTO proglab_Label – IBM DB2 and Microsoft SQL Server implementations. CONTINUE
Function	User defined function. SQLERROR() - Display Embedded SQL error then return to program. SQLABORT() - Display Embedded SQL error then exit program.

Label	Label in executing program:
	`DOSQLERR:`
	`  DEFC INT SQL_ERROR`
	`  CALL SQL_ERROR`
	`  STOP`

(C) Connect to database supplying user and password

This section connects the specified user and or passwd combination to the SQL database. This

command can be Embedded SQL implementation dependent. The user must be correctly configured for the target database.

(D) Create Parts table.
This section creates an SQL table called Parts. The table has two constituent data types, these are defined as an integer value PartNo and a character string PartName. The PartNo is defined as a non null unique value and is defined as the primary key. This definition provides a close match to the usual format of a record and id. The only data type that is truly common to all hosts and their languages is fixed length character strings, the integer value used here is for demonstration purposes and is not recommended.

(E) Add PartNo and PartName into table Parts.
This Embedded SQL statement inserts the values entered for PartNo and PartName into the SQL table Parts. PartNo is inserted as the first column whereas PartName is inserted as the second column of each row. Effectively PartNo is the record id and PartName is the first field in the record PartNo. The jBASE BASIC pre-processor parses the Embedded SQL statements and provides code to convert any specified jBASE BASIC variables to the format required by the Embedded SQL implementation. Any returned parameters are then converted back into jBASE BASIC variables.

(F) Commit updates to database.
This Embedded SQL statement makes all updates by Embedded SQL statements since the last SQL commit statement visible to other users or programs on the database. If a program executes an Embedded SQL statement and no transaction is currently active then one is automatically started. Each subsequent SQL statement update by the same program without an intervening commit or rollback, is considered part of the same transaction. A transaction terminates by either an Embedded SQL COMMIT, normal termination, or an Embedded SQL ROLLBACK statement, abnormal termination. An abnormal termination does not change the database with respect to any of the Embedded SQL updates executed since the last commit or rollback. Database updates made by a given transaction do not become visible to any other distinct transaction until and unless the given transaction completes with a normal termination. i.e. an Embedded SQL COMMIT statement.

EMBEDDED SQL COMPILER OPTION

In order to compile jBASE BASIC programs containing Embedded SQL statements the jBASE compiler option "Jq" must be invoked with the jBASE BASIC compiler command. The "Jq" option also expects an SQL implementation specifier, as described below.

-Jq<type>	**RDBMS**
d	IBM DB2
m	Microsoft SQL Server (Windows only)
o	Oracle

i	Ingres
s	Sybase
x	Informix

e.g. To compile the jBASE BASIC example program PartEntry.b for an Oracle SQL implementation database.

```
jcompile -Jqo PartEntry.b
```

In this example the SQL specifier is "o" for Oracle. Other specifiers are added as and when Embedded SQL implementations are required. e.g. The "i" option informs the jBASE compiler to invoke mechanisms for the Ingres Embedded SQL implementation. Although the Embedded SQL standard is the same, each SQL provider requires different manipulative techniques in order to compile and connect to the database.

The jcompile compiler pre-processes the jBASE BASIC program parsing the normal jBASE BASIC and Embedded SQL statements to produce an intermediate C program. The SQL implementation dependent pre-processor is then invoked to convert the Embedded SQL statements to the implementation defined internal functions. The resulting program is then is then compiled and linked.

The jBASE BASIC compilation should be executed in a user account which has been enabled for the required Embedded SQL implementation. Attempting to compile in an account not enabled for the required SQL implementation may cause compilation failure as certain environment variables for the implementation may not have been modified for the correct directory paths, etc.

TROUBLESHOOTING

When attempting to compile a program with Embedded SQL and you get an error along the lines of...

```
Command failed: nsqlprep PartEntry.sqc
SQL Pre Processor error -1
```

...this is an indication that either you have not loaded the Embedded SQL Kit and do not have the 'nsqlprep' command, or the 'nsqlprep' command does exist but it is not visible to the PATH environment variable.

Comment Sheet

Please give page number and description for any errors found:

Page	Error

Please use the box below to describe any material you think is missing; describe any material, which is not easily understood; enter any suggestions for improvement; provide any specific examples of how you use your system, which you think, would be useful to readers of this manual. Continue on a separate sheet if necessary.

Copy and paste this page to a word document and include your name address and telephone number and send to: support@jbase.com