jBASE

Programmer's Reference Guide

Copyright Notice

Copyright (c) 2002 TEMENOS HOLDINGS NV. All rights reserved.

This document contains proprietary information that is protected by copyright. No part of this document may be reproduced, transmitted, or made available directly or indirectly to a third party without the express written agreement of jBASE Software. Receipt of this material directly from jBASE Software or TEMENOS HOLDINGS NV constitutes its express permission to copy. Permission to use or copy this document expressly excludes modifying it for any purpose, or using it to create a derivative therefrom.

Acknowledgements

jBASE, jBC, jED, jSHELL, jLP, jEDI, jCL, jQL, j1, j2, j3 and j4 are trademarks of jBASE Software, Inc.

UNIX® is a registered trademark of X/Open Co. Ltd.

REALITY is a trademark of MDIS plc.

PICK is a trademark of Raining Data.

All other trademarks are acknowledged.

Summary of Chapters

Introduction	Gives an overview of jBASE.
jBC	Details the functionality and benefits derived from working with jBC. Describes the file and directory organization and outlines the library of functions available to the application programmer. Also offers some advice to the jBC programmer that should help you to produce efficient code.
jCL	Describes the components of the jCL language and lists the syntax and usage for each command.
jQL	Describes the components of the jQL language and lists the syntax and usage for each command.
List Processing	Describes the list processing capabilities of jBASE and lists the syntax and usage for each command.
jBASE Editors	Describes the features and functionality of the jED and ED editors. Particularly useful for programmers.

Table of Contents

Introduction to jBASE

jBASE offers a suite of database management products and development tools. jBASE, the flagship product, provides a multidimensional database, a development environment including a development language, and a middleware component allowing other mainstream and standards-based products to communicate with the jBASE products. The unique jEDI middleware enables access to other databases such as Oracle, DB2 and SQL Server. Microsoft Windows, all major Unix platforms including Linux, and IBM's e-Servers are supported.

Product Overview

At the highest level, jBASE can be considered to consist of three components, each of which may be used separately or together to solve application development issues.

Application Development Environment

The first component is the Application Development Environment. This consists of several products. First and most importantly is the Basic language (jBC). This is the traditional language for developing applications in jBASE. The language, while loosely based on Dartmouth Basic, has been greatly enhanced and includes syntax that deals efficiently and powerfully with the benefits of the multidimensional database architecture. In addition to jBC, we also offer several interface products, which provide similar capabilities to jBC but can be used from different development environments. These products, known generically as jBASE OBjEX, provide an interface from two environments, Microsoft's COM and Java. The COM based OBjEX allows access to the jBASE database and application development environment from such products as Visual Basic and Delphi as well as products that support Visual Basic for Applications (VBA) including members of the Microsoft Office Suite. jBASE OBjEX provides a similar capability for Java based development tools, such as JBuilder, Visual Age for Java, Cold Fusion, etc. Using these tools a developer can take advantage of all the functionality of the jBASE database management system as well as the additional benefits of the jBC language. Modules written in jBC, perhaps existing business rules, can be called from Java and COM based applications.

Middleware

The second component is Middleware. This component takes software developed in an ADE, (not necessarily jBC), and provides the ability to access a database, which may not necessarily be the jBASE database. The most important part of the jBASE middleware component is the jBASE External Device Interface (jEDI). The jEDI is a layer of software that is used by all Input and Output operations within the jBASE system. No matter what method an application uses to access the jBASE I/O mechanism the code used is identical. The jEDI itself can be thought of as having two parts. The first is the interface between the application and the jEDI. Every time an application accesses data from the jEDI interface, whether using Java, jBC, ODBC or directly from a 'C' program, the program will end up calling the same functions. This interface converts the I/O operation into what is known as a canonical form. The second part of the jEDI is the interface from the canonical form to the database itself. This interface, again a formal interface, can be thought of as a series of database drivers.

The jBASE middleware component provides the ability to access any type of data source using this mechanism. When the application requests data from the data source, the jEDI turns that request from a language dependent form into the canonical form and then into a form the data source can understand. These data sources are usually databases, but can be of different types. If there is a well-defined method of getting data out of an environment, a jEDI can be written to access the data and provide the interface to jBASE.

The standard jBASE product has jEDIs provided for the jBASE database. There are several types of jBASE files, j1 through j5, (j5 to be in a future release) and each of them has their own jEDI. Also included are jEDIs for the operating system files (e.g. directories and text files).

To further understand the power of the jEDI architecture, let's examine a real software application, GLOBUS, the world's premier Banking Software, from our parent company, TEMENOS. This product currently runs on jBASE and uses the standard jBASE database. The plan is to also offer GLOBUS on DB2 and Oracle. This will be accomplished by writing a jEDI for each of these two databases. The jEDI will translate the GLOBUS request for data, or the request to write some data, into the comparable operation for DB2 or Oracle.

This means that no matter where the data resides, jBASE, DB2 or Oracle, the GLOBUS application code will remain exactly the same, obviously benefiting application development schedules, maintainability and flexibility.

Currently, the GLOBUS application takes advantage of the multivalue capabilities of the jBASE database (a Non First Normal Form database). This functionality is not supported by flat databases such as DB2 and Oracle, therefore the jEDI must also convert the sophisticated multidimensional architecture of the jBASE database into the multiple joined tables of the flat relational model.

The question that everyone has about this architecture is how it will affect performance. The answer is that jEDI architecture does not. This is not to say that GLOBUS will perform the same using the jBASE database versus using DB2. This is not the case; the performance characteristics will be completely different for each scenario. However, assuming that a jEDI is written efficiently, the source code is simply compiled into an application calling the native database access methods, in exactly the same way as it would be done by any other Application Development Environment.

The jEDI architecture also provides several other benefits. The first is that application specific logic can be added to the jEDI. A very good example of this is the sophisticated security model of GLOBUS. This security model prevents unauthorized access to sensitive banking operations and data. This is currently done at the application level, so any access to the GLOBUS database must go through the application. This is not a ideal situation since it would be much more efficient and useful if clients could access GLOBUS data directly from analysis products such as Microsoft Excel or reporting tools such as Crystal Reports without having to go through intermediary steps. If the security model were implemented within the jEDI, all accesses would implicitly use this security model and could be accessed using whatever standard tools are required. (This functionality is planned for a future release of GLOBUS).

The jEDI interface proves beneficial in the way it is used by other products in the jBASE product suite. A good example is Transaction Journaling. Transaction Journaling records every time the database is updated and can be used to maintain a hot standby system. This is accomplished by passing the journal entry to another process, which applies that journal entry to another separate database. These journal entries are created at the jEDI level and are therefore database independent. This means that jBASE can mirror different types of databases, ensuring that a jBASE, Oracle or DB2 system are replicated quickly and efficiently using data written to any one of them from the GLOBUS application.

Another part of the jBASE middleware component is the jBASE Data Provider (jDP). This provides a standards based interface to the jBASE database (or more correctly to the jEDI). Any products adhering to the ODBC, OLE DB, JDBC or ADO standards can retrieve data from the jEDI interface and of course, therefore, from the jBASE database.

Database Management System

The third and final component of jBASE is the Database Management System. It is a Non First Normal Form database. This means tables can be embedded within other tables. This removes the necessity of having to perform the extremely resource hungry joins of the traditional Relational model.

jBASE provides very efficient support for multi-user applications. It has a locking mechanism that prevents users from overwriting data that is being updated by another user. It provides transaction management facilities where the application can be split into units of work, transactions, that group database activities together. An example in the banking world would be grouping a debit and credit together, ensuring that either both updates are done or neither.

Physically j1 through j4 jBASE files are hashed files. This is a very efficient way of ensuring a uniform distribution of data throughout the database, providing very fast access to all data held by jBASE. This method, however, needs a significant of administration and maintenance in a very volatile environment, where data is being added, removed or changed significantly over short periods of time. To improve on this potential inefficiency, we are introducing the j5 file system that will be a hybrid journaled and logged file system. This file system will provide self-sizing and enhanced resilience, as well as being a 64-bit file system that removes any file size limit.

Summary

jBASE is therefore really three separate parts that can be used independently but when used together provide a superb technological platform on which to base future application development. With its openness, flexibility and power, the jBASE product meets or surpasses the requirements for world-class business applications.

Hardware Considerations

jBASE is supplied for many different machine architectures, operating systems and distribution media. Before installing any jBASE product, it is important to check that the jBASE product matches the machine architecture, operating system and distribution media relevant to your requirements. You can verify these by examining the label on the distribution media and the Release Advisory document supplied with it.

jBC

- jBC is an operating system-resident programming language supported by the jBASE Database Independent Management Engine.
- jBC can access database files of any operating system-resident, Open Systems database.
- jBC is aimed primarily at writing business applications, and contains all the constructs needed to access and modify files and their data efficiently.
- jBC is a sophisticated superset of Dartmouth BASIC supporting structured programming techniques.
- jBC is a flexible and user extendible language.
- jBC contains the functionality needed to write efficient applications.
- jBC programs can call external functions written in C or jBC. C programs can be made to call functions written in jBC.
- jBC programs can mixed with embedded SQL statements allowing queries and updates on any SQL Database.
- jBC 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.
- jBC allows the application programmer working in an operating system 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'.
- jBC has other advantages over C such as the in-built debugger and easy file i/o.
- jBC programs may declare external functions, which are linked into the application by the operating system linker-loader. This means that jBC offers access to specialized functions written in C or any language that is link compatible with C.

Features of jBC

- 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 jBC 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 jBC

- Applications are running on an Open Systems platform;
- Applications are very efficient as the execution speed of jBC code is close to that of hand crafted 'C';
- Applications are portable between any binary compatible operating system environment.
- Applications integrate easily with other systems;
- Applications benefit from the steady improvements made in compiler optimization.
- Use of jBC offers tremendous productivity improvements over 'C';
- The close compatibility with UNIX allows the jBC developer to produce libraries of standard subroutines or function calls which can be used by any program;
- The standard UNIX command set is available to manage code libraries;
- Database access is provided 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;
- jBC 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 BASIC applications and development and programming skills in BASIC are fully retained;
- No need for costly retraining of programmers to 'C'. However, 'C' can also be freely used within the application system, thus allowing more flexibility;
- jBC provides connection to external devices and external databases in a manner that is transparent to existing applications;

jBC Environment

- jBC will run on any standard Windows or Unix system and with any standard shell or editor. An easy to use jSHELL is also provided.
- jBC programs can be written using any available editor. A context sensitive screen editor (jED) is provided that is designed specifically for jBC 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.
- A file or directory may be specified to hold all the jBC source. The finished executables may be held in a different file or directory if required.
- A global user library may be used to hold globally accessible user routines.

jBC Programming

- jBC source code may be written using any system editor, including the jED editor.
- The jBC compiler can be used to produce intermediate object code or executable files.
- Makefiles can be used to simplify the compilation process, especially if many files are involved. Their use will also make upgrading and software maintenance an easier task.
- 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/jbc/lib directory (usr\jbc\lib\ in Windows).

jBC Comparisons

jBC versus BASIC

- Both jBC and BASIC languages are derived from Dartmouth Basic.
- jBC is an enhanced variant of BASIC. It contains all the commands and constructs necessary for compatibility with other versions of BASIC. It also provides full interaction with the UNIX or Windows system and database files.
- jBC can be quickly modified to retain compatibility with any future enhancements to the BASIC language or its derivatives.
- The jBC compiler will produce code that will run many times faster than the same BASIC code compiled and run on any other RDBMS environment.
- jBC can access jBASE, CISAM, UNIX, and Windows files as well as records and files of other databases.
- jBC debug facilities are greatly superior to those provided with other versions of BASIC.

jBC versus 'C'

- The jBC compiler uses many features of the visual c++ compiler.
- The jBC compiler can compile 'C' source and object files, as well as jBC source code.
- Source compilation can be halted at any stage, and the resultant code examined.
- External 'C', and jBASE library access is available.
- jBC has a sophisticated debugger available as standard.
- jBC is able to provide full and easy access to UNIX , Windows or any third party database files.
- jBC has the tools to provide sophisticated string handling.
- jBC handles system signals and events automatically.

File and Directory Organization

Several directories and files have been set up which ensure the smooth and efficient use of the jBC programming environment.

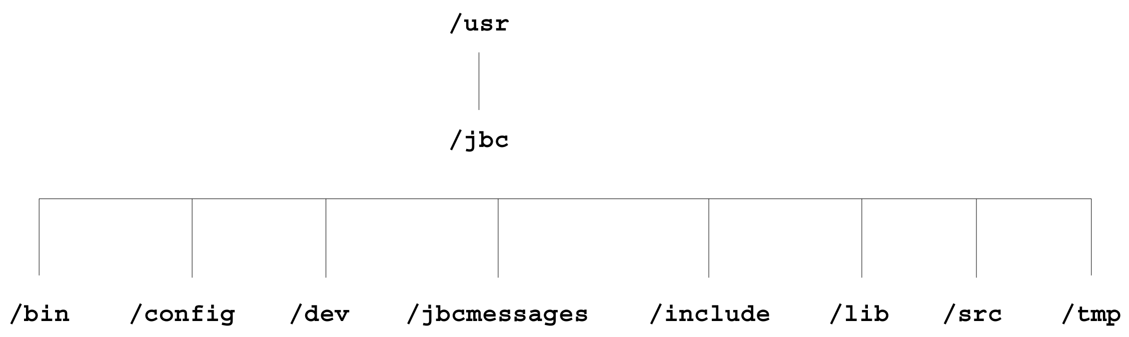
All the jBC files are held under the UNIX /usr/jbc directory (or the Windows \usr\jbc directory).

The main body of the jBC program and library files are held in the /usr/jbc directory. This contains all the run-time code, error and library files, as well as default system and terminal set-up limit.

jBC include files are held in the general UNIX directory for these, the /usr/jbc/include directory.

The jBC programming environment is organised as sets of executable programs, libraries and look-up files allowing the programmer to compile, run and debug jBC applications.

By default, the jBC libraries are held under the /usr directory - as shown below:



/bin Contains all the binary executables that make up the jBASE system including the compilers. On a runtime only system various components (such as the compilers) are missing from here.

/config This file contains various configuration files for systems such as background processing and the j-file(s)lock daemon.

/dev All external devices on the current system are described here.

jbcmessages This file contains all jBC error messages and may be edited by the user.

/include Various include files required at compile time are stored here. This directory is not present on a runtime only system.

/lib This directory contains all the libraries required for linking executing jBC compiled programs. Shared or Dynamic libraries are also stored here.

/src This directory contains any source code files that are distributed with the current jBASE release. It contains extended user exit support, templates for Makefiles and jEDI interface code.

/tmp This is a general purpose temporary directory for runtime use.

jBC Libraries

Shared Libraries

Shared libraries contain code for jBC or C functions that is not linked (joined) to the users jBC program until runtime. A single system will only ever load this library into memory once and any executing jBC programs will share it between them.

SVR3.2 Libraries

Under SVR3.2 shared libraries must be constructed with extreme care with all function addresses defined to the system before their compilation. All jBC libraries on this version of UNIX have been defined as shared (unless the particular flavour of UNIX does not support shared libraries). However libraries produced by the programmer to store callable jBC subroutines cannot be defined as shared and each program calling a subroutine will contain a copy of the subroutine code. This can make for large executable programs on such systems.

SVR4 Libraries

Under SVR4 shared libraries are more commonly referred to as Dynamic Libraries and are at the same time both simpler to use and more sophisticated. Dynamic Libraries are very easy to produce and information concerning addressing need not be supplied to the compiler. Hence not only are jBC libraries Dynamic but libraries produced to hold jBC subroutines will be as well. Large applications sharing many subroutines benefit greatly by executing on SVR4 systems.

Other Dynamic/Shared Libraries

There are a number of versions of shared and dynamic linking concepts. Where any form of shared library exists it has been used in the jBASE port to the particular flavor of UNIX.

jBC Compiler and Runtime

- The jbc command will normally produce an executable UNIX file with all stages in between transparent to the developer;
- It is a sophisticated optimizing compiler written with UNIX tools designed specifically to write compilers. The lexical analyzer for jBC is written using lex and the parser is written using yacc;
- Symbol table management and other supporting functions are written in C, as are the libraries that support the compiled application. This ensures that the whole system, including any application produced using jBC, is portable and very efficient;
- The jBC compiler is an optimizing compiler that minimizes the use of memory and temporary workspace. Unlike some compilers the code produced is neither interpreted or stack based;
- The jBC compiler first produces C code, which is then optimized to reduce the number of steps taken to execute jBC statements. This is then passed through the optimizing C compiler. The result is code that is optimized twice, once in terms of the general application and once at pure assembler level;
- The optimized object code produced by the jBC compiler is linked with the jBASE libraries and the standard C/Windows libraries before it finally ends up as an executable file;
- * The application specific code and the jBASE libraries are all written around the concept of shared program text. ;
- * jBC runtime code is treated in the same way as any other Unix or Windows executable or C object file.

General Programming Advice

jBC Programming Advice

The advice given here is provided as a guide for jBC programmers. Every programming department will have its own view of procedures and standards but we do recommend that you should read this section, even if you then decide against the advice given here.

Formatting

The jED editor and the jfb formatting tool contain formatting algorithms that most developers will find acceptable. Two different formatting methods are available where the CASE construct is concerned as there are two equally valid formats for this statement.

The default indentation will use four spaces as the indent level and start each line that is not a label at indentation level 2 (Column 8). However, this may be changed should the internal standards of a particular organization differ from this. Labels are formatted to start in the first column of the source line. This gives the general appearance shown here:

```
Label:
        CRT "The first main line"
        IF Variable = 10 THEN
           CRT "Variable = 10"
        END ELSE
           BEGIN CASE
           CASE A = 9
              CRT "A = 9"
           CASE 1
              CRT "A is something else"
           END CASE
        END
        LOOP
           A = 10
        WHILE A DO
           A--
        REPEAT
        FOR I = 1 TO 5
           CRT I
        NEXT I
```

The jED editor and the jfb command will also align comments sensibly at a particular column providing that the source code does not encroach upon this set column.

We recommend that all jBC programs are created and edited via the jED editor. Executing jED with the {b} option or by entering BION command within jED will automatically turn on context sensitive editing. This feature will cause automatic indentation of the source code as the programmer types it in.

JFB

The jfb command is used to produce source code listings in a standard format for listing to the terminal or printer. The format of the command is as follows:

jfb -Options FileName Itemlist (Options

Option	Explanation
-A or (A	alternate indenting of CASE statements.
-C or (C	indent comments with the source code not column 1
-Ln,m or (Ln,m	set indentation to n spaces, with initial set at n*m
-Mnn or(Mnn	set maximum number of indentations to nn, default 10
-N or (N	do wait for keyboard input between pages
-P or (P	send output to the current printer queue
-Snn or (Snn	set the percentage split of code to comments to nn%
-V or (V	display indentation with + character

For example, to list the file batch.b in sub-directory source to the printer, indenting by four spaces per level and starting non-labeled code at 8 spaces from the left margin:

jfb -L4,2 -V source/batch.b

Structured Coding

The use of structured code is widely recognized to improve readability and maintenance of code. The jBC code contains many structured statements that allow the programmer to avoid producing "spaghetti" code. In particular the LOOP and FOR statements coupled with BREAK and CONTINUE, will avoid the need for the GOTO statement within a looping construct. The use of CASE statements is also recommended.

The GOTO statement should be avoided at all costs, although there are some valid instances when it may be useful such as branching to error routines that will never return.

Naming Conventions

The jBC compiler is a case sensitive compiler. All keywords within the language MUST be specified in upper case. Identifier names, may however be specified in any combination of upper and lower case characters. Over time we have found that the use of capitalized variable names greatly improves the readability of the code. Thus we have adopted this as a standard technique. The term capitalization refers to naming identifiers like this: FileName; RecordName; StopFlag.

You will also find that the compiler will not recognize variable names that are reserved words. Therefore the statement:

CRT = 0

is not allowed. The compiler will point out the usage of keywords as identifiers at compile time. The migration tools supplied with the system will spot this usage and convert the names to their capitalized form. The above statement will become:

Crt = 0

jBC Includes

Before starting to compile basic source files, any include items required by the basic source code should be located. If include files are located in other directories, links should be created.

For example, MAINPROG is a basic source code item with an include statement:

INCLUDE IncludeFile StandardEquates

The IncludeFile is located in a general purpose account, now a directory, called anotherdir.

A Q Pointer in the MD must be used to specify the Includefile.

Once all include files references have been processed, the program is ready for compilation. The jBASE command BASIC should be used to compile the basic source code programs. The BASIC command should only be executed from the application directory - NOT from an alternate directory making use of the link file. The application id should also be used when using the BASIC command. The reasons for using the application id and directory is that the necessary environment variables for the compilation process will have been setup and all the permissions will be correct, whereas attempting to compile from another user id or directory may cause later very confusing problems due to incorrect permissions or pathname assignments. The format of the BASIC command is detailed below.

BASIC SourceFilename Itemlist (options

The BASIC command uses a sophisticated set of compilation tools to compile and link jBC source code into C object code. The BASIC command will produce a dollar item, an item with the same name as the source item prefixed by a $. The dollar item will be placed in the same file as the original source item.

Cataloging and Running Programs

The jBASE CATALOG command is used to create executables and dynamic link 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 jBC 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.

CATALOG

The CATALOG command is provided as a front end program to convert object files generated by the BASIC command into main program executables and shared libraries/DLLs of subroutines.

CATALOG -Llib -obin -v -cExternalCLibs BP PROGRAM1 SUB1

Where options are:

Option	Explanation
v	verbose
Llib	alternative lib directory to use for shared libraries
obin	alternative bin directory to use for executables
cExternalLibs	external C library functions

Main program executables by default are copied in the home directory "bin" directory. Subroutine object files are collated into evenly sized shared libraries and then by default placed in the home directory "lib" directory.

In order to be able to rebuild a shared library the object file is retained in the "objdir" subdirectory of the "lib" directory. These object files are no longer required once all the shared libraries have been debugged and ready to release.

The CATALOG command invokes the jBASE jBuildSLib command with the subroutine object file to construct the shared libraries. The jBuildSLib command then links several object files together with relevant references to other required libraries and creates a shared library/DD.

It should be noted that every time a *subroutine* is cataloged, jBuildSLib is invoked to rebuild the shared library/DLL. However when the CATALOG command is issued with an active select list of program names, the rebuilding (i.e. jBuildSLib) is deferred until the list has been fully processed. This means that each shared object/DLL is only rebuilt once, as opposed to once for each subroutine. So when cataloging subroutines, it is much faster to work from an active select list. The same is true when decataloging subroutines.

To force the CATALOG command to place executables and shared libraries in alternative directories to the "bin" and "lib" directory in the current home directory, the following environment variables can be set.

To redirect main executables use:

```
export JBCDEV_BIN=/usr/global/bin (Unix)
set JBCDEV_BIN=C:\GLOBAL\BIN (Windows)
```

To redirect subroutine shared libraries use:

```
export JBCDEV_LIB=/usr/global/lib (Unix)
set JBCDEV_LIB=C:\GLOBAL\LIB (Windows)
```

To link with external C function libraries:

```
export JBCDEV_CLIB=/usr/global/clib (Unix)
set JBCDEV_CLIB=C:\GLOBAL\CLIB (Windows)
```

Programmers should be aware that by convention jBC program names have an extension of ".b" (and also ".B" on Windows). The jbc command expects programs specified as "name.b". The BASIC and CATALOG commands do not require programs to have a **.b** extension because they add one when necessary. This can be seen when the verbose option is used on BASIC and CATALOG. If a main program has a **.b** extension, it is dropped when the program is cataloged. For example, if the name of the program is CUSTMAINT.b, the resulting executable created by CATALOG is called CUSTMAINT. If a subroutine has a **.b** extension, then it's internal name should **not** have the extension:

Valid	Invalid
UPDATECUST.B	UPDATECUST.B
SUBROUTINE UPDATECUST(a,b,c)	SUBROUTINE UPDATECUST.B(a,b,c)

In this instance the subroutine would be called as:

```
CALL UPDATECUST(x, y, z)
```

JLIBDEFINITION FILE

The jLibDefinition file is used by default to configure the size and naming convention for each shared library. If the "lib" directory already exists then the jLibDefinition file is read from the "lib" directory. If this is the first time the "lib" directory is created then the jLibDefinition file is read from the "config" subdirectory of the jBASE release directory.

The configuration options contained in the jLibDefinition file are as follows:

%n - Use sequence number from 0 upwards

%a - Use account name, truncated to 8 characters.
%f - Use source filename

libname	Naming convention to use for library
exportname	naming convention to use for export file
objectdir	naming convention to use for object directory
maxsize	maximum size of the shared objects based on object size; default is 1MB for Unix and 8MB for Windows
baseaddress	base address for the first DLL created in a library (Windows only)

e.g. a "libname" definition of lib%a%n.so would result in libraries named as follows:
libAccount0.so
libAccount1.so

WINDOWS CONSIDERATIONS

- Use of the %a definition should be restricted to small account/user names only, as the performance of scanning directories can be severely downgraded when the library file names as greater than 8.3 in length.
- Because of a virtual memory overhead per DLL, it is best to have as few DLLs as possible. The default size is 8MBytes. If it is set too high, the linker cannot build the DLL. As CATALOG bases the size of the DLL on the sum of the component objects files, this equates to a DLL that is approximately 5 Mbytes.
- An attempt is made to make sure that the users DLLs do not need to be relocated as this process causes them to become private (i.e. not shared). This is done by setting a unique base address for each generated DLL. The "baseaddress" entry is used for this, the default is 0x30,000,000 (or 768MByte). The jBASE system DLLs are based at 0x20,000,000 (512MByte) so this value should not be used. NT System DLLs are at 0x70,000,000. The first user DLL (normally lib0.dll) will be based at this specified address, with each subsequent one incremented by "maxsize". This should avoid any overlap since the "maxsize" value is based on the size of the objects and not the DLL which is usually about 75% of this figure.

DECATALOG

To remove released executables or subroutines, use the DECATALOG or the DELETE-CATALOG command.

```
DECATALOG FileName ProgramName
DELETE-CATALOG ProgramName    [Unix/Linux only]
```

These commands will remove an executable program from the "bin" directory or remove a subroutine from the shared library.

Running jBC Programs

Before a jBC program can be run it **must** be cataloged. CATALOG creates an executable file that can be launched in the same way as any other executable:

- from the command line
- from a Unix script file (e.g. .profile)
- from a Windows cmd or bat file
- from a Windows shortcut
- from a PROC
- from a PARAGRAPH
- using EXECUTE/PERFORM
- using ENTER

The RUN command

jBASE also provides the RUN command to launch an executable. This exists solely for compatibility with older systems. Because there is a resource overhead in using RUN we recommend RUN is not used. Note that using RUN does not remove the requirement to CATALOG jBC programs.

RUN SourceFileName ProgramName (options

The SourceFileName is used as a place holder only, so specifying a different source file does not change the version of the program that is run.

jBC DEBUGGER

Introduction

The jBC debugger is a fully featured, interactive diagnostic utility that gives the programmer full access to the program variables and files. It will allow examination of source code, save and restore of debug settings and full access to system commands from within the debug shell. As such it is a powerful tool for detecting and fixing errors within jBC source programs. The main features of the debugger are:

- Set and delete breakpoints to halt program execution. These can be simple line number breaks or based upon the result of an evaluated expression.
- Set and delete the tracking and display of variable

contents.

- Display any number of lines from within the current source.
- Locate text in the current source.
- Examine and set breakpoints in source files other than the current one.
- Display and modify the contents of any variable.
- Execute a chosen number of source program lines before re-entering debug.
- Redirect debugger interaction to another device or terminal.
- Save debugger status to a file and execute debugger commands held in a file.
- Execute system commands and return to debug.

Entering the Debugger

The jBC debugger will be entered a number of ways:

Once the debugger is entered, an identification message is displayed and the debug shell prompt is displayed. The message gives the reason for the program entering into debug, the line number about to be executed, and the source file name. The final line is the debug prompt, after which the user is expected to enter a debug command.

The following examples show the display after entering the debugger in various ways.

Using the -Jd Option at Runtime

```
ExampleProg -Jd
Option -Jd seen on command line
Source ExampleProg.b,Line 1, Level 2
jBASE debugger..
```

A DEBUG Statement in the Program

```
ExamleProg
DEBUG statement seen
Source ExampleProg.b, Line 39, Level 2
jBASE debugger..
```

Using <Ctrl-C> Key from the User Terminal

```
Interrupt Signal
Line 157, Source ExampleProg.b
jBASE debugger ..
```

Receiving a kill -16 Command from another Terminal (Unix only)

```
Signal 16 seen from signal handler
```

```
Line 73, Source ExampeProg.b
jBASE debugger ..
```

Run Time Error
For example, when a variable containing a string is used as if it contained a number, the following is seen:

```
Non-numeric value -- ZERO USED ,
Variable "XFER.ID", Line 78, Source ExampleProg.b
jBASE debugger ..
```

Debug Arguments at Run-time

When a jBC source program is executed, there are a number of command arguments that can be passed to the run-time libraries, some of which relate to the operation of jBC debug. These are as follows:

-Jd	The debugger is entered at the start of the program, immediately prior to executing the first jBC command.
-JD	The debugger is entered at the start of the program, immediately prior to executing the first jBC command. The debug session remains active, even if a new program is EXECUTEd or CHAINed.
-Jp{:Path...}	This specifies to the debugger where it can find the necessary sources it needs at run time. Path can comprise multiple jBC filenames or jBC filenames, as long as they are each delimited by a colon. When the debugger attempts to open the source, it will start looking in the leftmost filename specified. If this argument is not given, the default is the current directory. This option can be overridden from the debug prompt using the "p" command.
-JrDeviceName	The debugger output is redirected to device DeviceName rather than standard output. This allows debug to send its output to a file, pipe or a terminal other than the current one in use by the program. For example, -Jr/dev/tty8b will redirect output to device tty8b

Note: If the application performs a CHDIR() function, and the debugger needs to access a file in the "current" directory by default, then it will attempt to access it in the directory specified by the CHDIR() function and not the one from which the program was executed.

Examples

```
menu -Jd
```
This command will start up the menu program and enter the debugger before executing the first command with a message similar to:

```
Option -Jd seen on command line
Source MENU.PROCESSOR.b, Line 1, Level 2 jBASE debugger..
```

Normal debugging operations can now be carried out.

```
menu -Jd -Jpinvoices:inv.routines:mainlib
```

This command executes the menu program and enters debug. Any debug commands requiring reference to the source, such as "w", will then look for it in a directory other than the current one.

The directories searched are listed after the -Jp option, and they are searched in order starting from the leftmost directory given. If a required source for the main program or an external subroutine is not in the invoices directory/file, then the directory/file inv.routines followed by mainlib will be searched in turn. If the
source file is not found, an error message is returned specifying the source required.

As the executable code and libraries are usually held in separate directories from the original sources, this is a very useful option.

Debugger Commands

This section details all the commands available to the user from the jBC debug prompt.

References in the Command column to expr refer to an evaluated expression. Expressions are detailed after the command table. The current file name and current line number are internal debugger variables. On entry to the debugger these are set at the current program file name and line number about to be executed.

Many of the commands detailed here are not available when a program has been compiled with the limited debugger. The limited debugger is linked to the program when the -J04 argument is used on the jbc command line. The -JO4 option is normally only used on production release applications. The ? command will list all commands available. All commands are available when the full debugger is in use.

Restrictions

If you have a Command Level or Break/End Restart feature in effect, or the break key is disabled, the available options are restricted to:

a	Abort program
q	Quit program
c	Continue (may be allowed, depends on reason debug was entered) end Terminate debugger
o	Log off

Note that there are several ways in which the break key can be disabled. You can use commands such as INHIBIT-BREAK-KEY or BREAK- KEY-OFF. In these cases, the debugger is never entered. Another way is to execute a BREAK OFF statement within the program. In this case, the debugger will still be invoked if a run-time error occurs - such as trying to read a record from a non-file variable.

Command List

?
Display a help screen showing all available debug commands and the program status.

>filename
Open and truncate the file filename and send it the current breakpoints and trace table entries. This can be used in future to replicate the current environment by the use of the command. note that you may write debugger scripts yourself with an editor rather than use the > command.

<filename

Open the file filename, then read and execute each line as if it has been entered at the keyboard. Any current trace or breakpoint table entries are deleted then replaced by those recorded in filename.

!command
Spawn another process and execute the command. The previous command thus used can also be recalled and executed by the !! command.

<Ctrl>+D
Display the next 11 lines of source in the current file.

Nn
Set the current display line to line nn in the current file and then display the line. Note that the program execution counter remains unchanged, it is only the display pointer that is changed. A command such as s (see later) will correctly execute the next line in the programmed sequence, not the newly displayed line.

#text
Ignored, and so can be used as a comment line in debugger scripts later executed with the command.

a{-nn} {mm}
Kills the jBC program and any parent process or program that called it. The program aborts with an exit code of 203, and the kill signal is sent to any parent process. The nn value is used to change the exit code, whilst the mm value changes the signal number sent with the kill command. Operation of this command can be altered by setting the Command Level Restart option - see the System Administrators Reference Manual for more details.

b
Display all currently active breakpoints.

b {-t} nn{,file}
Set a breakpoint at line nn in the current file or that specified by the file modifier. If the -t option is specified then the breakpoint will cause a display of all the trace variables rather than halting the program.

b {-t} varname
This form of the b command will cause the debugger to be entered whenever the contents of the specified variable are changed.

b {-t} ex1 op ex2 {AND|OR .....}
Set a breakpoint at the line whose value is obtained by performing the operation op on expressions ex1 and ex2. The operator can be one of eq, !=, <>equal is also available. See later for a full description of expressions. The -t option will cause the debugger to display all the trace points rather than halting program execution.

c
Continue execution of the program.

d {-tbed} {*nn}
Delete breakpoint and/or trace table entries, and will normally prompt for confirmation. The t and b switches refer to trace and breakpoints respectively. The * switch deletes all of the specified entries without prompting. The nn switch deletes the entry nn in the given trace or breakpoint table, also without prompting. The d and e switches respectively disable or enable the given entry without removing it from the table.

e name
Edit the file specified by name. This file is then the file used by other debug commands such as <Ctrl>+D.

end
Synonym for "quit".

f {on|off}
A debug breakpoint is set for a filename change. This break can be set to on or off. If the program is continued (C command) the debugger will be entered the next time the source file changes.

h {-rs{n}} {nn|on|off }
Displays a history of the source lines executed, and current status of the debugger commands used. The on and off switches toggle the recording of lines executed, and when on, the nn value gives the number of executed lines to display (1024 maximum). The -r switch displays in reverse order, and -s{n} shows n source lines.

j {-g}
The j command displays a complete history of both GOSUB and external subroutine calls. When issued without options the

command will only display information about the current program or subroutine. The -g (global) option will show a breakdown of the entire application.

l {-acf{nn}} text
Locate the string text in the current file. The switches used are: *a* to look for every occurrence; *c* to make the search case insensitive; *nn* to limit the search to the next *nn* lines; *f* to start the search from the start of the file. The command *l/* will execute the previous locate.

m
Displays the current memory status. Shows space allocated by the function malloc().

n {nn}
Displays the next nn lines of source from the current file, which is automatically loaded by the debugger if the p command has been used or it resides in the current working directory.

off
Enter o or off to log off. If you enter off (or OFF), the effect is immediate. If you enter o (or O), you will be prompted for confirmation. The same restrictions apply as for the OFF command; if there are non-jBASE programs active, OFF will only terminate jBASE programs until it encounters the first non-jBASE program - probably the login shell.

p {pathlist}
Defines the list of directories and pathnames (delimited by :) that the debugger will then search to find source codes. p without a pathlist displays the current Path.

q {nn}
Quit the program. nn is the termination status returned to any calling program.

r device
The debugger will take all input from, and send all output to, the specified device. Note that if the device is another terminal (or Xterm shell), that you will need to prevent the target shell from interfering with the input stream by issuing the sleep command to it. A large value should be used or the sleep should be issued repeatedly in a loop.

s {-t{m}d} {nn}
Continue execution of jBC code in single line steps before returning to debug. The value nn changes the number of lines executed before returning to debug. The *-t* switch is used to display the trace table after every line executed, rather than wait for entry to debug. The *d* switch sets a delay before executing each line of code. m is used to set the delay in seconds (default is 5 deci-seconds).

S {-t{m}d} {nn}
Same as s except this will 'step over' subroutine calls, the code within the subroutine will not be displayed.

t
Display the current trace table.

t {-fg} expr
Add the value specified by expr to the trace table. When debug is entered, all the values in the table are displayed. The f switch is used to fully evaluate expr, whilst the g switch extends the display of expr to all levels.

v {-gmsrv} {expr}
Evaluate expr and display the result. The effects of the switches are: *g* to extend the display of expr to all data areas. *m* to allow variable modification within expr. When a variable is modified with the m option binary characters may be entered using the octal sequence \nnn. The sequence \010 would therefore be replaced by CHAR(8) in the modified variable. The sequence \\ evaluates to the single character \ and a sequence such as \x evaluates to the single character x (i.e. the \ will be lost).

w nn
Display a window of source code. The default is 9 lines with 4 before and after the current one. The value nn is used to change this parameter.

Debugger Redirection and Pipes

The debugger provides the ability to redirect the results of its internal command set to a file or through a pipe to a command. This is a very powerful feature of the debugger.

The following commands allow this feature:

v Display Variable(s)
h Display History Trace
b Display Breakpoints
t Display Trace Table

Here are some examples of this feature:

v | pg	Pipe through the pg filter
v X<3> | hd	Show field 3 in hex mode
t Varx >> VarxTrace	Set trace for Varx (redirect saved)
s -t 999	Assuming trace above, each step will display the value of Varx and append the output to file VarxTrace
t > tracetable	redirect trace points output to file
v Record > file	Display variable contents to a file

file

Modifying Trace and Breakpoint Tables

d {-bdet} {nn|*}

-b	refers the command to the breakpoint table only
-d	disable or un-set the table entry referred to
-e	re-enable the table entry referred to
-t	refers the command to the trace-variable table only
nn	the trace or breakpoint entry number to delete, enable or disable
*	refers to all the trace or breakpoint entries

This command is used to delete entries in the variable trace table and the breakpoint table. Unless the * switch is used, table entries can only be deleted one at a time. Used on its own, this command lists each variable in the trace table and each entry in the breakpoint table in turn, and prompts the user for deletion.

d -t
The command deletes all the entries in the variable trace table. This is done immediately without asking for confirmation.

d -b
The command deletes all the entries in the breakpoint table. This is done immediately without asking for confirmation.

d -bd
The d switch is used to disable the entries referred to. In this case, all of the breakpoints are disabled. As a result the execution of the code will continue as if there were no breakpoints set. The complementary e switch can be used to re-enable the entries.

*d **
This deletes all trace and breakpoint table entries without asking for confirmation. Care should be taken with this, as the system generated t0 entry is also deleted.

d -t 4
This command deletes the fourth entry, t4, in the variable trace table. No confirmation is asked for, and the later entries do not shuffle up the table, i.e. entry t5 will remain as t5.

d -b 2
This deletes the second entry in the breakpoint table. No confirmation is asked for.

*d -b **
This will delete all the breakpoint table entries without asking

for confirmation.

Execution History

h {-rs{n}}{nn|on|off}

-r	display lines in reverse order
-s	display n lines of source (default is 1)
nn	limit the history buffer to nn lines of source
on/off	toggle the saving of source lines executed

This command keeps track of the source command lines executed during a debug session. The last 1024 lines are held in a circular buffer, and when full, the most recent command line displaces the oldest. It can be toggled on or off, and it is normally switched off by default. The command and switches have no effect unless activated by switching on. Commands executed from subroutines and CHAINed programs will also be displayed.

h on
This will switch on the command line audit, and every line of code subsequently executed during the debug session will be logged for reference, until the command is switched off or the debug session ends. The following commands assume the command line history is switched on.

h 20
This sets the number of lines of code to display at 20 lines. The default, and maximum value is 1024.

h
Used in its simplest form, the command displays all the entries in the buffer to the maximum number set.

h -s3
This displays the last three lines of code executed.

h -rs10
The last 10 lines of code executed are displayed in reverse order, i.e. the command last executed is shown first.

h off
This switches off the history trace.

Locating Strings
l{-acf}{nn}text

-a	show all occurrences (defaults to the first occurrence)
-c	ignore the case of any text
-f	start the search from the first line of the source

-nn	limit the search to the next nn lines of source
text	text to locate

This command locates text in the source file currently being executed and displays the line or lines of code containing it. The file to be searched may be changed by using other debug commands such as *e*.

l Heading
Used in its simplest form, the command will search the source from the current line position, to the end of the file, for the first occurrence of the text "Heading". If an exact match is found, then the line is displayed.

l -c Heading
The -c switch is used to ignore the case of the text. In this case, the first line found with the "Heading" text in any variety of upper and lower case letters will be displayed.

l -a NAME
The -a switch is used to locate and display ALL the source lines containing the text "NAME" in upper case letters only.

l -ac name
This command will display all lines of source code that contain the text name, with the characters being in any combination of upper or lower case letters.

l -a22 NAME
The characters NAME will be searched for, and if located in the next 22 lines starting from the current one, each line where it is found will be displayed.

l -f INVOICE.b NAME
The -f switch is used to search from the beginning and display the first line found containing the characters NAME from the file INVOICE.b held in the /usr/tutor/BP directory.

l -acf ./PAYMENTS.b money
The most complex form of the command as shown will search the PAYMENTS.b source, held in the current directory, and display every line from it that has the text money in any variation of upper or lower case letters.

Execution - Single Stepping

s{-tcgd{n}}{nn}

-t	display trace table after each source line executed
-c	only count the lines of source in the same CALL level
-g	only count the lines of source in the same GOSUB level
-d{n}	enter a delay in increments of 100 milliseconds between executing lines of source. This is incremented by the n

d{n} value entered.

nn execute the next nn lines of source before re-entering debug

This command is used to execute the program in steps and to re-enter debug after the execution of a given number of lines of code. Traced variables are displayed after debug is re-entered, and any screen display within the executed code is shown as normal.

s
The simplest form of the command executes the next line of the code and then re-enters debug.

s -t
The next line of code is executed and the contents of all entries in the trace table are shown.

s -t4
The next four lines of code are executed displaying the trace table entries before re-entering debug.

s 20
This command executes the next 20 lines of code before re-entering debug.

s -td5 200
The command executes the next 200 lines of code. The *-d* switch sets a delay in increments of 100 milliseconds between each line executed. The 5 denotes that a 500 millisecond, or half second delay is set before executing the next line. The default value is 1, or 100 milliseconds. The *-t* switch ensures that the trace commands are shown after the execution of every line. While this process is continuing, debug can be entered by breaking into the program as normal. This is a very useful command to use when a run-time error occurs in a program, and the area of code responsible needs to be found quickly. With the *-d* switch set, it is also possible to speed up or slow down the execution of the code if the initial value chosen is too fast or slow. This is done by entering a number from the keyboard in the range *0-9*, which alters the delay to the given number of 100 milliseconds increments.

s -d3t 500
The command will execute the next 500 lines of code with a delay factor of 300 milliseconds between each line. The speed of execution can be increased or decreased by pressing the numbers *0-9* on the keyboard during execution. In addition to this, the *-t* switch means that the contents of the variables trace table will be displayed after **every** line of code executed.

Display Variable

V{-gvmrs}
V ANS
The simplest form of the command will display the contents of the variable next to the variable name, in this case ANS. This will only produce a display if the source is at level 1, or in the home directory. If the variable has not been assigned, the value (NULL) is displayed. If the value assigned happens to be null, however, then a blank (null) will be displayed next to the variable name.

v -g ANS
If the variable in question resides in a different data area to the local level (COMMON or NAMED COMMON), then the -g switch should be used to display the variable contents. This extends the display of the variable to data levels, and is particularly useful when executing a subroutine in a sub-directory or library.

v -m ANS
The -m switch displays the variable and contents, but in addition allows the user to modify the contents. An equal sign is shown after the variable contents, and any characters or numbers entered followed by a carriage return are taken to be the new value of the variable. Entering a carriage return leaves the variable contents unchanged. The character sequence \nnn is replaced by the binary character defined by the octal number nnn. Therefore the sequence \376 would be
replaced by a field mark.

v -gv ANS
This command displays the value held in variable ANS no matter what the current level of the source. In addition, the -v switch shows the type of variable (string or numeric), its memory location, and size.

v -r NAME
This command displays the contents of the variable NAME at the start of the next line. The -r switch provides a raw character view of the variable name and value.

v -s NAME
The -s switch shows a short view of the variable being the first 128 bytes.
The * and ? characters can also be used within the variable name as wild card characters. The ? denoting a single occurrence of any character, and the * denoting any number of occurrences of any character.

Examples

v A*	displays all variables beginning with the letter A
v A???	displays all four letter variables beginning with the letter A

v *-INV	displays all variables ending with the characters -INV
v *ENP*	displays all variables with the characters ENP within their name
v LIS(2,*)	displays every element in the second row of the dimensioned array LIS

Internal Debugger Variables

The jBASE debugger supports a set of internal variables.

Variable	**Description**
$c	CALL level
$f	current source file name
$g	GOSUB level
$m	memory usage (not available on all platforms)
$n	current source line number
$r	reason the debugger was invoked
$s{n}	display the current source line (in a window on n lines)
$u	display the variable memory usage

These can be treated as normal program variables. So the command:

```
v $f
test1.bjBASE debugger->
```

shows the program soure name is test1.b. The absence of a new line is intentional. The elements \n and \r can be used to build up more complex expressions such as:

```
jBASE debugger->/"Line "$n" in program "$f\n
```

Text literals are enlclosed in double or single quotes.

The internal variables can also be used as part of trace table variables:

```
jBASE debugger->t "I am currently in line number " $n " from
source file " $f \n
t 3: "I am currently in line number " $n " from source file " $f
\n
jBASE debugger->s
0003 K = J : "x"
I am currently in line number 3 from source file test1.b
jBASE debugger->
```

DEBUGGER SYMBOL TABLES

The jBC compiler produces debugging symbol tables for use by the debugger at runtime. These symbol tables are produced for each data area in the program. Issuing the command v -g will display the value of every symbol at every data level within the program. There are three types of data area within a jBC program:

Local Data Areas

Local data areas have scope only within the current source file you are debugging. This will either be the variables used within the main (calling) program or the variables assigned in the current subroutine. By default the v command will only operate on the current local data area. The g option will cause the scope of the v command to be extended across all known data areas.

Global Common Area

The Global Common area contains all the variables that were declared to the compiler using the COMMON statement without naming the common area.

Named Common Areas

There may be many instances of Named Common Data areas within a single executable. Each area maps directly to the named common definitions within your programs. Therefore a statement such as:

```
COMMON /JIM/ A,B,C
```

will produce a named common data area called JIM for the debugger to use.

> *Note: If your program contains the same variable name in both local and global common areas the debugger will operate on each instance of the name in turn.*

jBC Language Reference

This section details all the commands and functions available in the jBC language, together with their command syntax.

It starts with a description of the language constructs, and how to make use of the features to build an application. This is followed by detailed descriptions of the jBC commands and functions in alphabetical order.

If you are new to jBC, BASIC or related languages, you should study the language constructs first, before attempting to write any code. If you are an experienced programmer, you may still find the descriptions of the language constructs useful. They will certainly help you to identify any elements of the language that may differ from the other versions of BASIC with which you are familiar.

The jBC commands topic is designed to enable you to look up command syntax quickly. Examples are also provided to illustrate the usage of each command.

Syntactical Elements

argument	A value or variable passed to a procedure or function at the time of execution.
constant	A value that does not change throughout the program.
expression	A component part of a program that defines the computation of a value.
function	A function is a program unit in the C or jBC language. It is a section of code that can be passed values and may return a value.
identifier	A string of characters used to identify or name a program element that can be a variable, function statement or a higher level unit of the program.
label	A numeric or alphanumeric identifier associated with a line or statement in the program, and used to refer to it from another section of the program.
operand	A quantity on which a mathematical or logical operation is performed.
operator	An entity that can be applied to one or more values or operands to generate a result or resultant.
parameter	This refers to data passed to a function, subroutine or procedure.
resultant	The value output as a result of the action of a function, operator, expression or procedure upon given data.
statement	A unit with which a program is constructed.

subroutine	A self enclosed set of program statements that are not part of the main program flow. Program execution control is transferred to it and only on completion does control pass back to the main program body.
value	A string of numeric, alphabetic or alphanumeric characters representing a quantity, data or information.
variable	A string of characters used to identify a changeable value stored within the computer.

Data Records

Data is the form in which information is stored on a computer. The data stored, with its correct context, will provide the user with a particular piece of information.

Variables

A variable is a representation of data. The data stored in a variable can be either a string or a numeric, which can change dynamically during a program.

Strings can be of any size including the NULL string of 0 bytes in length.

In many systems variables are "typed", I.E. It can only hold a certain type of data. The jBC language uses "typeless" variables, allowing any kind of data to be stored within them. In addition, there are no limitations on what may be stored in a byte within a variable. The full range of 8 bit values 0x00 to 0xFF may stored within variables.

There is no limit to the number of variables that can exist within a jBC program.

A variable name identifies the variable and can be contain almost any sequence of ASCII characters, starting with an alphabetic, apart from spaces, commas, underscores, hyphens and mathematical operators.

A jBC reserved word cannot be used as a variable name and the compiler will signal an error condition if it detects the illegal use of a keyword.

Constants

A constant is similar to a variable except that it holds a value that remains unchanged throughout the execution of a program.

The value of a constant may be either of a numeric or a literal string enclosed in quotation marks.

A string constant must be enclosed in single or double quotation marks, ”aString” or the '/' , /aString/character. Any of the delimiters will be included as part of the constant if the whole constant is enclosed between a different delimiter.

Examples

EQUATE PI TO 3.14159265359

Sets the value of PI to 3.14159265359

Assigning Variables

An assignment statement is used to allocate a value to a variable. The value assigned can be a number, literal string or any valid expression.

COMMAND SYNTAX

```
variable = expression
```

variable is the reference that will be used within the program to access the expression assigned to it. The value of expression is stored in memory and will be accessed directly whenever variable is referenced.

If during the execution of a jBC program, a variable is encountered which has not been properly assigned, then a default value of zero is assigned to it, the debugger is entered, and an error message is displayed on the terminal:

```
Invalid or un-initialised variable -- ZERO used

VAR varname, Line lineno, Source filename
```

Examples

1. Number = 250

assigns the value 250 to variable Number.

2. Sum = 27 + 31 + 19

will evaluate the expression 27+31+19 and assign it to the variable Sum. The same result could be obtained by the statement:

Sum = 77, and in fact the compiler will evaluate this at compile time.

3. Percent = (Number / Sum) * 100

shows that other variables can be part of the expression to be evaluated.

4. Name = "Jim"

assigns the string Jim to the variable Name.

Reserved Words

Reserved words are words that cannot be used within a jBC program as an identifier, variable name, subroutine labels or statement labels. The use of a reserved word will be shown when the program is compiled, and compilation will fail.

jBC reserved words are :

ABORT CRT GETCWD MATREADU READP SYSTEM

ABS DATA GETENV MATVAR READPU TAN

ALPHA DATE GETLIST MATWRITE READT THEN

AND DCOUNT GO MATWRITEU READU TIME

ASCII DEBUG GOSUB MOD READV TIMEDATE

ASSIGNED DEFC GOTO NE READVU TO

AT DEL GROUP NEXT REGEXP TRANSABORT

BEGIN DELETE GT NOT RELEASE TRANSEND

BITCHANGE DELETELIST HEADING NULL REM TRANSQUERY

BITCHECK DIM ICONV NUM REMOVE TRANSTART

BITLOAD DIR IF OCONV REPEAT TRIM

BITRESET DO IN OFF REPLACE TRIMB

BITSET DTX INDEX ON RETURN TRIMF

BREAK EBCDIC INPUT OPEN REWIND UNASSIGNED

BY ECHO INPUTCLEAR OR RND UNLOCK

CALL ELSE INPUTERR OUT RQM UNTIL

CAPTURING END INPUTNULL PAGE RTNDATA USING

CASE ENTER INPUTTRAP PASSDATA RTNLIST VAR

CHAIN EQ INSERT PASSLIST SELECT WEOF

CHANGE EQU INT PERFORM SENTENCE WHILE

CHAR EQUATE IOCTL PERROR SEQ WITH

CHDIR EXECUTE LE PRECISION SETTING WRITE

CHECKSUM EXIT LEN PRINT SIN WRITELIST

CLEAR EXP LN PRINTER SLEEP WRITEP

CLEARFILE EXTRACT LOCATE PRINTERR SORT WRITEPU

CLOSE FIELD LOCK PROCREAD SOUNDEX WRITET

COL1 FIND LOCKED PROCWRITE SPACE WRITEU

COL2 FINDSTR LOOP PROGRAM SQRT WRITEV

COLLECTDATA FLOAT LT PROMPT STEP WRITEVU

COMMON FOOTING MAT PUTENV STOP XTD

CONTINUE FOR MATBUILD PWR STR

CONVERT FROM MATCHES READ SUB

COS FUNCTION MATPARSE READLIST SUBROUTINE

COUNT GE MATREAD READNEXT SYNC

Arithmetic Expressions

An arithmetic expression consists of one or more numeric variables, constants or intrinsic functions operated upon by arithmetic operators.

Arithmetic Operator Precedence

Operator precedence describes to the compiler what order a complex expression should be evaluated in. For instance multiplication is performed before addition in a statement such as: Var = 8 + 8 * 2

When an operator is left associative its arguments are in the same order as the standard addition operator. This means its arguments are picked from the left towards the right. A right associative operator takes its arguments reading right to left. An example of a right associative operator is ++, which increments the variable to its right in the statement Var = A++. The entire precedence of jBC operators is shown below. The highest precedence is 1.

Operator Operation Precedence

( ) Expression in parentheses 1 left

++ -- Increment and Decrement 2 right

^ Exponential 3 left

* / Multiplication and division 4 left

+ - Addition and subtraction 5 left

A "L#2" Format string 6 *

: Concatenation 7 left

> < Relational operators 8 left

= # Equivalence 9 left

AND Logical AND 10 left

OR Logical OR 11 left

+= -= Assignment 12 right

Expression Evaluation

Expressions are evaluated according to the precedence of their operators but if an expression contains two operands with the same precedence the expression to the left is evaluated first.

Expressions enclosed in parentheses are evaluated first and then treated as a single expression. Within the parentheses, normal operator precedence applies.

Where parentheses are nested, then evaluation begins with the innermost parenthesized expression, followed by the others in turn.

Examples

12+4+16/4+36/9 evaluates to 24

6*8/12-3*2^4+"7" evaluates to -37

(2^3+1)/(27/(9/3)) evaluates to 1

Numeric Strings

A string can be used as part of an arithmetic expression as long as it only contains numeric characters, and is not null. If a variable is used as part of an expression but at run time it contains data that cannot be converted into a number, then the program will drop into the debugger and issue an error message.

Logical Operators

Logical operators are used to perform Boolean tests on relational or arithmetical expressions.

The logical operators available in jBC are:

AND & are used to perform a logical AND

OR ! are used to perform a logical OR

A logical expression operates on the outcome result being TRUE or FALSE.

An expression that evaluates to be zero or NULL is considered FALSE, whilst all other outcomes are considered to be TRUE.

Logical operators have lowest precedence in any expression, and so any relational or arithmetic operation is performed first when used in a logical expression.

The result of a logical operation can be summarised:

Operation Result

X AND Y TRUE if X is true and Y is true.FALSE if X is false, or Y is false.

X OR Y TRUE if X is true or Y is true.FALSE if both X and Y are false.

The NOT function can be used to invert the result of the expression.

Pre and Post Increment Operators

jBASE allows the use of pre and post increment operators with variables in a similar manner to the C language. This allows the use of a statement such as:

Var++

This will increment Var by one. The operator -- decrements a variable by one.

The ++ and -- operators may also be used within expressions, where they perform slightly different operations dependant on where the are placed in relation to the variable. If the ++ (or --) operator is placed before the variable the variable will be incremented (decremented) before its value is used in the expression. If the operator is placed after the variable then value of the variable will be used in the expression and incremented or decremented only AFTER this.

So:

A= 5; CRT ++A Will display 6

A= 5; CRT A++ Will display 5

This can simplify the coding of iterative loops considerably.

Strings and String Operations

A string is a set of characters that may be numeric, alphabetic or alphanumeric (a mixture of numeric and alphabetic).

String Expressions

Strings may be referred to or built up as a string expression, which could be one of:

variable: A = "abcd1234"
constant: "ABCD1234"
Concatenation: "ABCD":"1234"
substring: A [5,4] (evaluates to 1234)

String Concatenation

Data strings are concatenated by using the CAT operator. The CAT and ':' operators are functionality equivalent.

Concatenation of two strings appends the second string to the first. Concatenation of a series of strings is done in order from left to right, and parentheses may be used to make a particular operation take precedence.

Substring Extraction and Assignment

Substring extraction takes out part of a string and assignment allocates that part to a variable or expression or as a value.

The original string can be a variable, dynamic array or dimensioned array, and the syntax used differs accordingly.

General String

S[start#,extract#] = expression

S is the original string variable, and start# gives the starting character position, whilst extract# gives the number of characters to extract from the string.

E.g. A="abcdef" ; X=A[3,2] assigns characters cd to X

Dynamic Array

S{*{field#{,value#{,subval#}}}*}{[start#,extract#]}=expr

S is the original dynamic array. If referenced, field#, value# and subval# refer to the array's field, value and subvalue position respectively, whilst start# gives the starting character position in the array element, and extract# gives the number of characters to extract from it.

E.g.   D001*2,3*[3,2] will extract the third and fourth characters from the third value in the second field of the file record D001. If D001 holds 'GE123^F]123]ABCDEF^29' then the characters CD will be extracted and assigned.

Dimensioned Array

S(row#{,col#}){*{field#{,value#{,subval#}}}*}

{[start#,extract#]}=expr

S is the original dimensioned array. If referenced, row#, and col# refer to the array's row and column position respectively. If referenced, field#, value# and subval# refer to the array's field, value and subvalue position respectively, whilst start# gives the starting character position in the array element, and extract# gives the number of characters to extract from it.

Using Negative Numbers

Negative numbers can be used as either the start or extract values in the expression. Specifying a negative number for the start of the substring will start the extraction those many characters from the end of the string. Therefore S[-1, 1] will return the last character in the string. Specifying a negative number for the extract value will extract up to and INCLUDING that number of characters from the end of the string. The following examples illustrate this:

With S="1234567890"
CRT S[-1, 1] displays "0"
CRT S[-3, 2] displays "89"
CRT S[2,-2] displays "23456789"
CRT S[-3,-2] displays "89"

Relational Expressions

A relational expression compares values or expression resultants to provide an outcome of true (1) or false (0). It consists of a relational operator applied to a pair of literal or arithmetic expressions.

RELATIONAL OPERATORS

Symbol		**Operation**
=	EQ	Equal to
>	GT	Greater than
<	LT	Less than
>=	GE	Greater than or equal to
<=	LE	Less than or equal to
# < >	NE	Not equal to
MATCH(ES)		To use pattern matching

Relational operators have a lower precedence than arithmetic and string operators, and so are performed after these within a complex expression.

An arithmetic relational expression consists of two numerics or numeric expressions separated by a relational operator.

All other combinations of a pair of string or numerics within a relational expression are treated as a string relational expression. In this case, numerics are treated as string values.

The null string as part of a relational expression is treated as being less than zero.

Program Layout

Blank Lines

A blank line within the jBC source code is one where no characters appear apart from the carriage-return, linefeed continuation characters.

Whitespace

Whitespace characters are used to separate words, commands and statements on a program line. It can be entered using the SPACE BAR or TAB KEY.

Presentation

The presentation of source code can greatly improve the appearance and readability of a program. Recommended programming style is shown in 'Programming Advice.'

Blank lines in the source program are ignored by the jBC compiler.

Spaces on a source program line are also ignored by the compiler unless they are part of a literal string.

This means that blank lines and spaces can be used to freely format the source code, making it more presentable.

Example

```
! Example of formatted code
CrtJim: ;* CRT subroutine
     CRT "Hello Jim" ;* Display the message
     IF   Day = 14 AND Month = 7 THEN ;* Special?
        CRT "Happy Birthday!" ;* Greetings
        CRT "29 Again!" ;* Hmmm!
     END
     RETURN ;* To caller
```

Comments and Comment Lines

A comment or comment line has no effect upon the execution of a jBC program, and may appear anywhere within it.

Comments may (and should) be used freely in a program to clarify an operation or to identify a particular part of the code.

Execution

A comment should start with one of the following tokens:

REM ! *

The comment character should be placed at the start of a line or statement and is ignored by the jBC compiler. The special comment line !! is used in conjunction with the jfb listing program. When the program is listed using jfb the !! comment will cause a page throw to be inserted in the listing. This can be useful for hard copy listings as it enhances the presentation of the code.

Example

```
REM This part of the code is to overcome
* the problem we have with different terminal
* types.
! START OF MAIN PROCESSING
NAME = RECORD<4,2> ;* Get name from record
```

Line Continuation

It is possible to continue jBC statement sequences or comments over more than one line.

This is done by using the \ character or the "..." ellipsis sequence at the end of a line. The line is then seen as one continuous line by the compiler. This is useful for formatting code lines that are hard to read on a single line.

Statements and Statement Labels

A jBC statement is an entity or command that can be sequenced with other statements to build up an application program.

Statements normally refer to a single operation and so are usually kept on a single line.

Several statements, however, may be placed on the same line, and where this is done, each statement must be separated from the next by a semicolon character.

jBC statements or program lines can begin with an optional label. This is used to allow the statement or line to be referenced from another part of the program.

Labels may be numeric or alphanumeric, but an alphanumeric label at the start of a line must end with a colon so that the system will recognize it. Reference to an alphanumeric label within the program does not use the colon as part of the label.

Program Files

Applications written with the jBC language would generally be composed of three distinct parts:

- Source Code
- Run-time Executable Code
- Libraries containing common functions and subroutines

Formatting Data for Input or Display

Variable Formatting

Numeric values in variable assignments, expressions or CRT, PRINT and INPUT@ statements can be formatted using format strings. This will produce a formatted output display of the raw data string.

Multiple format strings can be applied to a variable expression, and if used, will be applied from left to right. The syntax of this construct is as follows:

COMMAND SYNTAX

VARIABLE "{f}{£}{,}{d}{fieldwidth}"

f	This character specifies the field justification. L specifies left, and R specifies right justified within the field.
£	The £ character appends a £ character to the front of the field and takes up one position in the specified field width. The actual character output will depend on your locale setting - see the Internationalization chapter in the System Administrators Manual.

,	This puts in a , as a separator for thousands. It takes up one position in the specified field width.
d	This is a single digit giving the number of decimal places to display.
fieldwidth	Gives the width of the field in which the value is held, and can be padded out by: #... or #n fill with spaces given %... or %n fill with zeroes given *... or *n fill with asterisks given

Pattern Matching

Pattern matching functionality is provided allowing comparisons between string variable contents. This compares the string contents to a specified pattern or patterns, with the outcome being 1 (true) if the patterns match, or 0 (false) if the patterns are different.

Pattern matching is performed by using the MATCHES or MATCH operators.

The specified pattern for matching can contain any combination of the following within quotes:

{n}A	n is an integer that gives the number of alphabetic characters in the pattern.
{n}N	n gives the number of numeric characters in the pattern.
{n}X	n gives the number of alphabetic or numeric characters in the pattern.

The use of 0 as the value of n implies any number of the specified character, including a null value.

Conversion Codes for Data

One of the efficiency aspects of jBC and jBASE is that no matter what the format of data displayed on the terminal, only the raw data need be actually held on disk. This means that decimal points, £ signs, and other format incidentals are not saved, thus giving a saving on file storage capacity.

Raw data is subject to special conversion routines that enable their display in a meaningful format.

Similarly, previously formatted data can be stripped of its format and display characters before writing the record to disk.

jBC uses a set of conversion codes to perform output display formatting. These can also be used to deformat data before saving it in its raw or internal format.

Date Conversions

Dates are generally required in differing formats, and so an internal date format is used to store them on disk. As this is simply a number, a conversion code is required to format it to the required output display. The codes are also used to format strings.

Date conversion codes available are:

Code	Effect
D{y}{d}	The y value is the single digit 0 - 4 and specifies the number of digits in the year field. Default value is 4. The d value is a single non-numeric character used to delimit the day, month and year fields. If omitted, the month is given in alpha character format, otherwise the month is in two digit numeric format.
DI	This is the opposite of the above conversion, and when used to format a date output, will convert a date to internal format.
DD, DJ, DM, DMA, DQ, DW, DWA, DYDAY	These codes convert the day, month and year components of dates to and from internal formats.

The conversions can be used with the ICONV and OCONV functions (see later). The OCONV function converts internal formats to output formats. The ICONV statement converts external (Output) formats to internal form.

Program Flow

The following commands and constructs are used to change the execution path of the linear program flow by branching to another part of the current or even another program. The branches can be unconditional as with the GOTO and GOSUB statements, or conditional on the result of an expression as with the IF statement.

GOSUB

The GOSUB command is used to branch to a local subroutine, which is identified by the line label used. When a RETURN statement is encountered in the subroutine, the program is transferred back, and execution continues at the line following the GOSUB.

Care should be taken to structure the usage of GOSUB, and not to exit a subroutine other than with the RETURN command. RETURN always returns to the statement following the last executed GOSUB.

The GOSUB is generally used to perform ancillary processing outside the main program flow, such as error handling or logical separate functions of the program.

A similar command is GOTO, which unconditionally branches to the line beginning with the specified label. Where possible the GOTO statement should not be used as it makes code difficult to read and maintain. As the jBC compiler is an optimizing compiler, there is no performance penalty in using structured statements such as the LOOP or CASE constructs.

ON .... GOSUB

This construct is used to conditionally branch to one of a list of local subroutines depending upon the value of a variable or expression. The variable or expression resultant must be an integer in the range 1 to n, where the value n specifies the number of subroutines. Execution control passes to the nth subroutine in the list provided with the construct.

IF ... THEN

This construct allows the conditional execution of a sequence of statements. The conditional expression is evaluated and if the resultant value is 1 or true, then the sequence of statements following the THEN command are executed. If the value is 0 or false, the statements following the ELSE command (if any) are executed.

It is possible to perform actions based on complex testing of parameters by nesting, or successively repeating an IF ...THEN clause as part of another.

CASE

This construct allows the conditional execution of a sequence of statements and is used in a similar manner to the IF ... THEN construct. It differs in that the conditional expressions are examined one at a time, and so makes the program much more modular and easier to follow.

Looping Constructs

A loop is a series of statements that are executed repeatedly for a specified number of repetitions, or until specified conditions are met.

To repeat a procedure a set number of times, jBC provides the FOR ... NEXT loop. This allows the programmer to set a counter that is used to monitor the number of times the loop has been repeated. The repeat of the procedure can also be made dependent upon specified conditions.

The LOOP statement is used where the number of times a process needs to be repeated is not known, but depends upon the value of a particular parameter. The loop is repeated while a variable holds a given value, or until it contains a given value.

Loop constructs may be nested. This means that statements forming a complete loop may be enclosed within another loop. This in turn can be completely enclosed in another. Loops may be nested to any number of levels.

The BREAK (or EXIT) and CONTINUE statements can be used within loops to cause premature termination of a loop or to immediately begin the next iteration of the loop. They act upon the innermost loop in the program structure. BREAK or EXIT will cause the immediate termination of the innermost loop. CONTINUE will abandon the current iteration of the loop and immediately begin the next iteration of the loop.

Example

```
FOR I = 1 TO 10 UNTIL I = A
  CRT I
  A += C
  IF A = 5 THEN BREAK ;* End when A is 5
NEXT I
```

Subroutines

A subroutine is a self-contained piece of code that may be located outside the program itself. The code generally deals with a routine or procedure that is outside the mainstream processing line, such as error handling or functionally separate logic. On completion of the subroutine statements, execution of the program continues at the next statement after the subroutine call.

External subroutines are used to store commonly used routines utilized by many programs. A collection of external subroutines held in a file or directory is often referred to as a library.

A subroutine that is a part of the main program is accessed by the GOSUB statement, and a RETURN statement as the last subroutine command passes control back to the main body of the program, and execution continues on the line after the GOSUB statement.

A subroutine that is external to the program is accessed by the CALL statement. Such an external subroutine must always reside in its own source file and has the SUBROUTINE command as its first statement. A RETURN statement as the final statement returns execution control to the line following the CALL statement in the originating program.

Argument Passing

One or more arguments separated by commas may optionally be passed to an external subroutine as an argument list. An argument can be a variable, constant, array or expression representing a value.

When an argument list is passed to the subroutine, the SUBROUTINE statement must also contain an argument list with the same number of arguments as is being passed to it. Values passed in this way can be modified by the subroutine, and when the RETURN statement is encountered, the modified values are returned to the calling program.

COMMON and INCLUDE

The COMMON statement can also be used to pass values between programs. This has the advantage that this statement forces the variables to be stored in memory in the order in which they are presented.

Where the number of arguments to be passed is large, and where this may need to include common sections of code, then the use of the INCLUDE statement is recommended.

The INCLUDE statement allows common code to be stored outside the main body of the source program. The advantage is that changes to this code can quickly be made and have an immediate global effect on the source code, as well as maintaining a consistent and more efficient interface.

C Functions

C functions may be freely mixed with jBC programs and data may be passed to them and received from them. One or more C function is defined in its own .c file and is specified to the jBC compiler in exactly the same manner as a jBC .b file. C functions referenced in a jBC program must be declared to the compiler using the DEFC statement (see later).

Example

```
PROG.b
001 DEFC  INT Cfunc(INT) ;* C func returns integer
002 Number = 45
003 CRT Cfunc(Number)
004 BigNo = Cfunc(1000)
005 CRT BigNo
func.c
int Cfunc(Num)
int Num; /* Integer parameter */
{
return Num*1000; /* Multiply and return */
}
```

The programs are combined with jbc PROG.b func.c -o PROG. Note that the function is declared specifying the data types it expects but that the jBC compiler will resolve all type conversions for you. It is also possible to pass jBC variables to

C functions allowing the programmer to map C structures to jBC variables and vice versa.

Handling Arrays and Data Files

Arrays

An array is a variable where related data is held together as separate entities. Each entity is referred to as an element of the array. An array takes the form of an ordered set of data and each element can be addressed separately.

jBC allows the use of two types of array :

- Dimensioned arrays for holding data in the form of lists or tables.
  Related data is held using a single variable name, and indicating the number of separate elements involved. More details are available in the section on Dimensioned arrays later in the chapter.

- Dynamic arrays for holding file data and lists of unknown length.
  Data kept as records on the system may be accessed in the form of a dynamic array rather than in list or tabular format. Data held as a dynamic array reflects its structure as held on the physical disk. More details are available in the section on Dynamic arrays later in the chapter.

Dimensioned Arrays

A dimensioned array is a variable that has more than one piece of data stored within it. All the data within the array is usually related in an ordered fashion, and each separate entity is called an element of the array.

Dimensioned arrays generally take the form of a vector (list), or a matrix (table).

Vectors

ADDR(4)   6, New Road, Oldtown, OL8 7MB

The above one-dimensioned array is also referred to as a vector. The first element has a value of 6, the second a value of New Road, and so on. The first element is referenced by the variable ADDR(1), the second by variable ADDR(2) and so on.

Matrices

array **MULT(3,5)**

	1	**2**	**3**	**4**	**5**	**column**
row1	2	4	6	8	10	
row2	7	14	21	28	35	

```
row3      9     18    27   36    45
```

The above two-dimensional array or matrix holds part of a multiplication table. It consists of rows and columns, which are specified when referencing the element.

Setting up an Array

Before an array can be used in jBC, it must be set up using the DIM, DIMENSION or COMMON statements. The separate elements of the array can then be assigned values by reference to the variable in the normal way, or they can be read into the array from a file using the MATREAD or MATREADU statements.

Accessing Array Elements

Element of a dimensioned array can be referenced by giving its position within the array. Numbering positions always begin with 1. In the vector ADDR above, ADDR(3) holds the value Oldtown. In the matrix MULT above, the first element, 2, is referenced by variable MULT(row,column) i.e. MULT(1,1). The number 21 in the matrix is accessed by variable MULT(2,3).

Dynamic Arrays

A dynamic array is a vector in which the data corresponds to the fields of a jBASE file record.

A jBASE record is divided into fields, values, and subvalues. The delimiter used for each is displayed using ^ (carat),] (right bracket), and \ (backslash) respectively. These same delimiters are used to separate the different elements of the dynamic array, and can be obtained with the keyboard by using the control key with the character key in the jED editor.

Examples of dynamic arrays are:

1. MAIN Joe Smith^26^Male^Analyst
This is a dynamic array made up of four fields which give the name, age, sex and job of the person referenced.

2. PERS Joe Smith^26]2.4.68^Male]Married\3children
The details in the array have been extended to include values and sub-values. The second field holds two values for age and date of birth. The third field holds two values for sex and status. The status value is further split into sub-values for marital status and children.

Setting up a Dynamic Array

A dynamic array can be created by the concatenation of several variables together with the relevant field, value and sub-value delimiter characters.

A dynamic array is also obtained when a jBASE file record is read into a program using READ or READU statements. The elements of the array are automatically read from disk including the delimiters. This is because the dynamic array format is identical to the data storage format of jBASE files. (Note that databases not conforming to this structure can also be accessed in this manner by mapping the structure in the jEDI library).

Additionally, a value within a record can be read into a dynamic array variable by the READV and READVU statements.

Accessing Elements of a Dynamic Array

A dynamic array is referenced by the array name. Any of its component parts down to sub-value level can also be referenced by giving its position within the record:

DYN.ARRAY<field#, value#, sub-value#>

where field#, value# and sub-value# give the numeric position of the element for each entity.

Example

Using the above examples:

MAIN	holds	Joe Smith^26^Male^Analyst
PERS	holds	Joe Smith^26]2.4.68^Male]Married\3children
MAIN<3>	holds	Male
PERS<1>	holds	Joe Smith
PERS<2>	holds	26]2.4.68
PERS<2,1>	holds	2.4.68
PERS<3,2,1>	holds	Married
PERS<3,2,2>	holds	3children

File Handling

jBC is able to access data held in any file format with a file variable. A file variable is created when the OPEN command is executed.

Once a file is OPENed, the file variable is used by input/output statements to access file records. File records may be held in either of two formats - a dimensioned array or a dynamic array.

Record Access

DYNAMIC ARRAYS give a fast method of data access from disc as the system simply strips off the record key and places the rest of the record into the file.variable space. By specifying a single variable, a full record can quickly be read or written away.

Reading records into a DIMENSIONED ARRAY is less efficient as the system has to assign each field into a separate element of the array. Writing records to file is also less efficient as the data needs to be reassembled from the array.

File Data Manipulation

DYNAMIC ARRAYS access fields and values by scanning the array from the start. Where the record has a large number of fields, and access is needed for those towards the end of the record, this will be inefficient, as the system will scan through from the start. The same happens when inserting, deleting or adding data. The jBC compiler does however, optimise the access to dimensioned arrays and traversing them in any direction is very fast. jBASE handles dynamic arrays much more efficiently than traditional implementations of the language.

With DIMENSIONED ARRAYS, each field is accessed as an individual variable. Within jBC this means that a field will be obtained as fast whether it be at the start or end of the record. This makes it more efficient where data needs to be accessed from large records. Again the jBC compiler optimizes access to Dimensioned arrays making it far more efficient than older implementations of the language.

Guidelines

Use dynamic arrays to read or write data to or from a file. If a dynamic array record field or value is to be used several times, or needs to be string searched or manipulated, then assigning it to a variable first can improve performance. However, the optimization within the jBC compiler may actually do this for you.

Use dimensioned arrays where much work needs to be done on multiple record fields - e.g. to construct new records or to continually update many fields within large records.

Locking Mechanisms

The jBC data record locking mechanism is available to application programmers for maintaining data integrity. Any file record accessed by the application can have an update lock set, which will not allow other system users to concurrently update the same record.

Two types of locks are available.

Execution Locks

An execution lock is used to prevent several different jBC programs from simultaneously executing a defined process.

For example, a jBC archive application can set an execution lock that prevents other jBC applications from running a section of code until it is finished or the lock released.

An execution lock may be set by use of the LOCK statement, and are cleared with the UNLOCK statement or termination of the program.

Individual Record Locks

A record lock is used to prevent a file record from being updated by more than one process at the same time. This includes the jED editor as well as other jBC programs.

Record locking is an efficient way of maintaining data integrity as access is only stopped to a single record rather than to a whole file. Other users are not prevented from updating other records in the same file.

A record lock is set by using the READU, READVU or MATREADU statements, and they are released when the program terminates or by use of the WRITE, WRITEV, MATWRITE or RELEASE statements.

It is also possible to update a file record that is locked and to keep the lock set by using the WRITEU, WRITEVU or MATWRITEU statements.

Applications employing record locks can be made more efficient by making use of LOCKED clauses to continue productive processing whenever a record lock is encountered.

jBC Statements

Summary

@	function	Cursor and screen manipulation.
ABORT	statement	Program termination.
ABS	function	Returns absolute value.
ALPHA	function	Returns Boolean result for alphabetic check.
ASCII	function	EBCDIC to ASCII conversion function.
ASSIGNED	function	Returns Boolean result for variable assignment.
BITCHANGE	function	Toggles the state of the specified bit.
BITCHECK	function	Returns the current value of the specified bit.
BITLOAD	function	Assigns values in the local bit table.
BITRESET	function	Resets the value of the specified bit.
BITSET	function	Sets the value of the specified bit.
BREAK	statement	Break key manipulation. Loop Termination.
CALL	statement	Transfer of program execution to an external subroutine.
CALLC	statement	Transfer of program to an external C function.
CASE	statement	Conditional branching.
CATS	function	Concatenates the corresponding elements in two dynamic arrays.
CHAIN	statement	Transfer of process control.
CHANGE	statement	Sub-string replacement.
CHANGE	function	Sub-string replacement.
CHAR	function	Returns the ASCII character equivalent of a numeric expression.
CHARS	function	Returns the ASCII character equivalents of the numeric expressions in a dynamic array.
CHDIR	function	Changes the current directory.

CHECKSUM	function	Returns numeric checksum for the supplied expression.
CLEAR	statement	Initializes all variables to zero.
CLEARCOMMON	statement	Initializes all **unnamed** common variables to a value of zero.
CLEARDATA	statement	Clears data that has been stacked by the DATA statement.
CLEARFILE	statement	Clears all the data from a file.
CLEARSELECT	statement	Clears active select lists.
CLOSE	statement	Closes a previously opened file.
CLOSESEQ	statement	Closes a previously opened sequential file.
COL1 / COL2	functions	Position determination subsequent to use of the FIELD function.
COLLECTDATA	statement	Retrieves data passed from the PASSDATA clause of an EXECUTE statement
COMMON	statement	Declares a list of variables and matrices that can be shared among programs.
COMPARE	function	Compares two strings.
CONTINUE	statement	Used to skip code in a loop.
CONVERT	function	Converts a character string to another.
CONVERT	statement	Converts a character string to another.
COS	function	Returns the cosine of an angle.
COUNT	function	Returns the number of times that one string occurs in another.
CRT	statement	Outputs data to the terminal.
DATA	statement	Stores data for stacked input.
DATE	function	Returns the date in internal form.
DCOUNT	function	Counts the number of elements in a string separated by a specified delimiter.
DEBUG	statement	Passes control to the jBC debugger.
DECRYPT	function	Decodes an encrypted string.
DEFC	statement	Declares an external C function to the jBC compiler.

DEFFUN	statement	Declares an external jBC function to the jBC compiler.
DEL	statement	Removes a specified element from a dynamic array.
DELETE	statement	Deletes a record from a file.
DELETELIST	statement	Deletes a stored list.
DELETESEQ	statement	Deletes a sequential file.
DIMENSION	statement	Declares fixed length arrays to the compiler.
DISPLAY	statement	Outputs data to the terminal.
DOWNCASE	function	Converts all uppercase characters in an expression to lowercase.
DQUOTE	function	Encloses a value in double quotation marks.
DTX	function	Returns the hexadecimal representation of a decimal expression.
EBCDIC	function	Converts a string value from ASCII to EBCDIC.
ECHO	statement	Turns on or off the echoing of characters typed at the keyboard.
ENCRYPT	function	Encrypts strings.
END	statement	Designates the end of a program or subroutine.
ENTER	statement	Unconditionally passes control to another executable program.
EQUATE	statement	Declares a symbol equivalent to a literal, variable or simple expression.
EXECUTE	statement	Allows execution of other programs and commands.
EXIT	statement	Halts the execution of a program.
EXP	function	Returns the mathematical constant e to the specified power.
EXTRACT	function	Archaic method of extracting elements from a dynamic array.
FADD	function	Performs addition on two floating point numbers.
FDIV	function	Performs division on two floating point numbers.

FIELD	function	Returns one or more delimited fields from a string.
FIELDS	function	Returns a dynamic array of delimited fields from a dynamic array of strings.
FILELOCK	statement	Attempts to lock an entire file for exclusive use by this program. (Release 3.3.1 only)
FIND	statement	Finds the location of a specified string within a dynamic array.
FINDSTR	statement	Finds the location of a specified string within a dynamic array.
FMT	function	Formats a string to a specified pattern.
FMUL	function	Performs multiplication on two floating point numbers.
FOLD	function	Re-delimits a specified string with attribute marks.
FOOTING	statement	Defines a footing to be included at the bottom of an output page.
FORMLIST	statement	Creates an active select list from a dynamic array.
FOR	statement	Defines the start of a fixed increment loop construct.
FSUB	function	Performs subtraction on two floating point numbers.
FUNCTION	statement	Declares a user-defined function.
GET	statement	Gets input from an opened serial device.
GETCWD	function	Returns the name of the current working directory.
GETENV	function	Determines the value of the specified environment variable.
GETLIST	statement	Retrieves a previously stored list.
GOSUB	statement	Causes execution of a local subroutine.
GOTO	statement	Causes program execution to jump to the code at a specified label.
GROUP	function	Returns one or more delimited fields from a string.

GROUPSTORE	statement	Replaces one group of characters in a string with another group of characters.
HEADING	statement	Defines a heading to be included at the top of an output page.
HUSH	statement	Turns on or off the echoing of characters typed at the keyboard.
ICONV	function	Converts data in external format to internal format.
IF	statement	Allows conditional execution of statements.
IN	statement	Gets raw data from the input device.
INCLUDE	statement	Includes code from other files.
INDEX	function	Returns the position of a character or characters within another string.
INDICES	function	Returns information about a file's secondary index.
INMAT	function	Returns the number of dimensioned array elements.
INPUT	statement	Gets data from the input device.
INPUTCLEAR	statement	Clears the type-ahead buffer.
INPUTNULL	statement	Allows null input to be seen by the INPUT statement.
INS	statement	Allows the insertion of elements into a dynamic array.
INSERT	function	Allows the insertion of elements into a dynamic array.
INT	function	Truncates a numeric value to the nearest integer.
IOCTL	function	Returns file information.
LEFT	function	Extracts the first n characters from a string.
LEN	function	Returns the character length of an expression.
LN	function	Returns the value of the natural logarithm of a supplied expression.
LOCATE	statement	Finds the position of an element in a specified dimension of a dynamic array.
LOCK	statement	Sets an execution lock.

LOOP	statement	Defines the start of a loop construct.
LOWCASE	function	Converts all uppercase characters in an expression to lowercase.
LOWER	function	Lowers delimiters in a string to their next lowest value.
MAT	statement	Array element assignment.
MATBUILD	statement	Creates a dynamic array from a dimensioned array.
MATCHES	function	Allows pattern matching to be applied to an expression.
MATPARSE	statement	Assigns the elements of a dynamic array to a dimensioned array.
MATREAD	statement	Reads a record and maps the elements into a dimensioned array.
MATREADU	statement	Reads and locks a record and maps the elements into a dimensioned array.
MATWRITE	statement	Transfers the contents of a dimensioned array to a specified record on disc.
MATWRITEU	statement	Transfers the contents of a dimensioned array to a specified record on disc and preserves locks.
MOD	function	Returns the arithmetic modulo of two numeric expressions.
MSLEEP	statement	Pauses program execution for a specified number of milliseconds.
NEG	function	Returns the inverse of a value.
NOT	function	Inverts the Boolean value of an expression.
NULL	statement	Does nothing.
NUM	function	Returns Boolean true if the supplied value is numeric.
OBJEXCALLBACK	statement	Communicates with a calling OBjEX program.
OCONV	function	Converts data in internal format to external format.
ONGOSUB/ONGOTO	statements	Transfers program execution to a label based upon a calculation.
OPEN	statement	Opens a file or device to a descriptor variable.

OPENDEV	statement	Opens a file or device for sequential reading and/or writing.
OPENINDEX	statement	Opens a specific index definition for a file.
OPENPATH	statement	Opens a file given an absolute or relative path.
OPENSEQ	statement	Opens a file for sequential reading and/or writing.
OPENSER	statement	Opens a device for serial IO.
OUT	statement	Sends raw characters to the current output device.
PAGE	statement	Causes the current output device to page.
PAUSE	statement	Allows processing to be suspended until an external event occurs.
PERFORM	statement	Allows execution of other programs and commands.
PRECISION	statement	Defines the number of digits of precision to be subsequently used.
PRINT	statement	Sends data to the current output device.
PRINTER	statement	Controls the destination of output from the PRINT statement.
PRINTERR	statement	Prints standard jBASE error messages.
PROCREAD	statement	Retrieves data passed from a jCL program to a jBC program.
PROCWRITE	statement	Passes data back to the primary input buffer of a calling jCL program.
PROGRAM	statement	Documents source code.
PROMPT	statement	Defines the prompt characters used by INPUT.
PUTENV	function	Sets environment variables for the current process.
PWR	function	Raises a number to a specified power.
QUOTE	function	Encloses a value in double quotation marks.
RAISE	function	Raises delimiters in a string to their next highest value.
READ	statement	Reads a record from an opened file into a variable.

READL	statement	Reads a record and takes a read-only shared record lock.
READLIST	statement	Retrieves a previously stored list.
READNEXT	statement	Retrieves the next element in a list variable.
READNEXT	statement	Moves forward through an index.
READPREV	statement	Moves backward through an index.
READSEQ	statement	Reads data from a file opened for sequential access.
READT	statement	Reads tape devices.
READU	statement	Reads a record from an opened file into a variable respecting locks.
READV	statement	Reads a field from a record in an opened file into a variable.
READVU	statement	Reads a field from a record in an opened file into a variable respecting locks.
RECORDLOCKED	function	Returns the status of a record lock.
REGEXP	function	Pattern matching with regular expressions.
RELEASE	statement	Releases record locks.
REM	function	Returns the arithmetic modulo of two numeric expressions.
REMOVE	function	Successively extracts delimited strings from a dynamic array.
REPLACE	function	Archaic method of replacing elements in dynamic arrays.
RETURN	statement	Transfers execution to the caller of a subroutine/function or to a specific label in a program.
REWIND	statement	Issues a rewind command to attached device.
RIGHT	function	Extracts the last n characters from a string.
RND	function	Generates a random number.
RQM	statement	Pauses execution.
RTNDATA	statement	return specific data to the RTNDATA clause of another program's EXECUTE.
SADD	function	Performs string addition.
SDIV	function	Performs string division.

SELECT	statement	Creates a select list of elements in a specified variable.
SELECT	statement	Creates a select list of elements based on on a secondary index.
SELECTINDEX	statement	Creates a dynamic array of keys based on a single selection of an index key.
SEND	statement	Sends output to a device.
SENTENCE	function	Returns the command used to invoke a program and the arguments it was given.
SEQ	function	Returns the numeric ASCII value of a character.
SIN	function	Returns the sine of an angle.
SLEEP	statement	Pauses program execution for a specified number of seconds or until a specified time.
SMUL	function	Performs string multiplication.
SORT	function	Sorts a dynamic array.
SOUNDEX	function	Converts strings to their phonetic equivalents.
SPACE	function	Generates strings of spaces.
SPOOLER	function	Returns spooler information.
SQRT	function	Returns the square root of a number.
SQUOTE	function	Encloses a value in single quotation marks.
SSUB	function	Performs string subtraction.
STOP	function	Program termination.
STR	function	Performs string duplication.
SUBROUTINE	statement	Declares an external subroutine.
SUBSTRINGS	function	Returns sub-strings of elements from a dynamic array.
SUM	function	Sums elements of a dynamic array.
SYSTEM	function	Returns system information.
SWAP	function	Sub-string replacement.
TAN	function	Returns the tangent of an angle.
TIME	function	Returns the current system time.
TIMEDATE	function	Returns the current time and date.
TRANS	function	Retrieves a field from a file.

TRANSABORT	statement	Aborts and reverses the current transaction.
TRANSQUERY	function	Used to determine if currently in a transaction.
TRANSTART	statement	Marks the beginning of a transaction.
TRANSEND	statement	Marks the end of a successfully completed transaction.
TRIM	function	Removes characters from a string.
TRIMB	function	Removes trailing blanks.
TRIMF	function	Removes leading blanks.
UNASSIGNED	function	Returns Boolean result for variable assignment.
UNLOCK	statement	Releases a previously locked execution lock.
UPCASE	function	Converts all lowercase characters in an expression to uppercase.
WAKE	statement	Wakes a suspended process which has executed a pause statement.
WEOF	statement	Writes an end of file mark on an attached tape device.
WEOFSEQ	statement	Writes end of file mark on a file opened for sequential access.
WRITE	statement	Writes a record to a previously opened file.
WRITELIST	statement	Writes a list to the stored list file.
WRITESEQ	statement	Writes data to a file opened for sequential access.
WRITET	statement	Writes data to a tape device.
WRITEU	statement	Writes a record to a previously opened file preserving locks.
WRITEV	statement	Writes a record field to a previously opened file.
WRITEVU	statement	Writes a record field to a previously opened file preserving locks.
XLATE	function	Retrieves a field from a file.
XTD	function	Returns the decimal representation of a hexadecimal expression.

Output Formatting

Performs formatting of output data values for use with PRINT and CRT commands.

COMMAND SYNTAX

Variable MaskExpression

Also see: FMT

SYNTAX ELEMENTS

MaskExpression Numeric Mask Codes: [j][n][m][Z][,][c][$][Fill Character][Length]

Mask Code	Description
j	Justification R Right Justified L Left Justified T 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 (see OCONV)
n	Decimal Precision: A number from 0 to 9 that defines the decimal precision. It specifies the number of digits to be 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 to be equal to n (the decimal precision).
Z	Suppress leading zeros. **NOTE:** that fractional values which have no integer will 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	Print the literal CR after negative values.
	D	Print the literal DB after positive values.
	E	Enclose negative values in angle brackets < >
	M	Print a minus sign **after** negative values.
	N	Suppresses embedded minus sign.
$	Appends a Dollar sign to value.	
Fill Charact-er 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 may be enclosed in parentheses "( )".

EXAMPLES

Mask Expression	Source Value	Returned Value (columns) 123456789012345678901 234567890123456789012 345678901234567890
R2#10	1234.56	1234.56
L2%10	1234.56	1234.56000
R2%10	1234.56	0001234.56
L2*10	1234.56	12.34*****
R2*10	1234.56	*****12.34
R2,$#15	123456.78	$123,456.78
R2,&$#15	123456.78	$$$$$123,456.78
R2,& $#15	123456.78	$ 123,456.78
R2,C&*$#15	-123456.78	$***123,456.78CR
R((###) ###-###)	1234567890	(123) 456-7890
R((#3) #2-#4)	1234567890	(123) 456-7890
L& Text #2-#3	12345	Text 12-345
L& ((Text#2) #3)	12345	(Text12) 345
T#20	This is a test of the American Broadcasting System	This is a test of the American Broadcasting System
D4/	12260	07/25/2001

@

The @ function is used 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 which column on the screen the cursor should be moved to. **row** specifies which row (line) on the screen to position the cursor. **col** may be specified on its own, which will cause the cursor to the required column on whichever row it currently occupies.

NOTES

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

The terminal is always addressed starting from (0,0), 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 you may 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)

The @(ScreenCode) function is also used 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

jBASE has been designed 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 JBCEMULATE environment variable will 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

Emulation	**Code**	**Function**
ros	-16	turn off underline mode
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
```

ABORT

The ABORT statement terminates the program running as well as 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 use of the comma character. The expression(s) correspond to the parameters that need to be passed 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

This statement is used to terminate the execution of a jBC program together with any calling program. It will then optionally display a message, and return to the shell prompt.

The optional message displayed on terminating the program is held in the error file. For successful printing of the message, parameters such as linefeeds, clearscreen, date and literal strings may also be required.

Operation of this command can be altered by setting the Command Level Restart option - see the Systems Housekeeping chapter of the System Administrator"s Reference Manual for more details.

EXAMPLE

```
CRT "CONTINUE (Y/N) ?":; INPUT ANS
IF 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

The ABS function will return the mathematical absolute of the ()expression.

COMMAND SYNTAX

ABS(expression)

SYNTAX ELEMENTS

expression can be an expression 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

This can be expressed 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, null is returned 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)
```

This is the program output:

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

ADDS

Use the ADDS function to create a dynamic array of the element-by-element addition of two dynamic arrays.

Each element of array1 is added to the corresponding element of array2. The result is returned in the corresponding element of a new dynamic array. If an element of one array has no corresponding element in the other array, the existing element is returned. If an element of one array is the null value, null is returned 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
PRINT ADDS(A,B)
```

This is the program output:

```
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. If any character in expression is non alphabetic then the function returns FALSE (0).

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, a false (0) is returned for that element.

If both corresponding elements of array1 and array2 are null, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, a false is returned 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)
```

This is the program output:

```
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 been assigned a value.

COMMAND SYNTAX

ASSIGNED(variable)

SYNTAX ELEMENTS

ASSIGNED returns TRUE if the **variable** named has been assigned a value before the execution of this statement. If the **variable** has never been assigned a value then the function returns FALSE.

NOTES

This function has been provided as it has been implemented in older implementations of the language. It is far better to program in such a way, as this statement will not be needed.

See also UNASSIGNED.

EXAMPLES

```
IF ASSIGNED(Var1) THEN
    CRT "Var1 has been assigned a value"
END
```

BITCHANGE

BITCHANGE toggles the state of a specified bit in the local bit table, and return 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

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 returned will always be 0 (zero) or 1.

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 0 (zero), it will be set to 1 and 0 will be displayed. If bit 100 is set to 1 (one), the reverse will apply.

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 to be checked.

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 returned will always be 0 (zero) or 1.

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 0 (zero), 0 will be displayed. If bit 100 is set to 1 (one), 1 will be displayed.

BITRESET

BITRESET resets the value of a specified bit in the local bit table to 0 and returns the value of the bit before it was changed.

COMMAND SYNTAX

BITRESET(table_no)

SYNTAX ELEMENTS

table_no specifies the position in the table of the bit to be reset. If **table_no** evaluates to zero, all elements in the table are reset to 0 (zero) and the returned value is zero.

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 returned will always be 0 (zero) or 1.

BITRESET returns the value of the bit before it was changed - checking and resetting a flag can be accomplished 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 1, returns a value of 1, resets bit 112 to 0, and prints 1. If table entry 112 is 0, returns a value of 0, and prints 0.

BITSET

BITSET sets the value of a specified bit in the bit table to 1 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, all elements in the table are set to 1 (one) and the returned value is one.

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 returned will always be 0 (zero) or 1.

BITSET returns the value of the bit before it was changed - checking and setting a flag can be accomplished 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 0, returns a value of 0, sets bit 112 to 1, and prints 0. If table entry 112 is 1, returns a value of 1, and prints 1.

BREAK

The BREAK statement allows the break key to be configured.

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 more commonly known 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

As BREAK is used to terminate the innermost loop, it is ignored 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
```

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 may not use any of the built-in functions of jBC (such as COUNT), within the expression.

There is no limit to the number of parameters that may 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 to 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 jBC variable (@var). The program could pass back return values in variables. CALLC arguments can be simple variables or complex expressions, but not arrays. CALLC can be used 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:

1. Write and compile the C program.
2. Define the C program call interface.
3. Build the runtime version of jBASE (containing the linked C program).
4. 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 being permanently bound together. Linking between the separate pieces is accomplished at runtime (rather than compile time) through a DLL interface.

For CALLC, developers create a DLL and then call that DLL from jBASE. E-type VOC/MD entries for each function called from a DLL communicate interface information to 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 it is called indirectly:

```
SUB.NAME = DRAW.CIRCLE
CALLC @SUB.NAME(RADIUS,CENTER)
```

In the next example, CALLC is used as a function, assigning the return value of the subroutine PROGRAM.STATUS in the variable RESULT:

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

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 CASE structure is bounded by the **BEGIN CASE** and **END CASE** statements. Within this block, an arbitrary number of CASE **expression** statements may exist followed by any number of jBC statements. The **expression** should evaluate to a TRUE or FALSE result. At execution time, each **expression** is evaluated in order. If the **expression** returns a TRUE result, then the statements beneath it are executed. 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 1. 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".

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 variable Z is assigned the value:

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

```
A = "a" : @SVM : "b" : @VM : "c": @VM : "d"
B = "x" : @VM : "y" : @SVM : "z"
C = CATS(A, B)
```

The variable C is assigned the value:

```
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 jBC 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 jBC program).

NOTES

There are no restrictions to the CHAIN statement and you may CHAIN from anywhere to anywhere. However, you should try and make your programs follow a logical path that may be 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 jBC 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

Either string can be of any length and is not required to be the same length. The jBC 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
```

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. This is the entire range of the ASCII character set.

NOTES

jBC variables can contain any of the ASCII characters 0-255, thus there are no restrictions on this function.

This function is often used 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/jbc/src") THEN
    CRT "jBASE development system INSTALLED"
END
```

```
IF GETENV("JBCGLOBALDIR", 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

CLEAR can be used 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 that has been stacked by the DATA statement.

COMMAND SYNTAX

CLEARDATA

SYNTAX ELEMENTS

None.

CLEARFILE

The CLEARFILE statement is used 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** must have been the subject of an OPEN statement before the execution of CLEARFILE upon it. If the variable is omitted from the CLEARFILE statement, then the default file variable is assumed as per the OPEN statement.

NOTES

The CLEARFILE statement will remove every database record on the file it is executed against. It should therefore be used with great care.

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, 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

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 so the next INPUT statement forces a response from the user.

COMMAND SYNTAX

```
CLEARINPUT
```

Example

In the following example, the CLEARINPUT statement clears the terminal type-ahead buffer so the user must respond 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

The CLEARSELECT statement is used to clear active select lists.

COMMAND SYNTAX

CLEARSELECT {ListName | ListNumber}

SYNTAX ELEMENTS

ListName must evaluate to a jBC list variable. **ListNumber** is one of the numbered lists in the range 0 to 11. If neither **ListName** or **ListNumber** are specified then the default list (0) is cleared.

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

The CLOSE statement is used to CLOSE a previously opened file when it is no longer needed.

COMMAND SYNTAX

CLOSE variable{, variable ...}

SYNTAX ELEMENTS

The **variable list** is 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

It is good practice to hold open only those file descriptors that you wish to have constant access to. Although there is no limit to the number of files that may be opened within jBC, opening large numbers of files and leaving them open can consume resource that can be better used.

EXAMPLES

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

CLOSESEQ

Close 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

These functions are used in conjunction with the FIELD function to determine the character positions 1 position before and 1 position after the last field that was located.

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. They can then be used 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

The COLLECTDATA statement is used to retrieve data passed from the PASSDATA clause of an EXECUTE statement.

COMMAND SYNTAX

COLLECTDATA variable

SYNTAX ELEMENTS

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

NOTES

The COLLECTDATA statement can be used 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 JBC_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 jBC 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 will return 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.

CONTINUE

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

COMMAND SYNTAX

CONTINUE

The statement is used 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, character 1 in expression1 is converted 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 jBC program. This 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

The value returned by expression is assumed to be 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.

Also see 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")
```

This is the program output:

```
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, the THEN statements are executed.

If the record or file is not created, the ELSE statements are executed.

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 jBC 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, DISPLAY is can be used 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 it was typed in at the keyboard.

COMMAND SYNTAX

DATA expression {, expression ...}

SYNTAX ELEMENTS

The **expression** may evaluate to any data type. Each comma-separated expression will be viewed as one line of terminal input.

NOTES

The data stacked for input will subsequently be treated as input by any jBC program. Therefore it may be used before PERFORM/EXECUTE, CHAIN or any other method of transferring program execution. It may also be used to stack input for the currently executing program but cannot be used to stack input back to an executing program.

When stacked data is detected by a jBC program, 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 that will be used to count the fields.

NOTES

The delimiter string may consist of more than 1 character.

If expression1 is a NULL string, then the function will return a value of 0.

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

Also see COUNT.

EXAMPLES

```
A = "A:B:C:D"
CRT DCOUNT(A, ":")
```

will display the value 4.

DEBUG

The DEBUG statement causes the executing program to enter the jBC debugger.

COMMAND SYNTAX

DEBUG

NOTES

The debugger is described here.

EXAMPLES

```
IF FatalError = TRUE THEN
    DEBUG ;*enter the debugger
END
```

DECRYPT

The DECRYPT function decodes a string encrypted by the ENCRYPT function.

COMMAND SYNTAX

DECRYPT(string, key, method)

SYNTAX ELEMENTS

string specifies the string to be decrypted.
key is the value used to decrypt the string. It's use depends on **method**.
method is a number which indicates the decryption mechanism to use:

0	General purpose encryption scheme. This method will decrypt the string using the **key** value supplied.
1	Simple ROT13 algorithm. **key** not used.
2	XOR MOD11 algorithm. The first character of **key** is used as a seed value.

NOTES

See also ENCRYPT.

EXAMPLES

```
X = DECRYPT(X, Ekey, 0)
```

```
IF DECRYPT("rknzcyr,"",1) = "example" THEN CRT "ROT13 ok"
```

```
IF ENCRYPT("g{ehvkm","9",2) = "example" THEN CRT "XOR.MOD11 ok"
```

DEFC

The DEFC statement is used to declare an external C function to the jBC compiler and define its arguments and return types.

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. If **FuncType** is omitted then INT will be assumed. 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.

NOTES

A DEFC must be compiled for each C function before any reference is made to it or the compiler will not recognize the function name.

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

When jBC variables are passed to a C function you must utilize predefined macros to access the various data types it contains. These macros are fully documented in the manual "jBASE External Interfaces". This manual also gives examples of working C functions and documents other interfaces available to the jBC programmer.

C functions are particularly useful for increasing the performance of tight loops that perform specific functions. The jBC 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 jBC programmer may freely ignore the type of argument used when the C function is invoked as the jBC compiler will automatically perform type conversion.

EXAMPLE 1

```
DEFC INT cfunc( INT, FLOAT, VAR)
Var1 = cfunc( A, 45, B)
cfunc( 34, C, J)
```

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

EXAMPLE 2

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

DEFFUN

The DEFFUN statement is used to declare an external jBC function to the jBC compiler and optionally define its arguments. DEFFUN is used 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 that is passed to the function by the calling program. To pass an array, the keyword MAT must be used before the argument name. These parameters are entirely 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 jBC compiler. It 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 an empty string is returned by the function.

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)
```

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

EXAMPLE 2

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

DEL

The DEL statement is used 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

Non integer values for any of the expressions are truncated to integers.

Invalid numeric values for the expressions are ignored without warning.

The command operates within the scope specified, i.e. if only a field is specified then the entire field (including its multivalues and subvalues) is deleted. If a subvalue is specified, then only the subvalue is deleted 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

The DELETE statement is used 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 the default file variable is assumed.

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

If a lock was being held by the program against the file record, then the lock will be released.

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 either with the WRITELIST statement or the SAVE-LIST command from the shell.

NOTES

If POINTER-FILE is accessible then lists are saved in it. Otherwise they 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 a the variable to contain next record from sequential file.
FileVar	specifies the file descriptor of the file opened for sequential access.
Statements	conditional jBC statements

DELETEU

Use the DELETEU statement to delete a record without releasing the update record lock set by a previous READU statement (see READ statements).

The file must have been previously opened with an OPEN statement. If a file variable was specified in the OPEN statement, it can be used in the DELETEU statement. You must place a comma between the file variable and the record ID expression. If no file variable is specified in the DELETEU statement, the statement applies to the default file. See the OPEN statement for a description of the default file.

DIMENSION

The DIM statement is used to declare arrays to the compiler before they are referenced.

COMMAND SYNTAX

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

SYNTAX ELEMENTS

The variable may be any valid variable name that has not already been used or declared. The numbers define the size of each dimension and must be either constants or the subject of an EQUATE statement.

A number of arrays may be declared by a single DIM statement by separating their declarations with a comma.

NOTES

The array must be declared before it is referenced in the program source (compilation as opposed to execution). The compiler will display an error message if a variable is used as a dimensioned array before it has been declared.

The array variable may not be used as a normal variable or dynamic array before being dimensioned and the compiler will detect this as an error.

A dimension size may not be specified as 1 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)
```

DIV

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

COMMAND SYNTAX

```
DIV (dividend, divisor)
```

The dividend and divisor expressions can evaluate to any numeric value. The only exception is that divisor cannot be 0. If either dividend or divisor evaluates to null, null is returned.

Example

```
I=400; K=200
J = DIV (I,K)
PRINT J
```

This is the program output:

```
2
```

DIVS

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)
```

Each element of array1 is divided by the corresponding element of array2 with the result being returned in the corresponding element of a new dynamic array. If elements of array1 have no corresponding elements in array2, array2 is padded with ones and the array1 elements are returned. If an element of array2 has no corresponding element in array1, 0 is returned. If an element of array2 is 0, a run-time error message is printed and a 0 is returned. If either element of a corresponding pair is null, null is returned.

Example

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

This is the program output:

```
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)

SYNTAX ELEMENTS

expression in a string containing some alphabetic characters.

NOTES

Non-alphabetic characters are ignored.

The function LOWCASE is synonymous with DOWNCASE.

See also the output conversion codes for changing case.

DROUND

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 then the result is rounded 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.

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

The statement may be used 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. If TRUE then echoing will be turned on, and if FALSE, echoing will be turned off.

NOTES

The SYSTEM function can be used to determine the current state of character echoing. If echoing is enabled then SYSTEM(24) will return Boolean TRUE and if disabled it will return Boolean FALSE.

EXAMPLES

```
ECHO OFF
CRT "Enter your password ":
INPUT Password
ECHO ON
.....
```

This will disable character input echoing while a password is typed in.

ENCRYPT

The ENCRYPT function encrypts strings.

COMMAND SYNTAX

ENCRYPT(string, key, method)

SYNTAX ELEMENTS

string specifies the string to be encrypted.
key is the value used to encrypt the string. It's use depends on **method**.
method is a number which indicates the encryption mechanism to use:

0	General purpose encryption scheme. This method will encrypt the string using the **key** value supplied.
1	Simple ROT13 algorithm. **key** not used.
2	XOR MOD11 algorithm. The first character of **key** is used as a seed value.

NOTES

See also DECRYPT.

EXAMPLES

```
X = ENCRYPT("What's your point Vanessa?", Ekey, 0)

IF ENCRYPT("example,"",1) = "rknzcyr" THEN CRT "ROT13 ok"

IF ENCRYPT("example,"9",2) = "g{ehvkm" THEN CRT "XOR.MOD11 ok"
```

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 to be executed. 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 jBC COMMON data area can be passed to another jBC program by specifying the I option after the program name - see the examples. The COMMON data area can only be passed to another jBC program.

ENTER can be used 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 to be executed (the target program) is another jBC 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)

Each element of array1 is compared with the corresponding element of array2. If the two elements are equal, a 1 is returned in the corresponding element of a dynamic array. If the two elements are not equal, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, a 0 is returned. If either element of a corresponding pair is null, null is returned for that element.

Example

```
A=1:@VM:45:@SM:3:@VM:"one"
B=0:@VM:45:@VM:1
PRINT EQS(A,B)
```

This is the program output:

```
0]1\0]0
```

EQUATE

EQUATE is used 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 has to be computed for each access during run time, the address of a symbol is always known. This means that the processing overhead involved in accessing a particular value can be significantly reduced. See the example for a more detailed explanation of the other benefits.

Readability can be enhanced 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 will 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 expression with another substring. If you do not specify occurrence, each occurrence of substring is replaced.

COMMAND SYNTAX

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

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 1, it defaults to 1. If substring is an empty string, replacement is prefixed to expression. If replacement is an empty string, all occurrences of substring are removed. If expression evaluates to null, null is returned. 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","")
```

This is the program output:

```
AAAZZZCCCDDDZZZ
ZZZAAABBBCCCDDDBBB
AAACCCDDD
```

EXECUTE

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

COMMAND SYNTAX

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

Data, Dynamic Arrays and lists can be passed to programs written in jBC. Screen output and error messages can be intercepted 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. The command is executed by a new Bourne Shell (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 they are used. Any variable name may be used 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. Every newline normally sent to the terminal is replaced with a field mark in the variable.

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 program executed sets up a list then the RTNLIST clause may be used to place that list into a specified variable. If the variable is omitted then the list is placed in the default list variable.

PASSDATA variable
The data in the specified variable is passed to another jBC program. The executing jBC program should retrieve the data using the COLLECTDATA statement.

RTNDATA variable
The RTNDATA statement returns any data passed from an executing jBC program in the specified variable. The executing jBC 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 is used to halt the execution of a program and return a numeric exit code to the parent process. For compatibility with older versions of the language the EXIT statement may be used 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 must 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 then the program will enter the debugger with 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 e to the specified power.

COMMAND SYNTAX

EXP(expression)

SYNTAX ELEMENTS

The **expression** may consist of any form of jBC expression but should evaluate to a numeric argument or a runtime error will occur and the program will enter the debugger.

NOTES

The function will return a value that is accurate to as many decimal places as are 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".

FIELD

The FIELD function will return 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) are 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, 1 is assumed.

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, 1 is assumed.

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 3 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
0	1 if file.variable is a valid file variable; 0 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. file.variable specifies an open file. If file.variable is not 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 also 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 a FILELOCK has been executed, as will READU , READL , WRITE , CLEARFILE and so on.

The lock continues until either 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. It would be advisable to replace such with more judicious use of item locks.

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 will always use the NT locking mechanism.

The use 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 like this :

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

or like this when the file is originally created:

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

If the file continues to use jBASE record locking then the ON ERROR clause will be taken and the SYSTEM(0) and STATUS() functions will be 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 a fatal error is encountered during processing of the FILELOCK statement. If a fatal error occurs, and the ON ERROR clause was not 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 to compare every element of the dynamic array with. **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 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 is used to locate 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 to search every element of the dynamic array with. **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

The FLUSH statement causes all the buffers for a sequential I/O file to be written immediately. Normally, sequential I/O uses buffering for input/output operations, and writes are not always flushed immediately.

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, the THEN statements are executed, and the ELSE statements are ignored.

If the file cannot be written to or does not exist, the ELSE statements are executed.

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

Performs formatting of output data values for use with PRINT and CRT commands.

COMMAND SYNTAX

FMT(Variable, MaskExpression)

Also see: Output Formatting

SYNTAX ELEMENTS

MaskExpression Numeric Mask Codes: [j][n][m][Z][,][c][$][Fill Character][Length]

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 (see OCONV)
n	Decimal Precision: A number from 0 to 9 that defines the decimal precision. It specifies the number of digits to be 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 to be equal to n (the decimal precision).
Z	Suppress leading zeros. **NOTE:** that fractional values which have no integer will 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

Mask Code	Description	
	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	Print the literal CR after negative values.
	D	Print the literal DB after positive values.
	E	Enclose negative values in angle brackets < >
	M	Print 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 may be enclosed in parentheses "( )".

EXAMPLES

Format Expression	Source Value (X)	Returned Value (columns) (V) 12345678901234567890123456789012345678901234567890
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  This is a test of
                                   the American       the American
                                   Broadcasting       Broadcasting System
                                   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 acted upon independently and is returned as an element in a new dynamic array.

COMMAND SYNTAX

FMTS (dynamic.array, format)

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 the FMT function.

If dynamic.array evaluates to null, null is returned. If format evaluates to null, the FMTS function fails and the program enters the debugger.

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. Spaces are converted to attribute marks except when in enclosed in sub-strings. Extraneous spaces are removed.

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 causes all subsequent output to the terminal to be halted at the end of each output page. The statement allows an expression to be evaluated and displayed at the foot of each page. If output is currently being sent to the terminal, the output will be paused until a carriage return is entered 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 that will be printed at the bottom of every page of output. The string may contain a number of special characters that are interpreted and replaced in the string before it is printed. 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 output is being directed to the printer (a PRINTER ON) statement is in force then output sent to the terminal with the CRT statement is not paged. If output is being sent to the terminal then all output is paged.

EXAMPLE

```
FOOTING "Programming staff by weight Page "P"
```

FOR

The FOR statement allows the programming of looping constructs within the program. The loop is controlled by a counting variable and may 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 every iteration of the loop **var** is automatically incremented by 1. **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 a **WHILE** or **UNTIL** clause (not both), which will be evaluated 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. Expressions 3 and 4 should then be replaced with these variables. This can offer large performance increases where complex expressions are being used.

See also: BREAK, CONTINUE.

EXAMPLES

```
Max =DCOUNT(BigVar, CHAR(254))
FOR I = 1 TO Max STEP 2 WHILE BigVar LT 25
   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.

FUNCTION

Identifies a user-defined function which can be invoked by other jBC 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

The FUNCTION statement is used 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 as 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 the RETURN statement is used without a value then by default an empty string is returned.

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)
```

Each element of array1 is compared with the corresponding element of array2. If the element from array1 is greater 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 less than the element from array2, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either element of a corresponding pair is null, null is returned 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 not 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. It is assumed that if this is desired the application - or device drive - will handle it.

If the length and timeout expressions are not specified, 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 that either a THEN or ELSE clause, or both, be specified. If the data is received without error, the THEN clause is executed. If the data cannot be received (or a timeout occurs), the ELSE clause is executed.

See also: GETX

GETCWD

The GETCWD function allows a jBC program to determine the current working directory of the program. This will normally be the directory from which the program was executed but may have been changed with 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 indicating whether the command was successful or not.

NOTES

Refer to your UNIX or Windows documentation (sh is a good place to start), 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 jBC 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.

NOTES

See the UNIX documentation for the Bourne shell (sh) or the Windows on-line help for information on environment variables. Click here for information regarding environment variables unique to the jBASE system.

See also 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 jBC 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 jBC GETUSERGROUP function returns the group number for the user ID specified by uid. For Windows NT or Windows 2000, it returns 0.

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 GET 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 **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 not 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 the length and timeout expressions are not specified, 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 GETX syntax requires that either a THEN or ELSE clause, or both, be specified. If the data is received without error, the THEN clause is executed. If the data cannot be received (or a timeout occurs), the ELSE clause is executed.

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. This label identifies the start of a local subroutine. Label formats are defined earlier in this chapter.

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

The use of the GOTO command is not recommended as it obscures the readability of code and therefore is a hindrance to maintainability. All programs written using the GOTO construct can be written using structured statements such as LOOP and FOR. Opinions on this are divided but the consensus is that GOTO should be avoided.

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

The HEADING statement causes all subsequent output to the terminal to be halted at the end of each page. The statement allows an expression to be evaluated and displayed at the top of each page. If output is currently being sent to the terminal, output is paused until a carriage return is entered at the terminal - unless the N option is specified.

COMMAND SYNTAX

HEADING expression

SYNTAX ELEMENTS

The **expression** should evaluate to a string that will be printed at the top of every page of output. The string may contain a number of special characters that are interpreted and replaced in the string before it is printed. 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 output is being directed to the printer (a PRINTER ON statement is in force), output sent to the terminal with the CRT statement will not be paged. If output is being sent to the terminal, all output will be paged unless you specify the **N** option.

EXAMPLES

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

HEADINGE and HEADINGN

The HEADINGE statement is the same as the HEADING statement, however causes a page eject with the HEADING statement.

The HEADINGN statement is the same as the HEADING statement, however the page eject is suppressed.

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 that the conversion is to be performed upon. **expression2** should evaluate to the conversion code that is to be performed against the data.

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.

Also see OCONV.

EXAMPLES

```
InternalDate = ICONV("27 MAY 1997", "D")
```

In this example ICONV returns the internal form of the date May 27, 1997.

ICONVS

Use the ICONVS function to convert each element of dynamic.array to a specified internal storage format.COMMAND SYNTAXICONVS (dynamic.array, conversion)

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, null is returned. If an element of dynamic.array is null, null is returned 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 the OCONVS function.

0	The conversion is successful.
1	An element of dynamic.array is invalid. An empty string is returned, unless dynamic.array is null, in which case null is returned.
2	conversion is invalid.
3	Successful conversion of possibly invalid data.

IF

The IF statement is used to allow other statements to be conditionally executed.

COMMAND SYNTAX

IF expression THEN|ELSE statements

SYNTAX ELEMENTS

The expression will be evaluated to a value of Boolean TRUE or FALSE. If the expression is TRUE then the statements defined by the **THEN** clause will be executed (if present). If the expression is FALSE then the statements defined by the **ELSE** clause are executed.

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
```

but the clauses may be expanded 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
```

The single and multi-line versions of either clause may be combined 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)
```

Each element of dynamic.array is evaluated. If the element evaluates to true, the corresponding element from true.array is returned to the same element of a new dynamic array. If the element evaluates to false, the corresponding element from false.array is returned. If there is no corresponding element in the correct response array, an empty string is returned 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 specifies which occurrence of **expression2** should be searched for within expression1.

NOTES

If the specified occurrence of expression2 cannot be found in expression1 then 0 is returned.

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

The INMAT() function, used 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.

When the INMAT() function is used with the 'array' argument, it returns the current number of elements to which 'array' was dimensioned.

NOTES

In some dialects the INMAT() function is also used to return the modulo of a file after the execution of an OPEN statement. This is inconsistent with its primary purpose and has not been 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 CUSTREC array 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 exactly the same as for 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 will hold the characters that were input until the time-out.

The **FOR** clause requires either or both of the **THEN** and **ELSE** clauses. 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. The INPUTNULL statement should be used to define a character that indicates a NULL input.

The CONTROL-CHARS command can be used 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.

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 expressions 1 through 3 will cause the element to appended as the last Field, Value or Sub-Value rather than at a specific position. Only one of the expressions may be negative otherwise the first negative value is used correctly but 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

INSERT is the function form of the INS statement, which should be used in preference to this function.

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.

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. The I-descriptor must have been compiled before the ITYPE function uses it, otherwise you get a run-time error message.

i.type can be set to the I-descriptor to be evaluated 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: The @FILENAME should be set to the name of the file before ITYPE execution.

Example

This is the SLIPPER file contents:

```
JIM    GREG     ALAN

001 8  001 10   001 5
```

This is the DICT SLIPPER contents:

```
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
```

This is the program output:

```
3 records selected
JIM WEARS SLIPPERS SIZE 8
GREG WEARS SLIPPERS SIZE 10
ALAN WEARS SLIPPERS SIZE 5
```

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.

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 characters that are extracted. 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"
""
```

LEFT

Use the LEFT function to extract a substring comprising the first n characters of a string, without specifying the starting character position.

COMMAND SYNTAX

```
LEFT (string, n)
```

LEFT is equivalent to the following substring extraction operation:

```
string [ 1, length ]
```

If string evaluates to null, null is returned. If n evaluates to null, the LEFT function fails and the program terminates with a run-time error message.

Example

```
PRINT LEFT("ABCDEFGH",3)
```

This is the program output:

```
ABC
```

LEN

The LEN function returns the character length of the supplied expression.

COMMAND SYNTAX

LEN(expression)

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 dyamic.array must be a string value. The characters in each element of dynamic.array are counted, and the counts are returned.

The LENS function includes all blank spaces, including trailing blanks, in the calculation.

If dynamic.array evaluates to a null string, 0 is returned. If any element of dynamic.array is null, 0 is returned for that element.

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)

Each element of array1 is compared 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, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either of a corresponding pair of elements is null, null is returned for that element. If you use the subroutine syntax, the resulting dynamic array is returned 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 natural logarithm is calculated using the mathematical constant e as a number base.

EXAMPLES

```
A = LN(22/7)
```

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, it was found. 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 the statements defined by the **THEN** clause are executed. If **expression1** is not found in an element of the dynamic array the statements defined by the **ELSE** clause are executed.

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 jBC 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 the LOCK is already held by another program and an **ELSE** clause was 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 lock has been released by the other program.

NOTES

See also UNLOCK.

If the program was compiled with the environment variable JBCEMULATE set to r83, 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 which are lowered as follows:

ASCII Character	Lowered To
255	254
254	253
253	252
252	251
251	250
250	249
249	248

EXAMPLE

```
ValuemarkDelimitedVariable = LOWER(AttributeDelimitedVariable)
```

MAT

The MAT command is used 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

Note that 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

The MATBUILD statement is used 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 jBC 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 start 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 allows pattern matching to be applied to an expression.

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 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

Pattern matching strings are constructed from the rule table shown below. When reading the table n refers to any integer number.

Pattern	**Explanation**
n**N**	this construct matches a sequence of n digits
n**A**	this construct matches a sequence of n alpha characters
n**C**	this construct matches a sequence of n alpha characters or digits
n**X**	this construct matches a sequence of any characters
"string"	This construct matches the character sequence *string* exactly.

The pattern will be applied to all characters in expression1 and it must match all characters in the expression to evaluate as Boolean TRUE.

The integer value n can be specified 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

MATPARSE

The MATPARSE statement is used 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 jBC 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, each array element in the dynamic array is assumed to be separated 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.

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 start 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.

It is our opinion that MATBUILD and MATPARSE have been named the wrong way round, but imagine the confusion if we had changed them!

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 jBC 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 the record is found and can be read from the file then it is mapped into array and the **THEN** statements are executed (if any). If the record cannot be read from the file for some reason then array is unchanged and the **ELSE** statements (if any) are executed.

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 dependant on it are executed. If no **LOCKED** clause was specified then the statement blocks (hangs) until the lock is released by the other process. 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, 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 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 jBC variable that has previously been opened to a file using the OPEN statement. If **variable1** is not specified then the default file is assumed (see OPEN statement). The **expression** should evaluate to a valid record key for the file.

If the record is found and can be read from the file then it is mapped into array and the **THEN** statements are executed (if any). If the record cannot be read from the file for some reason then array is unchanged and the **ELSE** statements (if any) are executed.

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 dependant on it are executed. If no **LOCKED** clause was specified then the statement blocks (hangs) until the lock is released by the other process.

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>
```

For more detailed information on record locking, see the article The Keys to Record Locking.

EXAMPLES

```
MATREADU Xref FROM CFile, "XREF" ELSE MAT Xref = ''
MATREADU Ind FROM IFile, "INDEX" LOCKED
   GOSUB InformUserLock ;* 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 indeed an array and has been dimensioned 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 an array and has been dimensioned before its use in the statement.

EXAMPLES

```
DIM A(8)
MAT A = 99
....
MATWRITEU A ON "NewArray"
```

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 modulo is calculated as the remainder of expression1 divided by expression2. 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)
```

This is the program output:

```
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.

If no parameter is supplied then a default time period of 1 millisecond is assumed.

NOTES

If the debugger is invoked while a program is sleeping and then execution **c**ontinued, 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

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)
```

This is the program output:

```
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 behaviour 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, the THEN statements are executed otherwise the ELSE statements are executed. 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 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, null is returned for that element.

Example

```
X=5; Y=5
PRINT NOTS(X-Y:@VM:X+Y)
```

This is the program output:

```
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

The NUM function is used to test arguments for numeric values.

COMMAND SYNTAX

NUM(expression)

SYNTAX ELEMENTS

expression may evaluate to any data type.

NOTES

If every character in expression is found to be 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 jBC 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 a value of 1 is returned to the corresponding element in a new dynamic array. If the element is a nonnumeric string, it evaluates to false, and a value of 0 is returned.

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, null is returned. If an element of dynamic.array is null, null is returned for that element.

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 passed back 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 jBC 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 jBC program to ensure that the proper clean up operations are eventually made.The two parameters can be used to pass data between the jBC and OBjEX environments in both directions. They are defined as Variants in the OBjEX environment and as normal variables in the jBC 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

The OCONV statement is used 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 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 jBC language or supplied by the user (see external interfaces documentation).

NOTES

OCONV will return the result of the conversion of expression1 by expression2. Valid conversion codes are shown below:

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.

Conversion	**Action**
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
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. If the character "c" is supplied then this character is used instead of the period.
MCPN{n}	In the same manner as the MCP conversion all non printable characters are replaced. However the replacing character is followed by the ASCII hexadecimal value of the character that was replaced.
MCNP{n}	Performs the opposite conversion to MCPN. The ASCII hexadecimal value following the tilde character is converted back 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.

Conversion	**Action**
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)

The elements are converted 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, null is returned. If any element of dynamic.array is null, null is returned 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	An invalid element is passed to the OCONVS function; the original element is returned. If the invalid element is null, null is returned for that element.
2	The conversion code is invalid.

For information about converting elements in a dynamic array to an internal format, see the ICONVS function.

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

The value of expression is used 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 the index is found to be 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

The OPEN statement is used to open a file or device to a descriptor variable within jBC.

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 has been installed into the jBASE system. If the file has a dictionary section and this is to be opened by the statement then this may be specified 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 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 OPEN statement uses the environment variable JEDIFILEPATH to search for the file named. If this is not defined then the current working directory is looked in followed by the home directory of the current process. See the documentation on environment variables for more details.

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.

There is no limit to the number of files that may be opened by a jBC program.

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 jBC statements

NOTES

If the device does not exist or cannot be opened then the **ELSE** clause is executed. Once open a lock is taken 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. Locks are only taken on regular file types. 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 info on sequential processing, see READSEQ, WRITESEQ, or the sequential processing example.

OPENPATH

The OPENPATH statement is used to open a file (given an absolute or relative path) to a descriptor variable within jBC. 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 the OPENPATH 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:

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 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.

There is no limit to the number of files that may be opened by a jBC program.

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 jBC statements

NOTES

If the file does not exist or cannot be opened then the **ELSE** clause is executed. However if JBCEMULATE 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 **READONLY** is specified then the process takes a read lock on the file otherwise a write lock is taken. 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 which 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
jBASE 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

Serial IO to the com ports on NT and to device files on Unix can be achieved using the sequential file statements. In addition certain control operations can be performed using the IOCTL function.

Serial IO can be handled via the OPENSEQ statement however the OPENSER statement has also been provided.

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, a false is assumed for the missing element.

If both corresponding elements of array1 and array2 are null, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, null is returned. If one element is the null value and the other is any value other than 0 or an empty string, a true is returned.

Example

```
A="A":@SM:0:@VM:4:@SM:1
B=0:@SM:1-1:@VM:2
PRINT ORS(A,B)
```

This is the program output:

```
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.

Value	Description
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.

Value	Description
0	The file is closed 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.

Value	**Description**
0	The file was deleted.
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, and 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 jBC 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 jBC 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 a runtime error message is generated. On large files, use OSBREAD or READSEQ.

jBASE uses the ASCII 0 character (CHAR(0)) as a string-end delimiter. ASCII 0 cannot be used in any string variable within jBC. 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 a WAKE statement is executed 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 EXECUTE and PERFORM.

PERFORM

PERFORM is synonymous with PCPERFORM and EXECUTE.

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 either be 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).

See also: CRT

EXAMPLES

```
PRINT A "L#5"
PRINT @(8,20):"Patrick":
```

PRINTER

The PRINTER statement is used to control the destination of output from the PRINT statement.

COMMAND SYNTAX

PRINTER ON|OFF|CLOSE

NOTES

PRINTER ON will cause all subsequent output from the PRINT statement to be redirected to the print spooler.

PRINTER OFF will cause all subsequent output from the PRINT statement to be redirected to the terminal device.

PRINTER CLOSE will act as **PRINTER OFF** but in addition will close 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

Used 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**.

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

Used to retrieve data passed to programs from a jCL program.

COMMAND SYNTAX

PROCREAD variable THEN|ELSE statements

SYNTAX ELEMENTS

variable is a valid jBC identifier, which will be used to store the contents of the primary input buffer of the last jCL program called.

If the program was not initiated by a jCL program then the PROCREAD will fail and any statements associated with an **ELSE** clause will be executed. 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 jBC programs.

EXAMPLE

```
PROCREAD Primary ELSE
   CRT "Unable to read the jCL buffer"
   STOP
END
```

PROCWRITE

Used 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

Performs no function other than to document the source code.

COMMAND SYNTAX

PROGRAM progname

SYNTAX ELEMENTS

Progname can be any string of characters.

NOTES

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

Used 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("JBCLOGNAME=":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 JBC 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 jBC 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 assign 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 byte read.

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
```

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 jBC 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. What this basically means is 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 hopefully 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 dependant on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the lock is released by the other process. 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 record is written to by the same program with WRITE, WRITEV or MATWRITE statements.
- The record is deleted by the same program with the DELETE statement.
- 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, DELETE

For more detailed information on record locking, see the article The Keys to Record Locking.

READLIST

READLIST allows the program to retrieve a previously stored list (perhaps created with the SAVE-LIST command), into a jBC 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 that the element refers to. 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
```

READSELECT

See 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 jBC 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 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:

Value	**Meaning**
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 channel specified then 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 jBC 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 dependant on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the lock is released by the other process. 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 READU statement will be released by any of the following events:

- The record is written to by the same program with WRITE, WRITEV or MATWRITE statements.
- The record is deleted by the same program with the DELETE statement.
- 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, DELETE

For more detailed information on record locking, see the article The Keys to Record Locking.

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 jBC 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 value returned in **variable1** is controlled by the readv0 emulation setting. 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 you should 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 jBC 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 value returned in **variable1** is controlled by the readv0 emulation setting. 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 dependant on it are executed. If no LOCKED clause was specified then the statement blocks (hangs) until the lock is released by the other process.

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 record is written to by the same program with WRITE, WRITEV, MATWRITE or DELETE statements.
- 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, DELETE

For more detailed information on record locking, see the article The Keys to Record Locking.

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
```

RECORDLOCKED

The RECORDLOCKED function can be called 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
```

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 the file described by the default file variable is released.

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 a dangerous - especially in subroutines.

For more detailed information on record locking, see the article The Keys to Record Locking.

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

This 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 it will be 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 jBC compiler will optimise 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 (see ABS). The highest number **expression** can be on Windows is PWR(2,15) - 1. The highest number on unix is PWR(2,31) - 1.

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 jBC 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 jBC program executes another jBC 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

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

The SDIV function performs string division of two base 10 string numbers. The result is rounded 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

The SDIV 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 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 the 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
```

This is the program output:

```
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 the default file variable is assumed.

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.

If the **SETTING** clause is specified and the select fails, setvar will be set 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 a list is being built from record keys in a file, the list is not created immediately by scanning the file for all the record keys. Instead, only the first few keys will be extracted. When these keys have been taken from the list, the next few keys will be obtained and so on. This means that the list could contain records that are written to the file after the SELECT command is started.

Consider the situation where you open a file, SELECT it and then, on the basis 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 a variable is used to hold the select list, then it should be unassigned or null prior to the SELECT. If it contains a number in the range 0 to 10 then the corresponding **select list number** will be used to hold the list, although you can still reference the list with the variable name. This "feature" is for compatibility with older platforms. See 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 that either a THEN or ELSE clause, or both, be specified. If the data is send without error, the THEN clause is executed. If the data cannot be sent, the ELSE clause is executed.

See also: SENDX

EXAMPLES

See 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 the device has already been opened with OPENSER or OPENSEQ.

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 that either a THEN or ELSE clause, or both, be specified. If the data is send without error, the THEN clause is executed. If the data cannot be sent, the ELSE clause is executed.

See also: SEND

EXAMPLES

See Sequential File Processing Examples.

SENTENCE

The SENTENCE function allows a program to find out 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

The command line arguments are assumed to be space separated and when the entire command line is returned 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)

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, null is returned. If any element of dynamic.array is null, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as return.array.

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")
```

This is the program output:

```
84]71
71
```

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

Allows the program to pause execution for a specified time 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 time 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 the host clock is changed in the meantime then the program will not wake up at the desired time.

If the debugger is invoked while a program is sleeping and then execution **c**ontinued, 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

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

The SMUL function can be used 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

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"
```

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 data bases

SPACE

The SPACE function is a convenient way to generate 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 however as this is inefficient and should be accomplished 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)
```

This is the program output:

```
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

MultiValue	Description
4	Port number job was generated on
5	Creation date in internal format
6	Creation time in internal format
7	Job Status
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	JBCLOGNAME 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

MultiValue	**Description**
8	Real user id
9	JBCLOGNAME 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

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 form 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
```

This is the program output:

```
0001
```

SSELECTN

See SSELECT.

SSELECTV

See SSELECT.

SSUB

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

The SSUB 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 SSUB.

EXAMPLE

```
A = 2.3000000123456789
B = 5.0000000000000001
CRT SSUB(A,B)
```

Displays -2.6999999876543212 to the screen.

STOP

The STOP statement is virtually identical in function to the ABORT statement except that a calling jCL program will not be terminated.

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, null is returned. 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)
```

This is the program output:

```
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 jBC 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 statement 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)
```

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.

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 the 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)	Return the number of characters available in input buffer. Invoking SYSTEM(14) can cause in 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 JBCLOGNAME. If the system_19_timedate emulation option is set then the number of seconds since midnight December 31, 1967 is returned.
SYSTEM(20)	Return last spooler file number created
SYSTEM(21)	Return port number or JBCPORTNO
SYSTEM(22)	Reserved

SYSTEM(23)	Return status of the break key Enabled 0 Enabled 1 Disabled by BASIC 2 Disabled by Command 3 Disabled by Command and BASIC
SYSTEM(24)	Return 1 if echo enabled, 0 if echo disabled
SYSTEM(25)	Return 1 if background process
SYSTEM(26)	Return current prompt character
SYSTEM(27)	Return 1 if executed by PROC
SYSTEM(28)	Reserved.
SYSTEM(29)	Reserved.
SYSTEM(30)	Return 1 if paging is in effect (HEADING statement)
SYSTEM(31)	Reserved
SYSTEM(32)	Reserved
SYSTEM(33)	Reserved
SYSTEM(34)	Reserved
SYSTEM(35)	Return language in use as a name or number (ROS)
SYSTEM(36)	Reserved
SYSTEM(37)	Return thousands separator
SYSTEM(38)	Return decimal separator
SYSTEM(39)	Return money symbol
SYSTEM(40)	Return program name
SYSTEM(41)	Return release number
SYSTEM(42)	Reserved
SYSTEM(43)	Return port number of item lock
SYSTEM(44)	Return 99 for jBASE system type
SYSTEM(45)	Reserved
SYSTEM(46)	Reserved
SYSTEM(47)	Return 1 if currently in a transaction
SYSTEM(48)	Reserved
SYSTEM(49)	Return PLID environment variable
SYSTEM(50)	Return login user id
SYSTEM(51)	Reserved
SYSTEM(52)	Return system node name
SYSTEM(53)	Reserved

SYSTEM(100)	Return program create information
SYSTEM(101)	Return port number or JBCPORTNO
SYSTEM(102)	Reserved
SYSTEM(1000)	Return command line separated by attribute marks
SYSTEM(1001)	Return command line and options
SYSTEM(1002)	Return temporary scratch file name
SYSTEM(1003)	Return terminfo Binary definitions
SYSTEM(1004)	Return terminfo Integer definitions
SYSTEM(1005)	Return terminfo String definitions
SYSTEM(1006)	Reserved
SYSTEM(1007)	Return system time
SYSTEM(1008)	Return SYSTEM file path
SYSTEM(1009)	Return MD file path
SYSTEM(1010)	Return Print Report information
SYSTEM(1011)	Return jBASE release directory path. JBCRELEASEDIR
SYSTEM(1012)	Return jBASE global directory path. JBCGLOBALDIR
SYSTEM(1013)	Return 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)	Return relative PROC level
SYSTEM(1015)	Return effective user name. LOGNAME
SYSTEM(1016)	Return tape assignment information
SYSTEM(1017)	Return platform. UNIX, WINNT or WIN95
SYSTEM(1018)	Return configured processors
SYSTEM(1019)	Return system information (uname -a)
SYSTEM(1020)	Return 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(2003)	Return PID of root process

Entries above 2000 are for system use only.

SUBS

Use the SUBS function to create a dynamic array of the element-by-element subtraction of two dynamic arrays.

COMMAND SYNTAX

```
SUBS (array1, array2)
```

Each element of array2 is subtracted from the corresponding element of array1 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, the missing element is evaluated as 0. If either of a corresponding pair of elements is null, null is returned for that element.

EXAMPLE

```
A=2:@VM:4:@VM:6:@SM:18
B=1:@VM:2:@VM:3:@VM:9
PRINT SUBS(A,B)
```

This is the program output:

```
1]2]3\18]-9
```

SUBSTRINGS

Use the SUBSTRINGS function to create a dynamic array each of whose elements are substrings of the corresponding elements of dynamic.array.

COMMAND SYNTAX

SUBSTRINGS (dynamic.array, start, length)

start indicates the position of the first character of each element to be included in the substring. If start is 0 or a negative number, the starting position is assumed to be 1. If start is greater than the number of characters in the element, an empty string is returned.

length specifies the total length of the substring. If length is 0 or a negative number, an empty string is returned. If the sum of start and length is larger than the element, the substring ends with the last character of the element.

If an element of dynamic.array is null, null is returned for that element. If start or length evaluates to null, the SUBSTRINGS function fails and the program enters the debugger.

Example

```
A="ABCDEF":@VM:"GH":@SM:"IJK"
PRINT SUBSTRINGS(A,3,2)
```

This is the program output:

```
CD]SK
```

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. The result is truncated 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

The time is returned 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()
```

TIMEOUT

Use the TIMEOUT statement to terminate a READSEQ or READBLK statement if no data is read in the specified time.

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 or READBLK statement.

TIMEOUT causes subsequent READSEQ and READBLK statement to terminate and execute their 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
```

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

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 = TRANS("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
```

and you 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).

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 = TRANS("DICT VENDORS",VENDOR.ID,-1,"C")
PRINT DICT.REC
```

will display

```
A]1]Vendor Name]]]]]L]30
```

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, null is returned. If any element of dynamic.array is null, null is returned 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
```

UNLOCK

The UNLOCK statement releases a previously LOCKed execution lock.

COMMAND SYNTAX

UNLOCK {expression}

SYNTAX ELEMENTS

If **expression** is specified it should evaluate to the number of a held execution lock, which will then be released.
If **expression** is omitted then all execution locks held by the current program will be released.

NOTES

No action is taken if the program attempts release an execution lock that it had not taken.

See also LOCK.

EXAMPLE

```
LOCK 23 ; LOCK 32
......
UNLOCK
```

UDTEXECUTE

See EXECUTE.

UPCASE

See DOWNCASE/UPCASE.

WAKE

The WAKE statement is used to wake a suspended process, which has executed a PAUSE statement.

COMMAND SYNTAX

WAKE PortNumber

SYNTAX ELEMENTS

PortNumber is a reference to the target port to be awakened. 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. Default value is 0.

NOTES

If the WEOF fails then the statements associated with any ELSE clause will be executed. SYSTEM(0) will return the reason for the failure as follows:

Value	Meaning
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.

If no tape device has been assigned to the channel specified then the jBASE debugger is entered 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.
Statement s	conditional jBC statements

NOTES

WEOFSEQ forces the file to be truncated at the current file pointer. Nothing is actually 'written' to the sequential file.

EXAMPLES

See 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 jBC 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 a lock was being held on the record by this process it will be 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 cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

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.

WRITELIST

WRITELIST allows the program to store a list held in a jBC 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, the list will be written to the default external list.

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

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 jBC 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 Sequential File Examples

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 which tape channel to read data from. 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:

Value	Meaning
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 channel specified then 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 jBC 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 are executed.

The lock maintained by the WRITEU statement will be released by any of the following events:

- the record is written to by the same program with WRITE, WRITEV or MATWRITE statements.
- 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 jBC 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, **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 WRITEV statement will cause any lock held on the record by this program to be released. If you wish to retain a lock on the record you should 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
```

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 jBC 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, **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 are executed.

The lock taken by the WRITEVU statement will be released by any of the following events:

- The record is written to by the same program with WRITE, WRITEV or MATWRITE statements.
- The record lock is released explicitly using the RELEASE statement.
- 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 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, 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#, the entire record will be returned.

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
```

and you 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
```

XTD

The XTD() function converts hexadecimal number 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)
```

jCL

This section shows how to write and execute jCL (jBASE Command Language) programs. It also discusses how jCL manipulates data in the various buffers and select registers.

Most of the following text assumes that you will be using the functionally superior PQN variant of the jCL language. The PQ and PQN Differences section discusses the differences between the two variants.

JCL PROGRAM STRUCTURE

A jCL program is stored as a text record in a file.

The first line of a jCL program is usually PQ or PQN, unless you have elected to run the program as a UNIX executable (script), in which case the first line will contain #!usr/jbc/bin/jpq.

Subsequent lines contain jCL statements that can execute other programs, manipulate data in the buffers, control the flow of program execution and so on.

jCL program statements comprise an numeric label (optional), a command and any command arguments.

There are many facilities which enable you to control program flow and to call (and return) or jump to other jCL programs. You can also construct internal or external subroutines.

LABELS

Labels appear at the start of a line and are always numeric (1, 330, 1000, etc.). You should always put at least one space between the label and the statement.

GROUPING COMMANDS

You can place more than one command on a line by separating each command with a subvalue mark character (x"FC" - entered by typing <ctrl \>). The commands will be executed sequentially, left to right.

Some commands cannot be grouped on the same line because they require one or more dedicated lines in the program. These commands are:

( ), [ ], B, BO, F, F;

FB, F-CLEAR, F-FREE, F-KLOSE, F-OPEN, F-READ

F-UREAD, F-WRITE, GOSUB, H, IBH, IH

MVA, MVD, O, P, RI, RO

RSUB, RTN, U, X,

Lines which are a continuation of a T or L command from a previous line are also not suitable for grouping.

The M (Mark) command can be followed by other commands on the same line, but it must be the first command on a line. Commands following the M command, on the same line must be separated by a subvalue mark rather than a space (unlike numeric labels).

COMMENT LINES

Command line which start with a "C" or an "*" are treated as comments.

If you include a comment in a multiple command line, anything between the "C" or "*" and a following a subvalue mark (or the end of the line) will be ignored.

A label will remain active if the "C" or "*" is placed after it on the line.

EXECUTING JCL PROGRAMS

jCL programs can be executed in several ways:

you can the enter the name of the program from jSHELL,
you can "jump to" another jCL program of the same type by using the ( ) command,
you can "call" another jCL program of the same type, as a subroutine, by using the [ ] command,
you can use a PERFORM, EXECUTE or CHAIN statement from a jBC program, or
you can convert the program to a UNIX executable and call it from any shell. Change the first line to #!usr/jbc/bin/jpq and then use chmod to create an executable file.

Once you start a jCL program, it will remain active until:

- control is explicitly passed to another jCL program
- the jCL program is explicitly exited
- all of the lines of the jCL program are exhausted
- a fatal error is encountered.

Even when the jCL program temporarily passes control to another process such as jED or jBC, it will still remain in control (unless control is passed to a jBC program which then CHAINs or ENTERs another jCL program). Exiting from the called process will return control to the jCL program.

If you do not want to store the main body your jCL program in the MD file, you can create a "pointer" jCL program in the MD. For example, to run a jCL program called DAILY which is held a file called REPORTS, create an MD entry like this:

```
DAILYYREPORT
001 PQN
002 (REPORTS DAILY)
```

This will chain to jCL program DAILY in file REPORTS.

Note that the "pointer" program and the "pointed to" program can have the same name.

LOGIN JCL PROGRAMS

You can create jCL programs which will be run automatically, each time a user logs in or logs in to a specific account. Simply create a jCL program with the same name as the account or user, and put "jshell -" on the last line of the login script (usually ".profile").

This feature can be used for example, to configure the jSHELL environment, implement a security system, or display an initial menu.

For example, let's assume that all users who login to an account called SALES must be offered an initial menu, which is a jBC program, called MENU held in a file called PROGS. You would create a jCL program in the MD of the SALES account and call it SALES like this:

```
SALES
001 PQN
002 HRUN PROGS MENU
003 P
```

Each time a user logs in to the SALES account, the system will search for a program called SALES in the MD of the account. The jCL program will then run the MENU program from the PROGS file.

INTERNAL DATA STORAGE

jCL programs use a series of buffers to store and manipulate data internally. jCL uses these buffers in much the same way as variables in a jBC program.

The four main buffers are classed as "input" and "output", and sub-classed as "primary" and "secondary".

Nine file buffers are provided to allow data to be moved to and from file records. There is also a "fast" file buffer to give you quick access to records.

In addition to the main input/output and file buffers, there is a range of five "select" buffers which are used to manipulate lists (typically record keys).

You will need a thorough understanding of the way in which these buffers work if you want to write (or maintain) anything other than very simple jCL programs.

The four jCL main buffers are:

1. primary input buffer (PIB)
2. secondary input buffer (SIB)
3. primary output buffer (POB)
4. secondary output buffer (SOB)

Typically, the input buffers are used to receive and manipulate data, and the output buffers are used to issue shell commands to the system.

Most of the time you will use the primary buffers as the active buffers. The secondary buffers (particularly the secondary input buffer) are used by very few jCL commands.

PRIMARY INPUT BUFFER

The primary input buffer is a work area. It can be used to collect terminal input or for temporary storage of control parameters.

Individual parameters within the PIB are referenced by preceding the parameter number with a percent sign (for example, %3).

When you start a jCL program, the PIB will contain the jCL program name and any supplied arguments, exactly as they were entered.

When you execute a jCL program from a jBC CHAIN, EXECUTE/PERFORM statement, the PIB will contain the CHAIN, EXECUTE or PERFORM command.

If you execute a jBC program from jCL, you can access the contents of the PIB from jBC with the PROCREAD and PROCWRITE statements. Control returns to the jCL program when the jBC program ends - unless control is passed to a jBC program which then CHAINs or ENTERs another jCL program.

SECONDARY INPUT BUFFER

The secondary input buffer has three main roles:

collection of input from the terminal when an IN command is used,

collection of error codes generated from executed programs and jQL queries (see the IF E command), and
collection of spooler hold file numbers in the form Entry # n, where n is the hold file number.

Data in the SIB is overwritten each time you use one of these facilities.

When the primary input buffer is active, you can use the MS command to copy the entire contents of the secondary input buffer to the primary input buffer.

PRIMARY OUTPUT BUFFER

The primary output buffer is used to build shell commands, which are submitted for processing when the P command is executed. With the exception of other jCL programs and the LOGTO command, you can construct and execute any shell command in this way.

You should always use a jCL transfer function when invoking one jCL program from another.

A carriage return is automatically placed at the end of the command in the POB when it is executed by the P command.

SECONDARY OUTPUT BUFFER

The secondary output buffer is used to provide a "stack" of responses to be fed into interactive processes, such as jBC or jED, when they request input.

The SOB becomes active when you use the STON (stack on) command. You can then place data in the SOB before you execute say, a jBC program.

The stacked data is used to respond to INPUT requests, instead of taking input from the keyboard - rather like the DATA statement in a jBC program.

Note that the same internal storage area is used for both the SOB and the jBC DATA statement. If your jCL program is called from a CHAIN, EXECUTE or PERFORM statement in a jBC program, any data placed in the stack by a DATA statement will be used by jCL input statements.

If a jCL program is executed directly from the shell, the SOB will be cleared before the program starts.

Each line of data placed in the SOB using the H command must be terminated by typing a less-than character (<) to represent a carriage return.

Older systems required you to use a line continuation symbol (two less - than characters "<<") if you wanted to put more than 140 characters into the buffer. Although this functionality is still supported, you do not need to use the line continuation symbol in jBASE.

ACTIVE INPUT BUFFER

Many commands access the currently active input buffer rather than referring specifically to the primary or secondary areas. These include the A and Ap forms of the A command, and the B, D (without parameter), F, IBH, IH, IP and Sp commands.

When an IN or IBN command is executed, when the SIB will become the active input buffer. The SIB will then remain active until an RI, S (n) or MV %n source command is executed.

Be careful when using these commands. Try to structure your programs in such a way as to minimise the possible confusion between which buffer is currently active. As a minimum, include as many comments as you can.

ACTIVE OUTPUT BUFFER

When a jCL program is executed from the shell or you issue a STOFF (Stack Off) command, the POB becomes the active output buffer.

The SOB only becomes active when you use a STON (Stack On) command.

You can refer to the parameters in the active output buffer by using a hash sign (#).

Some operations, such as COPY and EDIT, need two buffers and you will need to switch between the buffers when creating these commands. The Primary buffer holds the command (as it would be entered at the shell prompt) and the secondary buffer holds any further input required by the command. If you fail to specify any required input in the secondary buffer, it will be prompted for at run time

BUFFER DATA AND POINTERS

Data in the input and output buffers is held as a series of parameters separated by blanks (PQ) or field marks (PQN).

You can refer directly to a specific parameter by using the parameter's sequence number. For example, you might refer to the 10th parameter as %10.

Alternatively, you can refer to the data in terms of its column (or character) position. For example, you might refer to the 21st column (character) of the primary input buffer as S(21).

Four buffer pointers are used:

- one for each of the active input buffers
- one for each of the output buffers

In the examples which follow, the buffer pointers are indicated by an up arrow like this a.

Field marks (the parameter separators in PQN-style jCL programs) are indicated by a carat like this ^.

The input buffer pointer points to a position in the active input buffer. You can move the pointer without affecting the data in the buffer by using the S, F and B commands. The S command positions the input buffer pointer to a specific location, while the F and B commands move the pointer forward or backward over the parameters.

Each output buffer has its own pointer. These pointers always indicate the end of the parameter data. If you move the pointer backwards and write data to the new position, the original data at the end of the buffer will be truncated.

The BO command is used to move the primary output buffer pointer back one parameter, or to clear the secondary output buffer.

FILE BUFFERS

jCL provides nine file buffers, numbered from 1 to 9. The file buffers are used as the medium for transferring data from and to files - reading and writing.

You should always open file buffers before you use them for reading or writing. They can also be used as temporary data storage areas.

The data held in a file buffer (like a database record) is divided into fields which are separated by field marks. If you refer to a field beyond the last field, the required number of additional fields (with null values) are created.

File buffer references are constructed by specifying an ampersand (&), followed by the file buffer number, a period (.) and then a numeric value (which might be an indirect reference) to define the field number you want.

For example, if file buffer 4 contains:

```
000 TEST1
001 ABC
002 DEF
003 GHI
004 JKL
```

&4.2 refers to the second field in file buffer 4 - in this case DEF, and &4.0 refers to the record key of the record in file buffer 4 - in this case TEST1.

The record key will be null until you perform a read operation or until you move data into field 0 of the file buffer.

The FB command provides a special "Fast Buffer" facility. You can use FB to read a record from a file into the fast buffer without having to open the file first. There is only one fast buffer. The fields are referred to as &n, where n is the field number. Any existing data is overwritten each time you use the FB command.

SELECT REGISTERS

Select registers are used to hold a list of record keys or other multi-valued fields, which have been generated by shell commands that produce lists. These include BSELECT, ESEARCH, FORM-LIST, GET-LIST, QSELECT, SEARCH, SELECT and SSELECT. Having executed one of these commands, you should use the PQ-SELECT and PQ-RESELECT jCL commands to load or reset the select registers.

Five select registers (numbered from 1 to 5) are provided. Each can contain a separate list.

The select registers are accessed to by using an exclamation mark (!) followed by the register number - for example !2. Each time a select register is accessed, the next field is taken from the top of the list.

You cannot manipulate the values in a select register. If you write to a select register, say with the MV command, the whole list will be replaced. If you want to use a value more than once, move it to another buffer for temporary storage.
For example, if the Select Register contains

```
KEY1^KEY2^KEY3^KEY4^...
```

The first access will retrieve KEY1, the second KEY2, the third KEY3 and so on.

BUFFER REFERENCES

Buffer reference symbols can be used to make direct or indirect references to the input and output buffers and the file buffers. You can only make direct references to the select registers.

Four special symbols are used:

% for the primary input buffer.
for the active output buffer.
& for a file buffer or the "fast" buffer.
! for a select register.

Referencing the primary input buffer or the active output buffer will not reposition the pointer.

Use the appropriate symbol with a literal value to use direct reference. For example:

% 23 Refers to the 23rd parameter in the primary input buffer.

#4 Refers to the 4th parameter in the current output buffer.

&4.19 Refers to field 19 in file buffer 4.

!3 Refers to the next value in select register 3.

Use a combination of symbols and literal values to create an indirect reference. For example:

% %3 Refers to the %3th parameter in the primary input buffer. If say, %3 contained the value 10, this example would refer to the 10th parameter of the primary input buffer.
&4.%5 Refers to the field in file buffer 4 defined by %5.

Indirect references can only be made to the primary input buffer (%), the active output buffer (#) or a file buffer (&). If the resulting value is non-numeric, a zero is used. In the case of the file buffer (&), you may only use an indirect reference to the parameter, not the file buffer number - for example, &5.%3 is legal, &%3.5 is not.

Direct or indirect buffer references can be used with the following jCL commands:

( ), F-OPEN, IBH, IFN, MVA
[ ], F-READ, IBP, IH, MVD
F;, F-UREAD, IF, IP, RI
F-FREE, FB, IF (multivalued), L, S
F-CLOSE, H, IF (mask), MV, T

BRANCHING

Wherever possible, avoid using the obsolete "M"ark command to control branching within the program. Using labels and the GO commands will help to preserve clarity and make program maintenance much easier.

GROUPING COMMANDS

You can put more than one jCL command on the same line by using a subvalue mark. For example:

```
001 PQN
002 10T "Enter file name :",+
003 IBP %1
004 F-O 1 %1
005 T "Cannot open ", %1, "..."\ GO 10
006 T "File ", %1, " opened OK"
```

In this example, if the file cannot be opened by the F-O command, the line immediately following the command will be executed (see the F-O command for details). If the file is opened, the next but one line will be executed. By grouping an error message and a branch on the "error" line (005), you will avoid the necessity of creating a separate subroutine to handle the error condition.

Note that you cannot use grouping like this where a command relies on a particular line structure - the F-O command for example.

You can use a mark with the ctrl \> but it must be the first jCL command on the line. For example, use:

```
M \ IBP:%1 \ IF # %1 X
```

not:

```
IBP %1 \ M \ IF # %1 X
```

READABILITY

To increase readability and make it easier to edit and debug your jCL program, you can indent lines by using leading blanks.

The incorporation of useful comments, using the C or * commands, will help with the future maintenance of your programs.

TIME AND DATE

The following commands provide a simple way of putting the current system time and date into the currently active input buffer:

```
S21
U147F
T
```

Puts the current time, in internal format (seconds since midnight), in the current input buffer (in this case, %21).

```
S10
U147F
D
```

Puts the current date, in internal format (days since 31 Dec 67), in the current input buffer (in this case, %10).

```
S8
U147F
T MTS
```

Puts the current time, in external format (MTS yields hh:mm:ss), in the current input buffer (in this case, %8).

```
S33
U147F
D D2/
```

Puts the current date, in external format (D2/ yields dd/mm/yy or mm/dd/yy - depending on your internationalization settings), in the current input buffer (in this case, %33).

VALIDATING DATE AND TIME

You can use pattern matching to input a valid date or time but it does not catch input like 10/32/94 or 25:25:25.

The example below checks for a valid date in D2/ format by converting it. This mechanism works because an invalid date converts to null.

```
001 PQN
002 10 T "Enter date (mm/dd/yy) :", +
003 IF # %1 XFinished
004 IBP %1
005 MV %2 "
006 IBH%1;D2/;
007 IF # %2 T B,"Oops!"\ GO 10
008 C Date OK
```

LONG STATEMENTS

To help with program clarity, you can construct long statements by using several H commands. Make sure there is a space at the end the first H command or before the second (and subsequent) commands. For example:

```
001 PQN
002 HGET-LIST listname
```

```
003 STON
004 P
005 HSORT filename WITH ...
006 H HEADING "..."
007 P
```

Older systems required you to use a line continuation symbol (two less- than characters "<<" the buffer. Although this functionality is still supported, you do not need to use the line continuation symbol in jBASE.

CONCATENATION

Use an asterisk (*) to concatenate (join) a series of values. For example:

```
001 PQN
002 MV %2 "string"
003 MV %1 "Text "*%2*" has been concatenated"
004 T %1
```

will display "Text string has been concatenated".

SPOOLER HOLD FILE NUMBERS

If a hold file is generated while a jCL program is executing, the hold file number is returned as an error message number in the secondary input buffer.

Hold file numbers are returned as Entry #n, where "n" is the hold file number, so that you can distinguish them from "real" error message numbers.

JCL COMMANDS

This section begins with a brief summary of the jCL commands, organized by function. Each jCL command is then described, in alphabetical order.

INPUT BUFFER COMMANDS

B	Moves the buffer pointer back to the previous parameter.
F	Moves the buffer pointer forward to the next parameter.
IBH	Inserts a text string containing embedded blanks into the active input buffer.
IH	Inserts a text string, creates a new null parameter, or nulls an existing parameter in the active input buffer.
RI	Clears all or part of the primary input buffer, and can be used to clear the secondary input buffer.

S	Moves the input buffer pointer to a specified parameter or column.

OUTPUT BUFFER COMMANDS

BO	Moves the active output buffer pointer back one parameter.
H	Inserts a literal into the active output buffer.
RO	Clears both output buffers and selects the primary as active.
STOFF	Selects the primary as the active output buffer.
STON	Selects the secondary (stack) as the active output buffer.

DATA MOVEMENT COMMANDS

A	Copies a parameter from the active input buffer to the active output buffer.
MS	Moves the secondary input buffer contents to the primary input buffer.
MV	Copies data between primary input buffer, active output buffer, file buffers and select registers.
MVA	Copies the specified source into the destination buffer and stores it as a multivalue.
MVD	deletes data from a multivalued parameter in the destination buffer.

INPUT/OUTPUT BUFFER OPERATIONS

IBN	Accepts input from the terminal as a single parameter with all blanks intact and places it in the secondary input buffer.
IBP	Accepts input from the terminal as a single parameter with all blanks intact and places it in the specified buffer or the active input buffer.
IN	Accepts input from the terminal and places it in the secondary input buffer.
IP	Accepts input from the terminal and places it in the specified buffer or the active input buffer.
IT	Transfers a tape record to the primary input buffer.

JUMP AND BRANCH OPERATIONS

( )	Terminates the current jCL program and begins execution of (chains to) another jCL program.
[ ]	Calls an external jCL program routine.
G, GO, GOTO	Transfers control to the jCL program statement with the specified label.
GO B	Transfers control backward to the previous M (mark) command and continues execution from that point.
GO F	Transfers control forward to the next M (mark) command and continues execution from that point.
GOSUB	Transfers control to the local subroutine with the specified label.
M	Marks a location to which a GO F or a GO B command transfers control.
RSUB	terminates execution of the local subroutine and returns control to the statement following the GOSUB that called the subroutine.
RTN	Returns control from an external jCL program subroutine to the jCL program that called the subroutine.

CONDITIONAL OPERATIONS

IF	Allows conditional execution of jCL program commands.
IF E	Tests for presence of an error condition after processing a shell command.
IFN	Performs numeric comparisons and allows conditional execution of jCL program commands.

FILE OPERATIONS

F-CLEAR, F-C	Clears the specified file buffer.
F-DELETE, F-D	Deletes an record from a file opened by an F-OPEN command.
F-FREE, F-F	Releases an record lock set by the F-UREAD command.
F-KLOSE, F-K	Closes the specified file buffer.
F-OPEN, F-O	Clears and opens a file buffer to allow reads and writes.

F-READ, F-R	Reads a record from a file into a file buffer.
F-UREAD, F-UR	Reads a record from a file into a file buffer and locks the record.
F-WRITE, F-W	Writes the contents of the specified file buffer to a file.
FB	Reads an record into a special "fast buffer" without first opening the file.

ARITHMETIC OPERATORS

+	Adds an integer to the current parameter in the active input buffer.
-	Subtracts an integer from the current parameter in the active input buffer.
F;	Performs arithmetic functions on constants and buffer parameters.

PROCESSING

P	Executes the shell command in the primary output buffer.
PU	Executes the UNIX command in the primary output buffer, using the UNIX shell.

TERMINAL AND PRINTER OUTPUT

L	Formats output to the printer.
O	Outputs a text string to the terminal.
T	Produces complex, formatted terminal output and display buffer values.

DEBUGGING

C or *	Includes a comment in a jCL program.
D	Displays all or part of the active input buffer.
DEBUG	Turns debug on or off.
TR	Invokes a trace for a jCL program and displays each command on the terminal before it is executed.
PP	Displays the command in the output buffer and prompts to continue.
PW	Displays the command in the output buffer and prompts to continue.

EXITING

( )	Terminates the current jCL program and begins execution of another jCL program.
X	Halts execution of a jCL program and returns control to the shell.

Terminates the current jCL program and begins execution of (chains to) another jCL program.

COMMAND SYNTAX

({DICT} file-name{, data-section-name} {key}) {label}

SYNTAX ELEMENTS

DICT specifies the dictionary section of **file-name**, if required.

file-name is the name of the file that contains the jCL program to be executed. Can be a literal, or a direct or indirect reference to a buffer or select register.

data-section-name specifies an alternative data section of the file. Can be a literal, or a direct or indirect reference to a buffer or select register.

key is the name of the jCL program to be executed. Can be a literal, or a direct or indirect reference to a buffer or select register. If key is not specified, the current parameter in the active input buffer will be used.

label specifies a label in the target jCL program from which to start execution.

NOTES

The ( ) command terminates the current jCL program and begins execution of another jCL program, of the same type.

The input buffers, output buffers, and file buffers are all passed to the second program, and all open files stay open.

The ( ) command is often used in the MD to "point" to another jCL program which contains the main body of code. See example 1.

The ( ) command can also be used to divide large jCL programs into smaller units, minimizing the search time for labels.

Using the ( ) command (or the [ ] command), will ensure that the contents of all buffers are available to the target program. If this is not a consideration, you can execute another jCL program with the P command (see later).

EXAMPLE 1

```
MENU
001 PQN
002 (JCL MENU2)
```

Immediately executes another jCL program called MENU2 in the file called JCL.

EXAMPLE 2

```
MENU
001 PQN
002 (JCLFILE)
```

When the word MENU is entered from the shell, it will be placed in parameter 1 of the primary input buffer - %1. The program will then unconditionally pass control to the MENU program in file JCLFILE.

EXAMPLE 3

```
DOIT
001 PQ
002 IP?
003 (JCL)
```

This example will prompt for the name of another jCL program and continue execution with the named jCL program in the file called JCL.

EXAMPLE 4

```
MENU
001 PQN
002 (JCL MENU2) 300
```

Immediately executes another jCL program called MENU2 in the file called JCL. Execution of MENU2 will begin at label 300.[ ]

Calls another jCL program as an external subroutine.

COMMAND SYNTAX

```
[{DICT} file-name{, data-section-name} {key}] {label}
```

SYNTAX ELEMENTS

DICT specifies the dictionary level of file-name, if required.

file-name is the name of the file that contains the jCL program subroutine. Can be a literal, or a direct or indirect reference to a buffer or select register.

data-section-name specifies an alternative data section of the file (default is the same name as the dictionary). Can be a literal, or a direct or indirect reference to a buffer or select register.

key is the name of the jCL program to be executed. Can be a literal, or a direct or indirect reference to a buffer or select register. If **key** is not specified, the current parameter in the active input buffer will be used.

label specifies a label in the target jCL program from which to start execution. Use of the **label** clause makes this command synonymous with the GOSUB command.

NOTES

Input buffers, output buffers, and file buffers are all passed through to the called program, and all open files stay open.

External subroutines can call other subroutines. There is no limit to the number of calls that you can make but the jCL programs must be of the same type.

When an RTN is encountered, control is returned to the calling jCL program. If an RTN is not encountered, execution will terminate at the end of the called program.

EXAMPLE

```
001 PQN
002 [SUBS SUB1]
003 ...
```

Calls the jCL program SUB1 in the SUBS file as an external subroutine.

+

Adds an integer value to the current parameter in the active input buffer.

COMMAND SYNTAX

+n

SYNTAX ELEMENTS

n is the integer to be added.

NOTES

If the number of characters in a parameter increases as a result of the + command, the other parameters will be moved to the right.

If a parameter is preceded by a + sign, the sign will be replaced by a zero.

If the buffer pointer is at the end of the buffer, a new parameter will be created.

If the referenced parameter is non-numeric, a zero is used.

EXAMPLE 1

Command PIB Before PIB After

```
+30 AAA^+20^333 AAA^050^333
a a
```

EXAMPLE 2

Command PIB Before PIB After

```
+100 BBB^AA^44 BBB^100^44
a a
```

EXAMPLE 3

Command PIB Before PIB After

```
+51 ABC^0000^55 ABC^0051^55
a a
```

-

Subtracts an integer from the current parameter in the active input buffer.

COMMAND SYNTAX

-n

SYNTAX ELEMENTS

n is the integer to be subtracted.

NOTES

If the number of characters within a parameter decreases with a - command, the result is prefixed with zeros to maintain the same number of characters as the original value.

Parameters within the input buffer can be preceded by a minus sign. If the buffer pointer is at the end of the buffer, a new parameter will be created. If the referenced parameter is non-numeric, a zero is used.

EXAMPLE 1

Command PIB Before PIB After

-300 AAA^345^666 AAA^045^666
a a

EXAMPLE 2

Command PIB Before PIB After

-20 AAA^ABC^666 AAA^-20^666

a a

EXAMPLE 3

Command PIB Before PIB After

```
-50 AAA^-50^666 AAA^-100^666
a a
```

EXAMPLE 4

```
001 PQN
002 OEnter a number+
003 S5
004 IBP
005 +7
006 T %5
007 -3
008 T %5
009 RTN
```

This example receives input from the terminal and places it in the 5th parameter of the primary input buffer. It adds 7 to the value stored in the 5th parameter and displays the result. It then subtracts 3 from the result and displays the new value.

jCL Statements

A	Copies a parameter from the active input buffer to the active output buffer.
B	Moves the active input buffer pointer back to the previous parameter.
BO	Moves the active output buffer pointer back by one parameter.
C	Defines a comment.
D	Displays parameters from the active input buffer.
DE	Displays the current value of LastError.
DEBUG	Turns the jCL debug function on or off.
F	Moves the active input buffer pointer forward to the next parameter.
F;	Provides a range of arithmetic functions.
F-CLEAR	Clears the specified buffer.
F-DELETE	Deletes a record from a file.
F-FREE	Releases a record lock.
F-KLOSE	Closes a specified file buffer.
F-OPEN	Opens a file for reading and writing.
F-READ	Reads a record from an open file into a file buffer.
F-UREAD	Reads and locks a record from an open file into a file buffer.
F-WRITE	Writes the contents of a file buffer as a record.
FB	Reads a record from a file into the special "fast" buffer without having to open the file first.
G/GO/GOTO	Transfers control unconditionally to another location in the program.
GO B	Transfers control to the statement following the most recent mark command executed.
GO F	Transfers control to the statement containing the next mark command.
GOSUB	Transfers control to a specific subroutine.
H	Places a text string in the active output buffer.
IBH	Places text in the active input buffer whilst retaining embedded spaces.
IBN	Prompts for input and places the entered data in the secondary input buffer.

	the secondary input buffer.
IBP	Prompts for input from the terminal.
IF	Allows conditional execution of jCL commands based on the evaluation of an expression.
IF E	Conditionally executes a command depending on the presence or absence of an error message.
IF S	Conditionally executes a command depending on the presence or absence of an active select list.
IFN	Allows conditional execution of commands depending on the result of numeric comparisons.
IH	Places a text string in the active input buffer.
IN	Prompts for input and places it in the secondary input buffer.
IP	Prompts for input and places it into the active input buffer or a nominated buffer.
IT	Reads a tape record into the primary input buffer.
L	Formats printed output.
M	Marks a destination for a GO F or GO B command
MS	Move the entire content of the secondary input buffer to the primary input buffer.
MV	Copies data between buffers or between buffers and select registers.
MVA	Copies a value from the source to the destination buffer and stores it as a multivalue.
MVD	Deletes a value from a multivalued parameter in the target buffer.
O	Outputs a text string to the terminal.
P	Submits the shell command created in the primary output buffer for execution.
PQ-RESELECT	Executed from a jCL program, resets the pointer of a select register to the beginning of the list of keys.
PQ-SELECT	Executed from a jCL program, loads a list of keys into a select register
RI	Resets (clears) the primary and secondary input buffers.
RO	Resets (clears) the active output buffer.
RSUB	Terminates execution of a local subroutine.
RTN	Terminates execution of an external subroutine.

S	Positions the primary input buffer pointer to a specified parameter or column.
STOFF	Selects the primary output buffer as the active output buffer.
STON	Selects the secondary output buffer as the active output buffer.
T	Produces formatted terminal output.
TR	Traces jCL program execution and displays each command before it is executed.
U	Executes a user exit from a jCL program.
X	Halts execution of the program and returns control to the shell.

PARAGRAPHS

Paragraphs can only be invoked automatically via the jshell, jsh, when executing in jsh mode. The jshell, in jsh mode, will attempt to read the first parameter of an entered command from the MD/VOC file as previously specified by the JEDIFILENAME_MD environment variable. If the record is determined to be a paragraph record then the paragraph interpreter, para, is invoked. A logon paragraph record can also be invoked by the jshell via the dash option, alternatively the paragraph record can be invoked from the command line using the paragraph interpreter.

e.g. para ./VOC/PARAGRAPHX

Format
The first line of a paragraph record begins with PA e.g.
PARAGRAPHX
001 PA

This line can then be followed by any of the following :
Comment
Any Line preceded by an asterisk. e.g:
* THIS IS A COMMENT LINE
Label
Any non spaced text suffixed by a colon. e.g:
Label:

Paragraph Prompt
A paragraph prompt to obtain values, e.g:
<<I2,Second,1N>>

Load parameter two from the command line into PromptId Second. If parameter two on the command line is null then prompt for value for Second ensure validates to 1 numeric. The paragraph prompt is formatted as follows:

<<{Code,}PromptId{,Mask}>>

Code

A	Always Prompt.
Cn	Use parameter n from command line. If value does not exist then null is used.
In	Use parameter n from command line. If value does not exist then prompt.
Sn	Use parameter n from original command line. If value does not exist then prompt.
P	Use input for all PromptIds of the same name. This is the default action.
R{s}Prompt	use s as separator, until null input. Default separator is spa e

	Default separator is space.
F(File, RecordId {,Attr {,Value {,Subvalue}}})	Input from record RecordId, in file File.
@(c,r)	Prompt at column c, row r.
@(BELL)	Sound BELL at prompt.
@(CLR)	Clear screen before prompt.
@(TOF)	Prompt at top left of screen.

PromptId Identifier used to name prompt values.

Mask Validate Input
Pattern Match
e.g. 0N - Match any numerics
7X - Match seven printable characters.
3A - Match three alpha characters.
text - Match text
(Conversion)
e.g. (D2/) - Match date dd/mm/yy or mm/dd/yy.

~~Mask means input should NOT match mask.

Note: A Paragraph prompt not mixed with a command line or paragraph statement will attempt to execute the resultant value.

Paragraph Statement

One of the following syntax statements.

DISPLAY text	Output text
DATA Input	Stacked input
GO Label	Continue at Label
LOOP	Start Loop
REPEAT	Repeat Loop
CLEARPROMPTS	Clear Prompt Identifier Values

IF Expression THEN Statement

where:
Expression - Includes PromptId, operators and/or @variables

@DATE	Current internal date
@TIME	Current internal time
@DAY	Current two digit day of month
@MONTH	Current two digit month of year

@YEAR	Current two digit year	
@LOGNAME	Environment PLID value	SYSTEM(49)
@USERNO	Current port number	SYSTEM(18)
@WHO	Current user name	SYSTEM(19)
@ABORT.CODE	Last error code	SYSTEM(17)
@SYSTEM.RETURN.CODE	Last error code	SYSTEM(17)
@USER.RETURN.CODE	TBD	

Operators include MATCHES, EQ, LE, LT, etc.

Command
Any executable command with or without paragraph prompts. e.g:

```
SSELECT<<I2,FILEX,10X>>
```

or

```
SELECT FILEX
```

EXAMPLES

```
PARAGRAPHX
001 PA
002 IF <<C3,THIRD>> EQ THEN GO CHK.SECOND
003 DISPLAY GOT 3 ARGUMENTS
004 GO END
004 CHK.SECOND:
005 IF <<C2,SECOND>> EQ THEN GO END
006 DISPLAY GOT 2 ARGUMENTS
007 END:
```

```
PARAGRAPHX
001 PA
002 * This is an example paragraph
003 IF<<C2,COMMAND>> NE THEN GO DO.CMD
004 DISPLAY ENTER
005 CLEARPROMPTS
006 DO.CMD:
007 DISPLAY ENTER
```

PQ AND PQN DIFFERENCES

The first line of a jCL program defines the program as one of two basic types, a PQ or a PQN style program.

Wherever possible, you should use the PQN style.

There are several differences between the two styles, as outlined in the following topics.

DELIMITERS

PQ-style programs usually use a blank as the delimiter between parameters in the buffers. PQN-style programs usually use a field mark.

PQN allows parameters to be null or contain blanks and more closely mirrors the native record structure.

PQ commands are still supported for compatibility but you should use the functionally superior PQN commands in new or revised jCL programs.

When pointers are moved, PQ commands will generally leave the pointer at the first character of a parameter. PQN commands will generally leave the pointer at a field mark.

Commands affected by this difference are A, B, BO, F, H, IH and IP.

BUFFERS

Buffer referencing, file buffers and select registers are only available with PQN commands.

COMMANDS

These commands are only used in PQN-style jCL programs:

F;, F-KLOSE, F-WRITE, L, MVD F-CLEAR, F-OPEN, FB, MS,
F-DELETE, F-READ, IBH, MV,
F-FREE, F-UREAD, IBP, MVA,

These commands are functionally equivalent in both PQ and PQN-style programs:

( ), G, IF E, RSUB, U
[], GO B, IFN, RTN, X
+, GO F, M, S,
-, GOSUB, P, STOFF,
C, IF, RI, STON,

JCL PQ-SELECT

Executed from a jCL program, loads a list of keys into a select register.

COMMAND SYNTAX

PQ-SELECT register-number

SYNTAX ELEMENTS

register number is the number of the select register (1 to 5) in which the keys are to be placed.

NOTES

To use PQ-SELECT you must first construct a list by using one of the list processing commands such as SELECT, SSELECT, QSELECT, BSELECT, GET-LIST, FORM-LIST, SEARCH or ESEARCH. Put the PQ-SELECT command in the stack so that it will be processed as part of the external job stream when the required list is active.

Retrieve the list elements one at a time, using a "!n" direct or an indirect reference.

You cannot execute PQ-SELECT directly from the shell.

EXAMPLE

```
PQN
HSSELECT SALES
STON
HPQ-SELECT 1
P
10 MV %1 !1
IF # %1 X Done
T %1
GO 10
```

This example selects all the records in the SALES file, loads the record-list into select register 1 and displays the keys one at a time.

JCL PQ-RESELECT

Executed from a jCL program, resets the pointer of a specified select register to the beginning of the list of record keys.

COMMAND SYNTAX

PQ-RESELECT register-number

SYNTAX ELEMENTS

register-number is the number (1 to 5) of the select register to be reset.

NOTES

This command is executed from the primary output buffer to reset the pointer of a specified select register back to the beginning of the list.

Each time you use the "!" reference to retrieve a value from the list, the value is not destroyed. The pointer is simply advanced to the next parameter in the list. PQ-RESELECT resets the pointer back to the beginning of the list so that you can perform another pass.

You cannot execute PQ-RESELECT directly from the shell.

EXAMPLE

```
HSELECT SALES
STON
HPQ-SELECT 1
PH
MV %1 !1
IF # %1 XNo records selected
HPQ-RESELECT 1
PH
10 MV %1 !1
IF # %1 XFinished
...
GO 10
```

jCL Commands

JCL A

Copies a parameter from the active input buffer to the active output buffer.

COMMAND SYNTAX

```
A{c|\}{p}
A{c|\}{p}({n},m)
```

SYNTAX ELEMENTS

c specifies a character to surround the string being copied. Can be any non-numeric character except left parenthesis "(" or backslash "\".

\ specifies that the value is to be concatenated with the current parameter in the active output buffer.

p specifies the number of the parameter to be copied. If p is not specified, the current parameter will be copied.

n specifies the starting column in the PIB. Copying continues until the end of the parameter is reached or the number of characters specified by m have been copied.

Notes

Used on its own, the A command will copy the parameter pointed to from the active input buffer to the active output buffer.

The A command can also be used with the IF command.

If the active input buffer pointer is at the end of the buffer, the A command will have no effect.

The c option is often used when you need to surround keys with single quotes, or values with double quotes.

If you include an n or m specification, the PIB will be selected. If the destination is the SOB (stack is "on"), c will be ignored.

If the stack is "off" (the POB is active), the A command will put the data in the output buffer as a separate parameter. The buffer pointers in the primary output buffer will be positioned at the field mark at the end of the copied parameter.

If the stack is "on" (the SOB is active), the A command will concatenate the value to the previous parameter in the output buffer. It will continue to copy until a field mark is encountered. When complete, the buffer pointers will be positioned at the end of the secondary output buffer.

EXAMPLE 1

```
Command    PIB Before      PIB After

A          AAA SALES^JAN   AAA^SALES^JAN
           ^               ^

           POB Before      POB After

           LIST^           LIST^SALES^
           ^               ^
```

Note the position of the buffer pointer after you issue the A command.

EXAMPLE 2

```
Command   PIB Before     PIB After

A         AAA^SALES^JAN  AAA^SALES^JAN
                   ^                  ^

          POB Before     POB After

          LIST^SALES^    LIST^SALES^JAN
                    ^                   ^
```

Issuing an A"3 command would have achieved the same result, except that the starting position of the PIB pointer would have been immaterial.

EXAMPLE 3

```
Command   PIB Before    PIB After

A\        ABC^DEF^GHI   ABC^DEF^GHI
          ^                          ^

          POB Before    POB After

          XXX^          XXXABC^
             ^                ^
```

EXAMPLE 4

```
Command      PIB Before      PIB After

A2 (2,-2)    ABC^MYLIST^JKL  ABC^MYLIST^JKL
                       ^                  ^

             POB Before      POB After

             SAVE-LIST^      SAVE-LIST MYLIST
                       ^                     ^
```

The command attempts to copy the second parameter from the PIB, starting with the second character, up to and including, the penultimate character, to the current output buffer.

JCL B

Moves the active input buffer pointer back to the previous parameter.

COMMAND SYNTAX

B

NOTES

The input buffer pointer is moved backwards to the preceding field mark or to the beginning of the input buffer. If the pointer is on the first character of a parameter (or a field marker), the pointer will be moved back to the field mark of the previous parameter.

EXAMPLE 1

Command	PIB Before	PIB After
B	ABC^DEF^GHIJK	ABC^DEF^GHIJK

EXAMPLE 2

Command	PIB Before	PIB After
B	ABC^DEF^GHIJK	ABC^DEF^GHIJK

JCL BO

Moves the active output buffer pointer back by one parameter.

COMMAND SYNTAX

BO

NOTES

The buffer pointer will move backward until it finds a field mark or the start of the buffer.

To completely clear the buffer, use the RO command. To clear specific parameters, use the MV #n command

EXAMPLE 1

Command	POB Before	POB After
BO	ABC^DEF^GHIJ K	ABC^DEF^GHIJK

EXAMPLE 2

Command	SOB Before	SOB After
BO	SAVE-LIST	SAVE-LIST

JCL D

Displays the current parameter of the active input buffer or specific parameters from the PIB.

COMMAND SYNTAX

D{n}{+}

SYNTAX ELEMENTS

n specifies the number of the PIB parameter to be displayed. If **n** is set to 0 (zero), all parameters in the primary input buffer will be displayed.

+ inhibits a NEWLINE at the end of output.

NOTES

D with no other qualifiers will display the current parameter of the active input buffer.

The pointer position will not be changed.

EXAMPLE 1

Command	Active Input Buffer	Display
D	ABC^DEF^GHI	DEF

EXAMPLE 2

Command	Active Input Buffer	Display
D3	ABC^DEF^GHI	GHI

EXAMPLE 3

Command	Active Input Buffer	Display
D0	ABC^DEF^GHI	ABC^DEF^GHI

JCL D

Displays the current parameter of the active input buffer or specific parameters from the PIB.

COMMAND SYNTAX

D{n}{+}

SYNTAX ELEMENTS

n specifies the number of the PIB parameter to be displayed. If **n** is set to 0 (zero), all parameters in the primary input buffer will be displayed.

+ inhibits a NEWLINE at the end of output.

NOTES

D with no other qualifiers will display the current parameter of the active input buffer.

The pointer position will not be changed.

EXAMPLE 1

Command	Active Input Buffer	Display
D	ABC^DEF^GHI	DEF

EXAMPLE 2

Command	Active Input Buffer	Display
D3	ABC^DEF^GHI	GHI

EXAMPLE 3

Command	Active Input Buffer	Display
D0	ABC^DEF^GHI	ABC^DEF^GHI

JCL DE

Displays the current value of LastError.

COMMAND SYNTAX

DE

NOTES

The DE command is mainly used when debugging your jCL programs.

EXAMPLE

If LastError contains 404]61^QLNUMSEL, the DE command will display:

```
404]61^QLNUMSEL
```

JCL DEBUG

Turns debug function on or off.

COMMAND SYNTAX

DEBUG [ON|OFF]

NOTES

When the DEBUG function is turned on, your jCL program will be suspended before each command (line) is executed. You will see the program name and the next command to be executed, and you will be prompted to enter one of the following commands:

Command	Description
<Enter>	execute current line
?	display list of available commands
e	toggle last error display
f	toggle file buffer display
h	display list of available commands
i	toggle input buffer display
n	toggle next line display between one and two lines
o	toggle output buffer display
q	quit (terminate) program
x	exit DEBUG function

After each input, the program name and the next line to be executed will be redisplayed, together with any additional information requested (for example, the content of the input buffers).

JCL F

Moves the active input buffer pointer forward to the next parameter.

COMMAND SYNTAX

F

NOTES

The input buffer pointer is moved forward to the next field mark, or to the end of the buffer.

EXAMPLE 1

Command	PIB Before	PIB After
F	ABC^DEF^GHI	ABC^DEF^GHI

EXAMPLE 2

Command	PIB Before	PIB After
F	ABC^DEF^GHI	ABC^DEF^GHI

JCL.F;

Provides a range of arithmetic functions.

COMMAND SYNTAX

F;element{;element}...;?[P|r]

SYNTAX ELEMENTS

element	*operators* or *operands* (see below)
?P	moves the result (top stack entry) into the current parameter of the primary input buffer.
?r	moves the result (top stack entry) into the reference r. Can be a direct or indirect reference to a buffer.

OPERATORS

+	Add last stack entry and penultimate stack entry. Store the result in stack entry 1.
-	Subtract last stack entry from penultimate stack entry. Store the result in stack entry 1.
*	Multiply the last two stack entries. Store the result in stack entry 1.
/	Divide stack entry 2 by stack entry 1. Store the result in stack entry 1.
R	Divide stack entry 2 by stack entry 1. Store the remainder in stack entry 1.
_	Reverse the last two stack entries.

OPERANDS

r	A direct or indirect reference to a buffer or select register value to be put on the stack.
{C}*literal*	Literal value specified by *literal*.

NOTES

The F; command uses a stack processor similar to, the one used by the F; function in jQL.

Commands are processed from left to right.

Each operand pushes a value on to the top of the push-down/pop-up stack.

The result of a calculation can be copied into any buffer with the question mark (?) operator.

EXAMPLE

Command	PIB Before	PIB After
F;C100;%2;-;?%3	6^8^999^7^GHI	6^8^92^7^GHI
Stack	100	

%2 pushes a value of 8 (the 2nd parameter of the PIB) onto the stack.

Stack	100
	8

-subtracts last entry from penultimate entry, and replaces last entry 1 with the result of 92.

Stack	92

?%3 then copies the result to the 3rd parameter of the PIB, replacing the old value.

JCL F-CLEAR

Clears the specified file buffer.

COMMAND SYNTAX

F-CLEAR file-buffer
F-C file-buffer

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer to be cleared.

NOTES

This command is equivalent to using the MV file-buffer.0 ",_ command

EXAMPLE

```
001 PQN
002 F-C 1
003 MV &1.0 "Key", "Field 1"
```

Clear file buffer 1. Set the record key to "Key" and the first field to "Field 1". Functionally equivalent to MV &1.0 "Key", " Field1",_. (Note the use of the underscore character as the last character of the command.)

JCL F-DELETE

Deletes a record from a file.

COMMAND SYNTAX

F-DELETE file-buffer
F-D file-buffer

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer which is associated with the file containing the record to be deleted.

NOTES

The record specified in field zero (&f.0) of the file buffer will be deleted. If the record does not exist, the command will have no effect.

The content of the file buffer is not altered.

An outstanding record lock will be released.

The file must have been opened with an F-OPEN command.

EXAMPLE

```
001 PQN
002 10 T "File name :",+
003 IBP %1
004 F-O 1 %1
005 T "File ", %1, " does not exist!"\ GO 10
006 MV &1.0 "Key"
007 F-D 1
```

If the file cannot be opened by the F-O command, the line immediately following the command will be executed (see the F-O command). If the file is opened, "Key" is moved into field 0 of file buffer 1. F-D 1 then attempts to delete record "Key" from the file.

JCL F-FREE

Releases a record lock set by the F-UREAD command.

COMMAND SYNTAX

F-FREE {file-buffer {key}}
F-F {file-buffer {key}}

SYNTAX ELEMENTS

file-buffer specifies a file buffer (1 to 9) assigned by an F-OPEN command.

key is the key of the record to be unlocked. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

NOTES

If file-buffer is not specified, all record locks set by the jCL program on the current port, within the current context level will be released.

If key is not specified, any outstanding locks for the current file will be released.

Record locks are also released by F-WRITE or F-DELETE commands, and at the end of the jCL program. Use the LIST-LOCKS command to see which records are currently locked.

EXAMPLE 1

```
F-FREE
```

Unlocks all records previously locked by this jCL program.

EXAMPLE 2

```
F-FREE 1
```

Unlocks the record specified by the key in field zero of file buffer 1.

EXAMPLE 3

```
F-F 1 Key
```

Unlocks the record "Key" in the file associated with file buffer 1.

JCL F-KLOSE

Closes a specified file buffer.

COMMAND SYNTAX

F-KLOSE file-buffer
F-K file-buffer

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer to be closed.

NOTES

F-KLOSE is normally only used when you have finished working with a remote file and need to close it.

EXAMPLE

```
F-K 1
```

Closes file buffer 1.

JCL F-OPEN

Clears a file buffer and opens a file for reading and writing.

COMMAND SYNTAX

F-OPEN file-buffer {DICT} file-name{,data-section-name} error-cmd-line

F-O file-buffer {DICT} file-name{,data-section-name} error-cmd-line

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer with which the file is to be associated.

DICT specifies the dictionary section of file-name, if required.

file-name is the name of the file to be opened. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

data-section-name specifies an alternative data section of file-name.

error-cmd-line is the line immediately after the F-OPEN command. Only executed if the specified file cannot be opened.

NOTES

If the file cannot be opened, the line immediately after the F-OPEN command will be executed. If the file is opened successfully, this line will be ignored.

File buffers are maintained when control is transferred between jCL programs.

The file will remain open until closed (see the F-KLOSE command) or until the end of the program.

EXAMPLE

```
001 PQN
002 F-OPEN 1 SALES
003 X ERROR: Can’t find the Sales File!
004 T C, (5,10), "Welcome to...",+
.
.
```

If the SALES file is opened, execution continues with line 004. Otherwise, the program terminates with an appropriate error message.

JCL F-READ

Reads a record from an open file into a file buffer.

COMMAND SYNTAX

F-READ file-buffer key
error-cmd-line

F-R file-buffer key
error-cmd-line

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer with which the file is associated.

key is the key of the record to be read. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

error-cmd-line is the line immediately after the F-READ command. Only executed if the specified record cannot be read.

NOTES

If an associated file has not been opened (see the F-OPEN command), the program will terminate with an error message.

If the specified record is not on file, the line immediately following the F-READ command will be executed. If the read is successful, this line will be ignored.

EXAMPLE

```
001 PQN
002 F-OPEN 1 SALES
003 X ERROR: Can't find the Sales File!
004 T C, (5, 10), "Welcome to...",+
...
015 F-READ 1 ABC
016 X ERROR: Record ABC not found!
017 T "Record ABC found"
```

Attempts to read record ABC from the SALES file into file buffer 1. If record ABC is not found, the program terminates with an appropriate error message. If the record is found, the message on line 17 is displayed and execution continues.

JCL F-UREAD

Reads and locks a record from an open file into a file buffer.

COMMAND SYNTAX

F-UREAD file-buffer key
error-cmd-line

F-UR file-buffer key
error-cmd-line

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer with which the file is associated.

key is the key of the record to be read and locked. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

error-cmd-line is the line immediately after the F-UREAD command. Only executed if the specified record cannot be read.

NOTES

The F-UREAD command is identical to the F-READ command, except that it also locks the record against access by another process, thus eliminating simultaneous updates.

If you attempt to F-UREAD a record which is already locked, execution will be suspended until the record is unlocked by the other process.

Record locks are released by F-DELETE, F-WRITE or F-FREE commands, or when the program terminates.

It is good practice to F-UREAD a record before you create it. This will reserve the key in the file and avoid double updates. Remember though that the command line immediately following the F-UREAD command will be executed because the record does not exist.

JCL F-WRITE

Writes the contents of a file buffer as a record.

COMMAND SYNTAX

F-WRITE file-buffer
F-W file-buffer

SYNTAX ELEMENTS

file-buffer is the number (1 to 9) of the file buffer with which the target file is associated.

NOTES

The key of the record is contained in field zero of the file buffer. If this field is null, the record will not be written.

F-WRITE will not alter the contents of the file buffer.

You should not normally attempt to write a record unless you have first locked it with an F-UREAD command. The lock will be released when the F-WRITE is complete.

If the file has not been opened (see the F-OPEN command), the program will terminate with an error message.

EXAMPLE

```
001 PQN
002 F-OPEN 1 SALES
003 X ERROR: Can't find the Sales File!
004 T C, (5, 10), "Welcome to...",+
.
015 F-UREAD 1 ABC
016 F-F 1 \ G 1002
017 T "Record ABC found"
018 MV &1.2 "Field 2"
019 F-WRITE 1
.
```

Line 15 reads and locks record ABC in file SALES. If the record does not exist, the lock is released on line 16 and control is transferred to label 1002. If the record is read successfully, field 2 is overwritten on line 18. The record is then written back to the file on line 19 and unlocked.

JCL FB

Reads a record from a file into the special "fast" buffer without having to open the file first.

COMMAND SYNTAX

FB {DICT} file-name{,data-section-name} {key} error-cmd-line
FB ({DICT} file-name{,data-section-name} {key}) error-cmd-line

SYNTAX ELEMENTS

DICT specifies the dictionary section of file-name, if required.

file-name is the name of the file to be opened. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

data-section-name specifies an alternative data section of file-name.

key is the key of the record to be read. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register. If key is not specified, the value at the active input buffer
pointer is used to supply the key.

error-cmd-line is the line immediately after the FB command. Only executed if the specified file cannot be opened or the record read.

NOTES

Each time you use the FB command, the previous contents of the buffer will be overwritten.

The FB command is useful if you are only reading one record from a file. Otherwise, you should use the F-OPEN and F-READ commands.

If the specified record is not on file, or if the file does not exist, the line immediately following the FB command will be executed. If the read is successful, this line will be ignored.

Subsequent references to the fast buffer use a special syntax. For example, to refer to the second field of a record in the fast buffer, you would use &2.

EXAMPLE 1

```
001 PQN
002 FB SALES ABC
003 T "ABC not on file" / G 1001
004 MV %3 &2
.
```

The FB command on line 2 attempts to read record ABC from file SALES. If the record cannot be found for any reason, a message is output and control transferred to label 1001 by line 3. If the record is read successfully, execution continues at line 004 which moves field 2 of the record into parameter 3 of the PIB.

EXAMPLE 2

```
001 PQN
002 T C, (5, 10), "Name :",+
003 IP %2
004 FB SALES %2
005 T "New record"
006 T "Area :",+
007 IP %3
```

Here the user is prompted for a name (the record key) and the fast buffer is used to read the record from the SALES file. If the record does not exist, a message is output but processing continues.

JCL GO B

Transfers control to the statement following the most recent M (mark) command executed.

COMMAND SYNTAX

G B
GO B
GOTO B

NOTES

GO B goes back to the last executed M, no matter where it is located in the program.

If a "mark" has not previously been executed, the program will terminate with an error message:

Can't find mark at line n in programname

EXAMPLE 1

```
001 PQN
.
010 MV #1 "SSELECT SALES BY MONTH"
011 STON
012 MV #1 "PQ-SELECT 1"
013 P
014 M
015 MV %1 !1
016 IF # %1 GO F
017 C Process the record
.
025 GO B
026 M
```

Lines 10 to 13 create a sorted list of the records in the SALES file. After each record is processed, the GO B command on line 25 returns control to the M command on line 14. When the end of the list is reached, the IF command on line 16 transfers control to the M command on line 26.

EXAMPLE 2

```
001 PQN
.
009 MV %1 ","
010 M
011 IF # %1 GO 30
012 M
013 IF # %2 GO 40
014 GO 50
015 30 MV %1 "ABC"
016 GO B
017 40 MV %2 "DEF"
018 GO B
019 50 ....
```

This example simply serves to demonstrate how the GO B command will remember the last M command rather than search backwards for it. First, the values in %1 and %2 are set to null in line 9 and the M at line 10 is recorded. When control reaches line 11, %1 is tested and control is transferred to label 30 on line 15. The intervening M command (at line 12) is not recorded. Line 15 assigns a value of ABC to %1 and line 16 transfers control back to the M on line 10. %1 does have a value now, so control moves on to line 12 and the M on this line is recorded. Next, %2 is tested at line 13 and control is transferred to label 40 on line 17. When the GO B is processed on line 18, control is transferred back to line 12.

JCL GO F

Transfers control to the statement containing the next M (mark) command.

COMMAND SYNTAX

G F
GO F
GOTO F

NOTES

The program is scanned forward from the current line, until the next M command is found. Program execution then jumps to that line.

If an M command cannot be found, the jCL program will terminate with an error message:

Can't find mark at line n in programname

EXAMPLE

```
001 PQN
.
005 GO F
006 10 MV %1 "ABC"
007 MV %6 "DEF"
008 MV %10 "HIJ"
009 M
.
```

The GO F command on line 5 causes the program to be scanned from line 6, looking for the next M command. In this case, control will be transferred to line 9.

JCL GOSUB

Transfers control to a specific subroutine.

COMMAND SYNTAX

GOSUB label
GOSUB label] label... (Multivalued form)

SYNTAX ELEMENTS

label specifies the label which marks the required subroutine.

NOTES

When an RSUB statement is encountered, control is transferred back to the line immediately following the calling GOSUB.

See also "[ ] {n}" command.

MULTIVALUED FORM

To use the multivalued form of the GOSUB command, you must specify one label for each result of a multiple comparison. For example:

```
IF %2 = A]B]C]D GOSUB 1000]2000]3000]4000
```

Separate the test values and the destination labels with value marks (ctrl]).

Note that this is a special case of the GOSUB command. If you need to mix command types in the resulting actions, you should not use this form of the GOSUB command. You can still achieve the same effect but each result must contain a separate command. For example:

```
IF %2 = A]B]" GOSUB 1000]GOSUB 2000]XFinished
```

In this case, if the result of the test for null is true the program will terminate with a suitable message.

EXAMPLE

```
010 GOSUB 1000
011 T "Returned from GOSUB"
.
031 1000 T "In subroutine"
032 IP %1
033 RSUB
```

The GOSUB command on line 10 transfers control to label 1000 on line 31. When the RSUB on line 33 is processed, control returns to line 11.

JCL G / GO / GOTO

Transfers control unconditionally to another location in the program.

COMMAND SYNTAX

G label
GO label
GOTO label
GO label] label... (Multivalued form used with multivalued IF command)

SYNTAX ELEMENTS

label specifies the location from which execution is to continue.

NOTES

If the label has not already been encountered in the program, GOTO will search for the label, from the current position. The target label must be found at the beginning of a command line, separated from the command by at least one space.

If the label cannot be found, or is defined more than once, the program will terminate with an error message.

MULTIVALUED FORM

To use the multivalued form of the GO command, you must specify one label for each result of a multiple comparison. For example:

```
IF %2 = A]B]C]D GO 10]20]30]40
```

Separate the test values and the destination labels with value marks (ctrl]).

Note that this is a special case of the GO command. If you need to mix command types in the resulting actions, you should not use this form of the GO command. You can still achieve the same effect but each value must contain a separate command. For example:

```
IF %2 = A]B]C]" GO 10]GO 20]GO 30]XFinished
```

In this case, if the result of the test for null is true the program will terminate with a suitable message.

EXAMPLE 1

```
001 PQN
002 F-OPEN 1 SALES
003 G 1001
004 T C, (5, 10), "Welcome to...",+
.
087 1001 T "ERROR: Can"t find the Sales File!"
088 IP %99
089 RTN
.
```

If the SALES file is opened, execution continues with line 004. Otherwise, control is transferred to label 1001 on line 87.

EXAMPLE 2

```
022 5 T "Option :",+
023 IP %1
024 IF %1 = A]B]C GO 10]20]30
025 GOTO 5
.
```

This example transfers control to label 10 if "A" is entered, to label 20 if "B" is entered and to label 30 if "C" is entered. If the response is not recognized, control transfers back to label 5.

JCL H

Places a text string in the active output buffer.

COMMAND SYNTAX

Htext-string

SYNTAX ELEMENTS

text-string is the text to be placed in the active output buffer. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

NOTES

The H command is used to place a text string in the currently active output buffer. Use the POB, to create a shell command. Use the SOB to create secondary commands (such as PQ-SELECT) or to "stack" a series of inputs required by the active process.

The string is moved into the buffer starting at the current location of the buffer pointer. At the end of the operation, the buffer pointer will be positioned immediately after the last character in the string.

If quotes are used to delimit the string everything within quotes will be treated as a single field and the string will be moved into the buffer as a single parameter.

If quotes are not used, each group of one or more spaces in the string will be replaced by a field mark as the text is moved into the buffer. Include a leading space if you want to add a new parameter. If you want to concatenate the string to the current buffer parameter, do not use a leading space.

The P command is used to process the contents of the POB and SOB.

USING H WITH THE PRIMARY OUTPUT BUFFER

When the shell command is issued, field marks will be replaced by spaces and a carriage return will be appended automatically.

USING H WITH THE SECONDARY OUTPUT BUFFER

A carriage return is not appended automatically to output from the SOB. Terminate each line with a less than character (<) to represent a carriage return.

EXAMPLE 1

```
001 PQN
002 HSLEEP 10
003 P
```

Forces the process to sleep for 10 seconds.

EXAMPLE 2

Command	POB Before	POB After
H		COPY
H SALES	COPY	COPY^SALES
H ABC	COPY^SALES	COPY^SALES^ABC
H-DEF	COPY^SALES^ABC	COPY^SALES^ABC-DEF
H (P)	COPY^SALES^ABC-DEF	COPY ^SALES^ABC-DEF^(P)

Note how COPY and SALES have become separate parameters but ABC and -DEF have been concatenated.

EXAMPLE 3

```
001 PQN
002 HGET-LIST LISTA
003 STON
004 HCOPY SALES<
005 H(SALES.HOLD
006 P
```

JCL IBH

Places text in the active input buffer whilst retaining embedded spaces and applying any required input/output conversions.

COMMAND SYNTAX

```
IBHtext
IBHreference;input-conversion;
IBHreference:output-conversion:
```

SYNTAX ELEMENTS

text is the text to be placed in the active input buffer. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register.

reference is a direct or indirect reference to a buffer or select register.

input-conversion is a jQL input conversion to be applied to the string before putting it in the buffer.

output-conversion is a jQL output conversion to be applied to the string before putting it in the buffer.

NOTES

The IBH command works in the same way as the IH command except that the string is moved as a single parameter and all spaces are maintained.

Depending on the position of the buffer pointer, IBH will either replace an existing parameter or adds a new parameter to the end of the input buffer. The rules are as follows:

- If the buffer pointer is at the beginning of an existing parameter, that parameter will be replaced with the new string.
- If the buffer pointer is within an existing parameter, IBH will concatenate the new string (without inserting a field mark), starting at the current location of the buffer pointer.
- If the buffer pointer is at the end of the input buffer, a new parameter will be created and the buffer pointer will be left pointing to the field mark preceding the new parameter.

In all cases, the position of the buffer pointer will remain unchanged.

Conversions containing colons or semicolons will not work (for example IBH;G1;1;).

EXAMPLE 1

Command	PIB Before	PIB After
IBHDEF GHI	ABC^XXX^Z	ABC^DEF GHI^Z

EXAMPLE 2

Command	PIB Before	PIB After
IBH XX	AA^BB^CC^DD	AA^BB^CC^DD^ XX

EXAMPLE 3

File buffer 1 contains:
000 Key
001 11350

Command	PIB Before	PIB After
IBH&1.1:D2:	AA^BB^CC^DD	AA^BB^27 JAN 99^DD

JCL IBN

Prompts for input, places the entered data in the secondary input buffer as a single parameter and maintains embedded spaces. The secondary input buffer becomes as the active input buffer.

COMMAND SYNTAX

IBN{c}

SYNTAX ELEMENTS

c is an optional prompt character which, once used, remains in effect until a new IBN, IBP, IN or IP command is issued. If **c** is not specified, the prompt character will default to the last prompt character used, or to a colon (:).

NOTES

The IBN command is similar to the IN command except that the input string is placed in the buffer as a single parameter and all spaces are maintained.

The new data replaces the content of the secondary input buffer, and the secondary input buffer will remain active until an RI, S(n) or MV %n source command is used.

If the user responds with ENTER only, a null parameter will be created.

EXAMPLE 1

Input	SIB Before	SIB After
ABC	XXX	ABC

EXAMPLE 2

Input	SIB Before	SIB After
ABC DEF	XXX	ABC DEF

EXAMPLE 3

Input	SIB Before	SIB After
<ENTER>	XXX	

JCL IBP

Prompts for input from the terminal. Input data is kept as a single parameter and embedded spaces are retained.

COMMAND SYNTAX

IBP{c{r}}

SYNTAX ELEMENTS

c is an optional prompt character which, once used, remains in effect until a new IBN, IBP, IN or IP command is issued. If c is not specified, the prompt character will default to the last prompt character used, or to a colon (:).

r is a direct or indirect reference to a buffer or select register which is to receive the data. If you use a reference, the prompt character c must be specified.

NOTES

The IBP command is similar to the IP command except that the input is placed in the buffer as a single parameter and embedded spaces are maintained.

If you do not specify a buffer reference, the active input buffer will be used.

The new data will always replace the parameter pointed to by the buffer pointer but the position of the pointer will not be changed.

If the user responds with RETURN only, a null parameter will be created.

EXAMPLE 1

```
Command    PIB Before   Input    PIB After
IBP?       AAA^BBB      CCC      AAA^BBB^CCC
               ^                         ^
```

EXAMPLE 2

```
Command   PIB Before    Input    PIB After
IBP?      AA^BB^CC      XX X     AA^XX X^CC
            ^                      ^
```

EXAMPLE 3

```
Command   PIB Before    Input    PIB After
```

```
IBP?       ABC^DEF^GHI   <RETURN> ABC^^GHI
               ^                      ^
```

EXAMPLE 4

Command	File Buffer 2 Before	Input	File Buffer 2 After
IBP:&2.2	000 Key	BBB	000 Key
	001 AAA		001 AAA
	002 XXX		002 BBB
	003 CCC		003 CCC

EXAMPLE 5

Command	File Buffer 2 Before	Input	File Buffer 2 After
IBP:&2.2	000 Key	<RETURN>	000 Key
	001 AAA		001 AAA
	002 XXX		002
	003 CCC		003 CCC

JCL IF E

Conditionally executes a command depending on the presence or absence of an error message after running a shell command.

COMMAND SYNTAX

IF {#} E command
IF E operator msg-key command

SYNTAX ELEMENTS

tests for the absence of an error message.

operator performs a value comparison. Operators are:

=	Equal to
#	Not equal to.

msg-key is the key of a system message from the error file.

command is a valid jCL command.

NOTES

Any system messages generated as a result of running a shell command (see the P command) will cause the system message number to be placed in the SIB, in multivalue format. The value tested is the first multivalue (the STOP code) of the error text returned from the last command.

The IF E command is most often used to test for an error condition but can be used to detect any resulting system message. IF # E command tests for the absence of a system message.

Some jCL commands, particularly those that operate on the PIB, will destroy the contents of the secondary input buffer. You should therefore use the IF E command as soon as control returns from the shell command.

EXAMPLE

```
021 HCOUNT SALES WITH VALUE > "1000"
022 PH
023 IF E = 401 G 100
```

If the SALES file does not contain any records which match the criteria, the system will generate the message "No records counted". Using the PH command will stop the message being output on the terminal but the message key 401 will still be placed in the SIB where it can be detected by line 23.

JCL IF

Allows conditional execution of jCL commands based on the evaluation of an expression, or the presence (or absence) of a reference.

COMMAND SYNTAX

IF{#} reference command

or

IF reference operator expression command

SYNTAX ELEMENTS

tests for the absence of a value.

reference can be a direct or indirect reference to a buffer or select register, or an A command without a surround character. Using an A command will not move a value but simply provides a reference to the parameter to be tested.

operator performs a value comparison. Operators are:

=	Equal to
#	Not equal to
<	Less than
>	Greater than
[	Less than or equal to
]	Greater than or equal to

expression follows the operator and can be one of the following:

- A direct or indirect reference to a buffer or select register.
- A string. If the string contains embedded spaces or the first character is an exclamation mark (!), percent sign (%) or ampersand (&), enclose the string in single or double quotes.
- A pattern format string. Refer to the IF (with Mask) command.

command is any valid jCL command.

NOTES

If the test result is true, the command at the end of the statement is executed. If the test yields false, command is ignored and processing continues with the next line.

Buffer pointers will not be moved if you use a direct or indirect reference to a buffer or select register.

Comparative tests are performed on a character-by-character, left-to- right basis. For example, AAB is greater than AAAB and 20 is greater than 100. Use the IFN command if you want to perform numeric comparisons. You can use also perform multiple conditional tests in a single statement. For example:

```
IF %1 > A IF %1 < D T %1  is B or C
```

EXAMPLE 1

```
021 IF %1 T "%1 is not null"
022 IF # %1 T "%1 is null"
023 ...
```

Line 21 tests for a value in the 1st parameter of the PIB and outputs a message if the parameter is not null. Line 22 tests for opposite case.

EXAMPLE 2

```
021 IF &1.1 = ABC GO 10
022 MV %3 &1.1
023 10 ...
```

If &1.1 contains ABC execution will branch to line 23. Otherwise, the value in &1.1 will be moved into %3.

EXAMPLE 3

```
010 T "Continue (Y/n) :",+
011 IP %1
012 S1
013 IF # A G 10
014 IF A(1,1) = Y G 10
015 IF A(1,1) = y G 10
016 XFinished
017 10 ...
```

If the user enters Y, y or <ENTER> to the prompt on line 11, execution will continue at label 10 on line 17. Otherwise, the program will terminate.

IF (MULTIVALUED)

Compares an expression with one of several different expressions or values, and conditionally executes one of several commands.

COMMAND SYNTAX

IF reference = expression{]expression}... command{]command]}...
IF reference # expression{]expression}... command.

SYNTAX ELEMENTS

reference can be a direct or indirect reference to a buffer or select register, or an A command without a surround character. Using an A command will not move a value but simply provides a reference to the parameter to be tested.

expression follows the operator and can be one of the following:

- A direct or indirect reference to a buffer or select register.
- A string. If the string contains embedded spaces or the first character is an exclamation mark (!), percent sign (%) or ampersand (&), enclose the string in single or double quotes.
- A pattern format string. Refer to the IF (with Mask) command.

] represents a value mark (Ctrl])

command is any valid jCL command.

NOTES

The multivalues feature of the IF command enables one IF statement to be used instead of a series.

Do not use O or X commands unless they are the last in the series. Commands after an O or X will be ignored.

The equal (=) operator, will perform a logical OR on the expressions. If the reference is equal to any expression, the condition is true. If more than one expression is true, the command corresponding to the first true expression is executed. If there are more expressions than commands, and the true expression does not have a corresponding command, the last command in the series will be executed. If there are more commands than expressions, remaining commands are ignored.

If you use the not equal (#) operator, a logical AND is performed on the expressions. The reference must not equal any expression in the series for the condition to be true. In this case, the first command specified is executed. Subsequent commands are ignored.

If a direct or indirect reference to a buffer or select register identifies a multivalued parameter, the same jCL statement is executed regardless of which of the multivalues is true. This means that each value will not access a different command - as it would have if you had specified it directly in the IF statement. For example:

```
MV %1 A]B]C]D
IF %1 = X]Y]Z]C GO 10]20]30]40
```

will cause program execution to continue from label 40.

GO AND GOSUB COMMANDS

To use the special multivalued forms of the GO and GOSUB commands, you must specify one label for each result of a multiple comparison. For example:

```
IF %2 = A]B]C]D GO 10]20]30]40
```

Note that this is a special case of the GO and GOSUB commands. If you need to mix command types in the resulting actions, you should not use this form of the GO and GOSUB commands. You can still achieve the same effect but each result must contain a separate command. For example:

```
IF %2 = A]B]C]" GO 10]GO 20]GO 30]XFinished
```

In this case, if the result of the test for null is true the program will terminate with a suitable message.

EXAMPLE 1

```
IF %1 = A]B]C G 10
```

If %1 is equal to A, B or C, control is transferred to label 10.

EXAMPLE 2

```
IF %2 # "AA"]&1.1](2N)]" GOSUB 1001
```

If %2 is not equal to AA, the content of &1.1, two numerics or null, control is transferred to subroutine 1001.

EXAMPLE 3

```
IF %3 = (#0N)]($ON) GO 10]MV #4 "DOLLARS"
```

If %3 equals a pound sign (#) followed by any (or no) numerics, control transfers to the statement with label 10.
If %3 equals a dollar sign ($) followed by any (or no) numerics, the string "DOLLARS" is moved into the output buffer.

IF (WITH MASK)

Conditionally executes commands based on a comparison with a specified pattern.

COMMAND SYNTAX

```
IF reference operator (pattern) command
```

SYNTAX ELEMENTS

reference can be a direct or indirect reference to a buffer or select register, or an A command without a surround character. Using an A command will not move a value but simply provides a reference to the parameter to be tested.

operator performs a value comparison. Operators are:

=	Equal to.
#	Not equal to.

(pattern) is a format string enclosed in parentheses that tests for a specified numeric, alphanumeric or alphabetic string. A pattern format string can consist of any combination of the following:

(nA)	Tests for n alphabetic characters. If n is 0 (zero), any number (or no) alphabetic characters are assumed to match.
(nC)	Tests for n alphanumeric characters. If n is 0 (zero), any number (or no) alphanumeric characters are assumed to match.
(nN)	Tests for n numeric characters. If n is 0 (zero), any number (or no) numeric characters are assumed to match.
(nP)	Tests for n printable characters. If n is 0 (zero), any number (or no) printable characters are assumed to match.
(nX)	Tests for n characters. If n is 0 (zero), any number (or no) characters are assumed to match.
(string)	Tests for one or more specified characters. If string contains numeric characters, %, #, &, !, or spaces, it must be enclosed in quotes.

For example, the pattern (2N"-"3A"-"2N) specifies a format of two numeric characters, a hyphen, three alphabetic characters, another hyphen and then two numeric characters.

command is a valid jCL command.

NOTES

0X can only be used as the last specification in a pattern.

A null value will match any of the defined patterns such as (0A), (0N), or (0X).

If you want to test for spaces in the pattern, use double quotes, for example:
IF %n = (2A" "3N" "0X)

Ambiguous literal strings must be enclosed in quotes.

EXAMPLE 1

```
IF %3 = (3N1A) GO 10
```

If %3 matches three numerics followed by one alphabetic character, branch to label 10.

EXAMPLE 2

```
IF &1.2 # (2N"AAA") GOSUB 1001
```

If &1.2 does not equal two numerics followed by AAA, control will be transferred to subroutine 1001.

EXAMPLE 3

```
IF %1 # (2N*2A*0N) MV %1 "99*AA*1234"
```

If %1 does not equal two numerics, an asterisk, two alphabetics, an asterisk, and a number (or no numerics), move a string that does into %1.

EXAMPLE 4

```
IF &1.1 = ("(6X)") GO 100
```

If &1.1 contains "(ABC123)", control will be transferred to label 100.

EXAMPLE 5

```
IF &1.1 = ("("0X")") GO 100
```

If &1.1 contains "(ABC123)", control will not be transferred to label 100.

EXAMPLE 6

```
IF &1.1 = ("("0X) GO 100
```

If &1.1 contains "(ABC123)", control will be transferred to label 100.

JCL IF S

Conditionally executes a command depending on the presence or absence of an active select list.

COMMAND SYNTAX

IF {#} S command

SYNTAX ELEMENTS

tests for the absence of an active select list.

command is a valid jCL command.

NOTES

IF S will execute command if there is an active select list. IF # S will execute command if there is not an active select list.

EXAMPLE

```
021 HSELECT SALES WITH VALUE > "1000"
022 PH
023 IF S G 100
```

If the SELECT command generates an active select list, control will be transferred to label 100.

JCL IFN

Allows conditional execution of commands depending on the result of numeric comparisons.

COMMAND SYNTAX

IFN reference operator expression command

SYNTAX ELEMENTS

reference can be a direct or indirect reference to a buffer or select register, or an A command without a surround character. Using an A command will not move a value but simply provides a reference to the parameter to be tested.

operator performs a value comparison. Operators are:

=	equal to
#	not equal to
<	less than
>	greater than
[	less than or equal to
]	greater than or equal to

expression can be one of the following:

- A direct or indirect reference to a buffer or select register.
- A string. If the string contains embedded spaces or the first character is an exclamation mark (!), percent sign (%) or ampersand (&), enclose the string in single or double quotes.

command is a valid jCL command.

NOTES

IFN tests numbers by value, rather than their ASCII sequence of characters.

Leading zeros are ignored, as are trailing decimal zeros. For example, if %1 contains 00123.000, IFN %1 = 123 will be true.

If a submitted value equates to null or is non-numeric, zero will be used. Leading plus or minus signs are allowed.

EXAMPLE 1

```
IFN %1 > 50 G 100
```

If %1 is greater than 50, control will be transferred to label 100.

EXAMPLE 2

```
IFN %1 [ 0 G 100
```

If %1 is less than or equal to 0, control will be transferred to label 100.

EXAMPLE 3

```
IFN A < %3 G 100
```

If the current parameter is less than %3, control will be transferred to label 100.

JCL IH

Places a text string in the active input buffer, clears an existing parameter, or creates new null parameters.

COMMAND SYNTAX

```
IHtext
```

```
IHreference;input-conversion;
```

```
IHreference:output-conversion:
```

```
IH\
```

```
IH \
```

SYNTAX ELEMENTS

text is the text to be placed in the active input buffer. Can be a literal (not enclosed in quotes), or a direct or indirect reference to a buffer or select register. The text must not contain subvalue marks.

reference is a direct or indirect reference to a buffer or select register.

input-conversion is a jQL input conversion to be applied to the string before putting it in the buffer.

output-conversion is a jQL output conversion to be applied to the string before putting it in the buffer.

\ (backslash). If there is no preceding space, the current parameter in the active input buffer will be nulled. If there is a preceding space, a new null parameter will be created at the current buffer pointer position. A backslash in any other position will be treated as part of the literal text.

NOTES

Each group of one or more spaces in the text will be replaced with a single field mark, thereby creating new, separate parameters to replace the current single parameter. Leading and trailing spaces within text are ignored.

Use the IBH command if you want to insert text into the active input buffer as a single parameter with all blanks intact.

If the buffer pointer is at the beginning of an existing parameter, that parameter will be replaced by the new text.

If the buffer pointer is in the middle of a parameter, that parameter will be truncated from the current location and the new parameter (without a leading field mark) will be concatenated.

If the buffer pointer is at the end of the input buffer, one or more new parameters will be created.

The position of the buffer pointer will not be changed.

CREATING NULL PARAMETERS

If the buffer pointer is at the start of a parameter, IH\ will null the parameter. The characters between the field marks are deleted but the field marks are retained.

If the buffer pointer is in the middle of a parameter, IH\ removes the remaining characters, from that point to the end of the parameter.

If the buffer pointer is at the end of the buffer, IH\ creates a new null parameter.

IH \ creates a new null parameter at the position pointed to by the input buffer pointer. Note the space between the H and the backslash character.

EXAMPLE 1

Command	PIB Before	PIB After
IHXXX	AB^CD^YY^ZZ	AB^CD^XXX^ZZ

EXAMPLE 2

Command	PIB Before	PIB After
IH GH IJ	AB^CD^EF	AB^CD^EF^GH^IJ

EXAMPLE 3

%3 contains 9873

Command	PIB Before	PIB After
IH%3:D2:	AB^CD^9873^GH	AB^11^JAN^95^9873^GH

EXAMPLE 4

Command	PIB Before	PIB After
IH%4	AB^CD^EF^GH IJ^	AB^CD^EF^GH^IJ^

This example demonstrates how, in effect, you can replace a space with a field mark within a parameter. The 4th parameter of the PIB is copied back into the same location but the space is replaced by a field mark.

EXAMPLE 5

Command	PIB Before	PIB After
IH\	AB^CD^EF^GH	AB^CD^^GH

EXAMPLE 6

Command	PIB Before	PIB After
S(7)		
IH\	AB^CDEFGH^IJK	AB^CDE^IJK

This example demonstrates how to truncate a parameter.

EXAMPLE 7

Command	PIB Before	PIB After
IH \	AB^CD^EF^GH	AB^CD^^EF^GH

EXAMPLE 8

Command	PIB Before	PIB After
IH \	AB^CDEFGH^IJK	AB^CDE^^FGH^IJK

JCL IN

Prompts for input and places it in the secondary input buffer. Selects the secondary input buffer as the active buffer input.

COMMAND SYNTAX

IN{c}

SYNTAX ELEMENTS

c is an optional prompt character which, once used, remains in effect until a new IBN, IBP, IN or IP command is issued. If **c** is not specified, the prompt character will default to the last prompt character used, or to a colon (:).

NOTES

The new data replaces the content of the SIB, and the SIB will remain active until an RI, S(n) or MV %n source command is used.

Leading and trailing spaces are removed and groups of one or more embedded spaces are replaced by a single field mark. Use the IBN command if you want to maintain embedded spaces.

If the user responds with ENTER only, a null parameter will be created.

When the command has been completed, the buffer pointer will be positioned at the beginning of the buffer.

EXAMPLE 1

Input	SIB Before	SIB After
ABC		ABC

EXAMPLE 2

Input	SIB Before	SIB After
ABC DEF	XYZ	ABC^DEF

EXAMPLE 3

Input	SIB Before	SIB After
<ENTER>	WWW^XXX	

JCL IP

Prompts for input and places it into the active input buffer or a nominated buffer.

COMMAND SYNTAX

IP{c{r}}

SYNTAX ELEMENTS

c is an optional prompt character which, once used, remains in effect until a new IBN, IBP, IN or IP command is issued. If c is not specified, the prompt character will default to the last prompt character used, or to a colon (:).

r is a direct or indirect reference to a buffer or select register which is to receive the data. If you use a reference, the prompt character c must be specified.

NOTES

If you do not specify a buffer reference, the active input buffer will be used.

The new data will replace the parameter at the current buffer pointer position but the pointer will not be moved.

Leading and trailing spaces will be removed and groups of one or more embedded spaces will be replaced by a single field mark. By replacing a parameter with data that contains spaces, you can insert several new parameters.

When you place data containing embedded spaces into a file buffer, the new parameters will replace successive fields in the buffer. For example, if the response to an IP?&2.1 command is:

<SPACE>AA<SPACE><SPACE>BB<SPACE>CC"

fields one, two, and three, of file buffer 2 will be replaced with "AA", "BB", and "CC".

If the user responds with RETURN only, a null parameter will be created.

If you want to keep the input data exactly as entered, use the IBP command.

EXAMPLE 1

Command	PIB Before	Input	PIB After
IP?	AAA^BBB	CCC	AAA^BBB^CCC

EXAMPLE 2

Command	PIB Before	Input	PIB After
IP?	AA^BB^CC	XX X	AA^XX^X^CC

EXAMPLE 3

Command	PIB Before	Input	PIB After
IP?	ABC^DEF^GHI	<ENTER>	ABC^^GHI

EXAMPLE 4

Command	File Buffer 2 Before	Input	File Buffer 2 After
IP:&2.2	000 Key	BBB	000 Key
	001 AAA		001 AAA
	002 XXX		002 BBB
	003 CCC		003 CCC

EXAMPLE 5

Command	File Buffer 2 Before	Input	File Buffer 2 After
IP:&2.2	000 Key	BB CC DD	000 Key
	001 AAA		001 AAA
	002 XXX		002 BB
	003 DDD		003 CC
			004 DD

JCL IT

Reads a tape record into the primary input buffer.

COMMAND SYNTAX

IT{C}{A}

SYNTAX ELEMENTS

C performs an EBCDIC to ASCII conversion before the data is put into the buffer.

A masks 8-bit ASCII characters to 7 bits (resets the most significant bit to a 0).

NOTES

The IT command will read a tape record into the primary input buffer.

The new data will be placed at the beginning of the buffer and will replace all existing buffer data.

EXAMPLE

Command	PIB Before	PIB After
IT	ABC	*tape data*

JCL L

Formats printed output.

COMMAND SYNTAX

L element{, element}...

SYNTAX ELEMENTS

element can be any of the following:

"text"	Prints literal text. Enclose the text in quotes.
r{;input;}	Prints the result of a direct or indirect reference to a buffer or select register (r). If required, a jQL input conversion can be applied before output.
r{:output:}	Prints the result of a direct or indirect reference to a buffer or select register (r). If required, a jQL output conversion can be applied before output.
n	Skips n lines before printing. You cannot use n in a HDR print format specification.
(c)	Sets the print position to column number c. Can be a direct or indirect reference to a buffer or select register.
+	Suppresses NEWLINE being output at the end of the L command. You cannot use + in a HDR print format specification.
C	Closes the print file and forces printing. You cannot use C in a HDR print format specification.
E	Ejects to top-of-form (form feed). You cannot use E in a HDR print format specification.
N	Redirects subsequent printer output to the terminal. Normally only used for debugging. Must be the only element in an L command.
HDR	Allows you to define a page heading. HDR, must be the first element in the L command.
L	Outputs a line feed in the heading. Only used in a HDR specification.
P	Outputs the current page number in a heading. Only used in a HDR specification.
T	Outputs current time and date in a heading. Only used in a HDR specification.
Z	Resets the current page number in a heading to zero. Only used in a HDR specification.

NOTES

Output from the L command creates (or adds to) a print file.

The print file will remain open until a shell command is issued by using the P command. If the shell command also generates print output, this will be added to the same print file. You can close the print file and avoid any unnecessary output by choosing any shell command which does not generate print output, or by using the L C command. Alternatively, you can hold the print file open indefinitely by using the O option of the SP-ASSIGN command.

Continuation lines (which do not start with an L) can be created by terminating each preceding line with a comma (,).

A + specified as the last element the command will cause the output from the next L command to be appended.

If you specify a heading statement (HDR) which contains direct or indirect references, these will be evaluated (and included in the heading) as the command is processed. Subsequent changes to the source values will have no effect on the heading.

EXAMPLE

```
MV %1 "Quarter 4"
L HDR,"PAGE ",P,(10),T
L 5,"Sales Report for ",%1
```

Output:

```
PAGE 1 10:25:17 10 OCT 1999

Sales Report for Quarter 4
```

JCL M

Marks a destination for a GO F or GO B command.

COMMAND SYNTAX

M

NOTES

The M command is used with the GO B and GO F branch commands to mark the destination of the jump.

As a jCL program is executed it "remembers" the location of the last M command it passes through. When a GO B command is encountered, the program can then branch straight back to the remembered location. Remember that an M command must be executed for it to be remembered.

GO F scans forward from the current location until it finds an M command and then resumes execution from there.

If commands are grouped on a line, the M command must be the first command on the line.

EXAMPLES

See the GO B and GO F commands for examples of usage.

JCL MS

Move the entire content of the secondary input buffer to the primary input buffer.

COMMAND SYNTAX

MS

NOTES

MS copies the entire content of the secondary input buffer and inserts the parameters before the parameter currently pointed to in the primary input buffer. The primary input buffer must be active when MS is executed. After the move, the secondary input buffer will be empty.

Hold file numbers are returned as Entry #n, where "n" is the hold file number.

EXAMPLE

```
001 PQN
002 HSP-ASSIGN HS
003 P
004 HCOPY SALES ABC (P
005 P
006 S10
008 MS
...
```

The COPY command on line 4 creates a print file and the hold file number is written to the SIB. The S10 command on line 6 positions the PIB buffer pointer to parameter 10. The MS command on line 8 moves the contents of the SIB into the PIB starting at the 10th parameter of the PIB.

JCL MV

Copies data between buffers or between buffers and select registers.

COMMAND SYNTAX

MV destination source{,source}...{,*{n}}{,_}

or

MV destination source{*source}...

SYNTAX ELEMENTS

destination is a direct or indirect reference to the destination buffer or select register which is to receive that data.

source is the data you want to copy. Can be a direct or indirect reference to a buffer or select register, or a literal string enclosed in single or double quotes.

,* copies all source parameters starting with the specified parameter. If ***** is the last operand in the source field, the destination buffer or select register will be truncated after the last copied parameter.

,*n copies the number of source parameters specified by **n** starting with the specified parameter. The destination buffer or select register will not be truncated.

,_ specifies that the destination is to be truncated after the source is copied.

***source** specifies the source values are to be concatenated into one field in the destination buffer or select register.

NOTES

If the source is a literal string containing just two double quotes, the destination will be nulled.

If the input buffer (%n) is specified as the destination, it will be selected as the active input buffer and the buffer pointer will left at the beginning of any copied data.

If the field or parameter number in destination is larger than the current number of fields or parameters, intervening null values will be created automatically.

Specify an asterisk (*) as the last character in the source element if you want to copy all the following source buffer parameters. For example:

```
MV &2.2 %1,*
```

will copy the 1st parameter of the PIB to field 2 of file buffer 2. Parameter 2 of the PIB will be copied to field 3 and so on.

If you want to copy a series of parameters, specify the number of additional parameters to be copied after the asterisk. For example:

```
MV &2.2 %1,*3
```

will copy the 1st, 2nd, 3rd and 4th parameters of the PIB into fields 2, 3, 4 and 5 of file buffer 2. The remainder of file buffer 2 will not be changed.

If you specify a series of values as the source, each value will copied to successive locations in the destination. For example:

```
MV %2 "ABC"*&2.1*!1
```

will copy ABC into PIB parameter 2, field 1 of file buffer 2 into PIB parameter 3, and the next value from select register 1 into PIB parameter 4.

Two or more source values separated with an asterisk (*), will be concatenated into a single parameter in the destination. For example:

```
MV %2 "ABC",&2.1,!1
```

will concatenate ABC, field 1 of file buffer 2 and the next value from select register 1, and place the result into PIB parameter 2.

Intervening parameters in the destination can be preserved by using commas as place holders in the source. For example:

```
MV %1 "ABC",,,"DEF"
```

would copy ABC into PIB parameter 1 and DEF into PIB parameter 4. PIB parameters 2 and 3 would not be affected.

EXAMPLE 1

Command	PIB Before	PIB After
MV %5 "XXX"	ABC	ABC^^^^XXX

EXAMPLE 2

Command	PIB Before	PIB After
MV %4 %1	ABC^DEF^GHI	ABC^DEF^GHI^ABC

EXAMPLE 3

PIB contains: QTR^ABC^DEF

Command	POB Before	POB After
MV #3 %1	SORT^SALES^	SORT^SALES^QTR^

EXAMPLE 4

Command	PIB Before	PIB After
MV %1 "AA",,"CC"	XX^BB^YY^ZZ	AA^BB^CC^ZZ

EXAMPLE 5

File buffer 2 contains:

```
000 Key
001 111
002 AAA
003 BBB
```

Command	File Buffer 1 Before	File Buffer 1 After
MV &1.1 &2.2,*	000 KEY1 001 WWW 002 XXX 003 YYY 004 ZZZ	000 KEY1 001 AAA 002 BBB 003 YYY 004 ZZZ

EXAMPLE 6

PIB contains: ABC^DEF^GHI^JKL^MNO

Command	File Buffer 1 Before	File Buffer 1 After
MV &2.1 %3,_	000 Key 001 XXX 002 YYY	000 Key 001 GHI

JCL MVA

Copies a value from the source to the destination buffer and stores it as a multivalue.

COMMAND SYNTAX

MVA destination source

SYNTAX ELEMENTS

destination is a direct or indirect reference to a buffer or select register which is to receive the data.

source is the data to be copied. The source can be a direct or indirect reference to a buffer or select register, or a literal string.

NOTES

New values will be copies to the destination in ascending ASCII sequence.

If a new value already exists in the destination buffer, it will not be copied.

If the source data is multivalued, it will be copied to the destination without modification. This might create duplicate values and invalidate the ascending sequence.

If the destination is the input buffer, the buffer pointer will be left at the beginning of the destination parameter.

EXAMPLE 1

PIB contains: ABC^DEF^GHI

Command	File Buffer 1 Before	File Buffer 1 After
MVA &1.1 %3	000 Key 001 FFF]HHH 002 YYY	000 Key 001 FFF]GHI]HHH YYY

EXAMPLE 2

File buffer 2 contains:

```
000 Key
001 GG]YY
002 AAA
```

Command	File Buffer 1 Before	File Buffer 1 After
MVA &1.1 &2.1	000 Key 001 FFF]HHH 002 YYY	000 Key 001 FFF]GG]YY]HHH YYY

JCL MVD

Deletes a value from a multivalued parameter in the target buffer.

COMMAND SYNTAX

MVD target source

SYNTAX ELEMENTS

target is a direct or indirect reference to a buffer or select register which contains the data to be deleted.

source is the data you want to delete. Can be a literal string, or a direct or indirect reference to a buffer or select register.

NOTES

Values in the target must be in ascending ASCII sequence, otherwise the result of the command will be unpredictable.

If the source does not exist or if the multivalue cannot be matched, the command will no effect, except perhaps to move the buffer pointer.

If the source element exists more than once in the target parameter, only the first occurrence will be deleted.

If the target is the primary input buffer, the buffer pointer will be left at the beginning of the specified target parameter.

EXAMPLE

```
Command        File Buffer 1      File Buffer 1 After
               Before

MVD &1.1 DEF   000 Key            000 Key
               001 AAA]DEF]GHI    001 ABC]GHI
               002 YYY            002 YYY
```

JCL O

Outputs a text string to the terminal.

COMMAND SYNTAX

O{text}{+}

SYNTAX ELEMENTS

text is the text to be displayed.

+ inhibits a NEWLINE at the end of the output and leaves the cursor positioned to the right of the last character displayed. This is often used when prompting for input.

NOTES

The O command has largely been replaced by the T command which also provides cursor positioning and special display features.

If no text is supplied, a blank line will be output.

EXAMPLE 1

Command	Terminal output
OSALES SYSTEM	SALES SYSTEM

EXAMPLE 2

Command	Terminal output
OEnter Password + IP:	Enter Password :

JCL P

Submits the shell command created in the primary output buffer for execution.

COMMAND SYNTAX

P{P}{H}{Ln}{X}{U}{W}

SYNTAX ELEMENTS

P displays the primary and secondary output buffers. In ROS emulation mode, displays the command and prompts to continue. Normally only used for testing or debugging.

H suppresses (hushes) any terminal output that would normally be displayed. The H and Ln options can be combined as PHLn.

Ln sets an execution lock where n represents a lock number from 0 to 255. The lock is after command has been executed. Any other process attempting to set the same lock is forced to wait. The H and Ln options can be combined as PHLn.

X terminates the program after the command is executed. Cannot be used with any other options.

U specifies that the command is to be submitted for execution by UNIX.

W causes the command to behave like PP when used in a non-ROS emulation.

NOTES

When the P command is executed, control is passed to the shell and only returns when the shell process has been completed. After the P command has been executed, both output buffers are cleared and the stack is turned off.

Commands and data in the secondary output buffer are made available to processes which require further input.

If you need to preserve buffer contents when passing control between jCL programs, use the ( ) or [ ] command instead.

If you use the PP variants (PP, PPH, PPLn), the content of both output buffers will be displayed before they are executed and you will be prompted with a question mark (?). Enter:

Y	to continue.
S	to cancel execution but continue the program.
N	to cancel execution and exit the program.

EXAMPLE 1

```
003 HLIST SALES QTR VALUE
004 P
```

Copy the jQL command LIST SALES QTR VALUE to the output buffer and execute it.

EXAMPLE 2

```
003 HCOPY SALES ABC
004 STON
005 H(SALES.HOLD<
006 PP
```

Place the COPY command in the primary output buffer. Turn the stack on. Put the response to the TO: prompt into the secondary input buffer. Issue the PP command to display the contents of the buffers and prompt for input before continuing.

EXAMPLE 3

```
003 Henv | grep EMULATE
...
006 PU
```

Issue the UNIX grep command.

JCL RI

Resets (clears) the primary and secondary input buffers.

COMMAND SYNTAX

RI
RIp
RI(n)

SYNTAX ELEMENTS

p specifies starting parameter from which to clear to the end of the buffer. Can be a number, or a direct or indirect reference to a buffer or select register.

(n) specifies the starting column from which to clear to the end of the buffer. Can be a number, or a direct or indirect reference to a buffer or select register.

NOTES

The RI command clears the entire PIB and SIB.

RIp clears the PIB starting from parameter p and continuing to the end of the buffer.

RI(n) clears the PIB starting from parameter n and continuing to the end of the buffer.

The buffer pointer will be left at the end of the PIB. The primary input buffer becomes the active buffer and the secondary input buffer will be cleared.

EXAMPLE 1

Command	PIB Before	PIB After
RI	ABC^DEF^GHI	

EXAMPLE 2

Command	PIB Before	PIB After
RI3	ABC^DEF^GHI	ABC^DEF

EXAMPLE 3

Command	PIB Before	PIB After
RI(6)	ABC^DEF^GHI	ABC^D

JCL RO

Resets (clears) the active output buffer.

COMMAND SYNTAX

RO

NOTES

The RO command clears the active output buffer.

The buffer pointer is left at the beginning of the buffer.

EXAMPLE 1

Command	POB Before	POB After
STOFF		
RO	ABC^DEF	

EXAMPLE 2

Command	SOB Before	SOB After
STON		
RO	GHI^JKL	

JCL RSUB

Terminates execution of a local subroutine and returns control to a statement line following the GOSUB command that called the subroutine.

COMMAND SYNTAX

RSUB {n}

SYNTAX ELEMENTS

n specifies that control should be returned to the n'th statement line after the calling GOSUB. Can be a number, or a direct or indirect reference to a buffer or select register.

NOTES

If n is not specified, control will return to the statement immediately following the calling GOSUB.

An RSUB without a corresponding GOSUB will ignored.

EXAMPLE 1

```
010 GOSUB 1001
011 ...
012 ...
.
051 1001 T "Press <CR> to continue...",+
052 S10
052 IP?
053 RSUB
```

The RSUB command on line 53 will return control to line 11.

EXAMPLE 2

```
010 GOSUB 1001
011 ...
012 ...
.
051 1001 T "Press <CR> to continue...",+
052 S10
052 IP?
053 RSUB 2
```

The RSUB command on line 53 will return control to line 12.

JCL RTN

Terminates execution of an external subroutine and returns control to a statement following the [ ] command that called the subroutine.

COMMAND SYNTAX

RTN{n}

SYNTAX ELEMENTS

n specifies that control should be returned to the n'th statement line after the calling [ ] command. Can be a number, or a direct or indirect reference to a buffer or select register.

NOTES

If n is not specified, control will return to the statement immediately following the calling [ ] command.

A RTN without a corresponding [ ] command will terminate the program.

EXAMPLE 1

MENU	**MENU2**
051 [SUBS MENU2]	**066 RTN**
052	
053	

jCL program MENU calls MENU2 from line 51. When MENU2 reaches line 66, control will be returned to MENU at line 52.

EXAMPLE 2

MENU	MENU2
051 [SUBS MENU2]	066 RTN 2
052	
053	

jCL program MENU calls MENU2 from line 51. When MENU2 reaches line 66, control will be returned to MENU at line 53.

JCL S

Positions the primary input buffer pointer to a specified parameter or column, and selects the primary input buffer as active.

COMMAND SYNTAX

Sp
S(n)

SYNTAX ELEMENTS

p specifies the parameter to which the buffer pointer should be set. Can be a number, or a direct or indirect reference to a buffer or select register.

(n) specifies the starting column to which the buffer pointer should be set. Can be a number, or a direct or indirect reference to a buffer or select register.

NOTES

Sp sets the buffer pointer to the field mark preceding parameter p.

If the specified column or parameter number is less than 2, the buffer pointer is placed at the beginning of the active input buffer.

If the specified column or parameter number is beyond the end of the buffer, the buffer pointer will positioned at the end of the buffer. Use the MV command to create parameters beyond the current end of the buffer.

All data in the secondary input buffer will be cleared.

EXAMPLE 1

Command	PIB Before	PIB After
S3	ABC^DEF^GHI	ABC^DEF^GHI

EXAMPLE 2

Command	PIB Before	PIB After
S(4)	12345^ABC	12345^ABC

JCL STOFF

Selects the primary output buffer as the active output buffer.

COMMAND SYNTAX

STOFF
ST OFF

NOTES

The stack can be turned on (STON) or off (STOFF) at any point within a jCL program.

At the start of a jCL program or after execution of an RO or P command, both output buffers will be empty, and the stack will be off.

When the stack is off, H commands will place their data in the primary output buffer.

JCL STON

Selects the secondary output buffer as the active output buffer.

COMMAND SYNTAX

```
STON
ST ON
```

NOTES

The STON command selects the SOB as the active output buffer. With the stack turned on, all data moved to an output buffer with an MV, H or A command will be placed in the secondary output buffer.

The stack is often used to feed responses to interactive processes. It can also be used to store a series of commands that you might need for example when you are dealing with select lists. For example, do a GET-LIST, followed by a SORT-LIST and then run a jBC program.

Typically, you would create the command necessary to start the external job stream in the primary output buffer. Next you would turn the stack on and store all the necessary responses (or commands) to the external process. When you issue the P command to execute the external program, instead of taking it"s input from the terminal, the program will be fed directly from the stack.

Terminated successive responses in the stack, with a less-than-character (<) to represent a RETURN key depression. A single response or the last response in the stack does not require the less than character (<) because a RETURN is generated by the P command.

EXAMPLE

Command	POB Before	POB After
STON	COPY^SALES^ABC	COPY^SALES^ABC
H(SALES.HOLD		
	SOB Before	SOB After
		(SALES.HOLD<

JCL T

Produces formatted terminal output.

COMMAND SYNTAX

T element{, element}

SYNTAX ELEMENTS

element is literal text, a reference or a formatting instruction:

"text"	Outputs the specified text. The text must be enclosed in double quotes.
r{;input;}	Outputs the value obtained by the direct or indirect reference to a buffer or select register specified by r. An optional jQL input conversion can be applied to the value prior to output.
r{:output:}	Outputs the value obtained by the direct or indirect reference to a buffer or select register specified by r. An optional jQL output conversion can be applied to the value prior to output.
(c,r)	Sets the cursor to the column c and row r. Can be direct or indirect buffer references.
(c)	Sets the cursor to the column c in the current row.
*cn	Outputs the character c n number of times. Value n can be a direct or indirect reference to a buffer or select register.
(-n)	Provides terminal independent cursor control or video effects.
+	Outputs carriage return/line feed at the end of the output line. On ROS emulations, this clause will inhibit the carriage return/line feed at the end of the output line.
B	Sounds terminal bell.
C	Clears the screen (outputs top-of-form).
D	One second delay. Often used when outputting error messages.
Ir	Converts the integer r (0 to 255) into its equivalent ASCII character. r can be a direct or indirect reference to a buffer or select register that contains the integer.
L	Terminates a loop started with a T element. The elements between the T and L are executed three times.

Sn	Outputs the number of spaces specified by n. The value n can be a direct or indirect reference to a buffer or select register than contains the number of spaces.
T	Marks the top of a loop. The loop is terminated by the L element. The elements between T and L are executed three times.
U	Moves the cursor up one line.
Xr	Converts the hex value r (x"00" to x"FF") into its equivalent ASCII character. The value r can be a direct or indirect reference to a buffer or select register that contains the hex value.

NOTES

The T command must be followed by a single space and each element must be separated by a comma. Continuation lines (which do not start with a T) can be created by ending the preceding line with a comma.

TERMINAL INDEPENDENT CURSOR CONTROL

Terminal independent cursor control is available using the same table of negative numbers as used by the jBC @ command. See the jBC @ command for more details.

EXAMPLE 1

Commands	Terminal Output
MV %1 "99"	
T C, "%1 = ",%1	clear screen
	%1 = 99

EXAMPLE 2

Commands	Terminal Output
T "Enter Password :",+	
IP %3	Enter Password : _

EXAMPLE 3

```
Command          Terminal Output

T *=6            ======
```

EXAMPLE 4

```
Commands             Terminal Output

MV %1 "9873"

T                    DATE: 11 JAN 95
"DATE:",S2,%1:D2:
```

EXAMPLE 5

```
Command                         Terminal Output

T                               ERROR bell (flashing)
(0,10),T,(0),"ERROR",B,D,(
O),S5,D,L
```

EXAMPLE 6

```
Commands                        Terminal Output

T X1B,"J",+                     erase to end of screen

T I27,I74,+                     erase to end of screen

T (-3),+                        erase to end of screen
```

JCL TR

Traces jCL program execution and displays each command (source line) before it is executed.

COMMAND SYNTAX

TR {ON}
TR OFF

NOTES

The TR command will help you to test and debug your jCL programs.

TR or TR ON turns the trace on. TR OFF turns the trace off.

You can easily provide a general purpose trace program by creating a jCL program called TRACE in the file defined by the JEDIFILENAME_MD environment variable. The TRACE program should look like this:

```
TRACE
001 PQN
002 TR ON
003 (%1 %2)
```

Run the TRACE program like this:

```
TRACE jcl_file jcl_program
```

If the target program relies on the initial content of the PIB, you will have to insert a line into the program like this:

```
002 MV %1 %2,*
```

to move the PIB parameters to their required positions - otherwise the word TRACE will appear in %1 and the jCL program name (normally in %1) will be in %2.

EXAMPLE

```
001 PQN
002 TR
003 MV %21 ","," "
004 MV %98 1,A
005 S21
006 U147F
007 T MTS
008 S23
009 U147F
010 D D2
011 T "Today"s date is ", %23
012 T "the time is ",%21
013 TR OFF
```

will output:

```
MV %21 ","," "
MV %98 1,A
S21
U147F
S23
U147F
T "Today"s date is ", %23
Today"s date is 01/01/95
T "the time is ",%21
the time is 19:30:32
```

Line 2 turns the trace on. Line 13 turns it off.

JCL U

Execute a "user exit" from a jCL program.

COMMAND SYNTAX

Unxxx

SYNTAX ELEMENTS

n represents the user exit entry point

xxx is the id of the user exit.

NOTES

User exits are user written functions which are used to manipulate buffers and perform tasks beyond the scope of the standard jCL commands.

See the Time and Date topic and the TR command for more examples of "standard" user exits.

EXAMPLE 1

```
012 U31AD
```

Returns the current port number.

EXAMPLE 2

```
012 U307A
013 300
```

Causes the process to "sleep" for 300 seconds.

EXAMPLE 3

```
003 U407A
004 12:0
```

Causes the process to "sleep" until 12:10.

JCL X

Halts execution of the program and returns control to the shell.

COMMAND SYNTAX

X{text}{+}

SYNTAX ELEMENTS

text is any text to be displayed on exit.

+ suppress a NEWLINE at exit or after text output.

NOTES

The X command returns control directly to the shell.

EXAMPLE 1

```
F-OPEN 1 SALES
XCannot Open SALES file!
```

The X command stops execution of the program if the file SALES cannot be opened, and displays a suitable message.

LIST PROCESSING COMMANDS

The two shell commands associated with jCL which permit list handling are the PQ-SELECT and PQ-RESELECT commands. See the chapter on List Processing for more information on lists.

PQ-RESELECT

Executed from a jCL program, resets the pointer of a specified select register to the beginning of the list of record keys.

COMMAND SYNTAX

```
PQ-RESELECT register-number
```

SYNTAX ELEMENTS

register-number is the number (1 TO 5) of the select register to be reset.

NOTES

This command is executed from the primary output buffer to reset the pointer of a specified select register back to the beginning of the list.

Each time you use the "!" reference to retrieve a value from the list, the value is not destroyed. The pointer is simply advanced to the next parameter in the list. PQ-RESELECT resets the pointer back to the beginning of the list so that you can perform another pass.

You cannot execute PQ-RESELECT directly from the shell.

EXAMPLE

```
HSELECT SALES
STON
HPQ-SELECT 1
PH
MV %1 !1
IF # %1 XNo records selected
HPQ-RESELECT 1
PH
10 MV %1 !1
IF # %1 XFinished
...
GO 10
```

PQ-SELECT

Executed from a jCL program, loads a list of keys into a select register.

COMMAND SYNTAX

PQ-SELECT register-number

SYNTAX ELEMENTS

register-number is the number of the select register (1 to 5) in which the keys are to be placed.

NOTES

To use PQ-SELECT you must first construct a list by using one of the list processing commands such as SELECT, SSELECT, QSELECT, BSELECT, GET-LIST, FORM-LIST, SEARCH or ESEARCH. Put the PQ-SELECT command in the stack so that it will be processed as part of the external job stream when the required list is active.

Retrieve the list elements one at a time, using a "!n" direct or an indirect reference.

You cannot execute PQ-SELECT directly from the shell.

EXAMPLE

```
001 PQN
002 HSSELECT SALES
003 STON
004 HPQ-SELECT 1
005 P
006 10 MV %1 !1
007 IF # %1 X Done
008 T %1
009 GO 10
```

This example selects all the records in the SALES file, loads the record-list into select register 1 and displays the keys one at a time.

PQ AND PQN DIFFERENCES

The first line of a jCL program defines the program as one of two basic types, a PQ or a PQN style program.

Wherever possible, you should use the PQN style.

There are several differences between the two styles, as outlined in the following topics.

DELIMITERS

PQ-style programs usually use a blank as the delimiter between parameters in the buffers. PQN-style programs usually use a field mark.

PQN allows parameters to be null or contain blanks and more closely mirrors the native record structure.

PQ commands are still supported for compatibility but you should use the functionally superior PQN commands in new or revised jCL programs.

When pointers are moved, PQ commands will generally leave the pointer at the first character of a parameter. PQN commands will generally leave the pointer at a field mark.

Commands affected by this difference are A, B, BO, F, H, IH and IP.

BUFFERS

Buffer referencing, file buffers and select registers are only available with PQN commands.

COMMANDS

These commands are only used in PQN-style jCL programs:

F;, F-KLOSE, F-WRITE, L, MVD
F-CLEAR, F-OPEN, FB, MS,
F-DELETE, F-READ, IBH, MV,
F-FREE, F-UREAD, IBP, MVA,

These commands are functionally equivalent in both PQ and PQN-style programs:

( ), G, IF E, RSUB, U
[], GO B, IFN, RTN, X
+, GO F, M, S,
-, GOSUB, P, STOFF,
C, IF, RI, STON,

jQL

Overview	Overview of jQL
Defaults	Defaults used by jQL
Conversion Processing	Conversion codes supported by jQL
Subroutines	Calling subroutines from jQL
File Definitions	Discussion of file definition records.
I-TYPES	Discussion of I-TYPE file definition records.

BSELECT	Creates a select list based on output specifications.
COUNT	Counts records.
EDELETE	Deletes selected records from a file.
ESEARCH	Searches records for specified strings.
I-DUMP	Outputs the entire contents of items.
LIST	Generates a formatted report of records and fields from a specified file.
LIST-LABEL	Outputs data in a format suitable for producing labels.
LISTDICT	Generates a report of data definition records.
REFORMAT	Generates a formatted report of records and fields to a file or tape.
SELECT	Generates a list of record keys or specified fields based on the selection criteria specified.
SORT	Generates a sorted formatted report of records and fields from a specified file.
SORT-LABEL	Outputs data in a format suitable for producing labels.
SREFORMAT	Generates a sorted formatted report of records and fields to a file or tape.
SSELECT	Generates an sorted list of record keys or specified fields based on the selection criteria specified.

OVERVIEW

The jBASE Query Language (jQL) is a powerful and easy to use facility which allows you to retrieve data from the database in a structured manner and to present the data in a flexible and easily understood format.

The language is characterized by the use of intuitive commands that resemble everyday English language commands. For example, if you wanted to review a particular set of sales figures you might phrase your request like this:

"Show me the sales figures for January sorted in date order."

This request would translate into a jQL command like this:

```
LIST SALES WITH MONTH = "JANUARY" BY DATE
```

By using the jQL command LIST with a file named SALES and your predefined data definition records such as MONTH and DATE, you can construct complex ad-hoc reports directly from the command line interface.

jQL contains a rich range commands for listing, sorting, selecting and controlling the presentation of your data.

jQL is a safe language for end users. With the exception of the EDELETE command, jQL will not alter the contents of the source data files.

All jQL command sentences begin with a verb-like command such as LIST or SELECT, followed by a file name such as SALES or PERSONNEL, and then a series of qualifiers and modifiers with which you control elements such as eligible data, report formatting, any totals that you want to appear and so on.

Most data files on the system will have two storage areas assigned, one for the data (the data section) and one for the data definition records (the dictionary section). Some files might be single level and others might have multiple data sections. See the File Management chapter of the System Administrators Guide for more details.

Typically, all of the data fields in a file will be defined by data definition records kept in the dictionary portion of the file. These data definition records do not have to exist - you can use defaults provided in the environment variables or even the dictionaries of other files - but where you need to manipulate say dates (which are held in internal format), or to join data that is held in different files (perhaps even on remote systems), you will find that one or more definition records will be required for each data field.

The data definition records are simple to create and maintain. They allow you to specify for example, the position of the data in a record (its field number), a narrative to be used as a column heading, any input or output conversions required (such as for dates), the data type (left or right justified, or text that will break on word boundaries) and a column width which will be used in the reports.

Input and output conversion codes can also be used to manipulate the data by performing mathematical functions, concatenating fields, or by extracting specific data from the field.

jQL COMMAND SENTENCE CONSTRUCTION

A jQL command sentence must contain at least a command and a file name. The command specifies the process to be performed and the filename indicates the initial data source.

Optional clauses can be added to refine the basic command. You can use clauses to control the range of eligible record keys, define selection and sorting criteria, or to specify the format of the output, and so on.

COMMAND SYNTAX

jQL-command file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier} {output-specification} {format-specification} {(options}

SYNTAX ELEMENTS

jQL-command is one of the verb-like commands detailed later. Most commands will accept any or all of the optional clauses.

file-specifier identifies the main data file to be processed. Usually the data section of a file, but could be a dictionary or a secondary data area.

record-list defines which records will be eligible for processing. Comprises an explicit list of record keys or record selection clauses. An explicit list comprises one or more record keys enclosed in single or double quotes. A selection clause uses value strings enclosed in single or double quotes and has at least one relational operator. If no record list is supplied, all records in the file will be eligible for processing unless an "implicit" record list is provided by preceding the command with a selection command such as GET-LIST or SELECT.

selection-criteria qualify the records to be processed. Comprises a selection connective (WITH or IF) followed by a field name. Field names can be followed by relational operators and value strings enclosed in double quotes.

sort-criteria specify the order in which data is to be listed. Comprises a sort modifier, such as **BY** or **BY-DSND**, followed by a field name. Can also be used to "explode" a report by sorting lines corresponding to multivalued fields by value, and to limit the output of values (see output-specification).

USING file-specifier defines an alternate file to be used as the dictionary. There is no rest DBL-SPC, that define the overall format of the report.

options comprise letters enclosed in parentheses which modify the action of the command - to redirect output to a printer for example.

NOTES

Any element of a jQL command sentence (with the exception of the command and filename) can be substituted with a macro - see later.

When the **REQUIRE-SELECT** modifier is included in a jQL sentence it verifies that a select list is active before processing the sentence.

RESERVED WORDS AND SYMBOLS

The following words and symbols have specific meanings when used in a jQL sentence. They should only be used as described later in this chapter.

!	#	&		<	<=	=	>	>=
A	AFTER	AN	AND	ANY	ARE			
BEFORE	BETWEEN	BREAK-ON	BSELECT	BY	BY-DSND	BY-EXP	BY-EXP-DSND	
CAPTION	CHECK-SUM	COL-HDR-SUPP	COUNT					
DATA	DBL-SPC	DET-SUPP	DICT					
EACH	EDELETE	EQ	ESEARCH	EVERY				
FILE	FOR	FOOTING						
GE	GRAND-TOTAL	GT						
HDR-	HEADER	HEADIN						

SUPP		G					
ID-SUPP	IF	IN					
LE	LIST	LIST-INDEX	LIST-LABEL	LPTR	LT		
NE	NO	NOPAGE	NOT				
OF	ONLY	OR					
PAGE	PG						
REFORMAT	REQUIRE-SELECT						
SELECT	SORT	SORT-LABEL	SREFORMAT	SSELECT	SUBVALUE	SUM	SUPP
T-DUMP	T-LOAD	TAPE	THE	TOTAL			
USING							
VALUE							
WITH	WITHIN	WITHOUT					

ENTERING A jQL COMMAND SENTENCE

A jQL command sentence is entered at the shell in response to a command prompt (:) or a select prompt (>). The select prompt is displayed if an implicit record list has been created by a command such as SELECT or GET-LIST whilst in jSHELL. Each sentence must start with a jQL command and can be of any length. Having constructed your sentence, you submit it for processing by pressing the RETURN key.

If you enter an invalid command, the system will reject it and display an appropriate error message.

EXAMPLE

```
SORT SALES WITH PART.NO = "ABC]" BY POSTCODE CUST.NAME POSTCODE
TOTAL VALUE DBL-SPC HDR-SUPP (P
```

where:

SORT	is the jQL command.
SALES	is the filename.
WITH PART.NO = "ABC]"	is the selection criterion. Select all records which contain a part number that

	starts with ABC.
BY POSTCODE	is the sort criterion.
CUST.NAME POSTCODE TOTAL VALUE	is the output specification. Column 1 will contain the key of the SALES file, column 2 will contain the customer name and column 3 will contain the POSTCODE. Column 4 will contain VALUE and will be totaled at the end of the report.
DBL-SPC HDR-SUPP	are the format specifications. Double-space the lines and suppress the automatic header.
(P	is an option. Redirect output to the system printer, rather than to the terminal.
PART.NO, CUST.NAME, POSTCODE, VALUE	are references to data definition records which are defined in the dictionary level of the SALES file.

DEFAULT DATA DEFINITION RECORDS

When you issue a jQL command that does not contain specific references to data definition records, and you do not suppress output of the report detail, the system will attempt to locate any default data definition records that you may have set up.

For example, if you issue the command "LIST SALES", the system will look in the dictionary of the SALES file for a data definition record named "1". If it finds "1", this will become the default output for column two. The system will then look for a data definition record named "2" and so until the next data definition record is not found. If "1" is not found in the file dictionary, the system will search the default dictionaries for the same sequence of data definition records.

When you issue a jQL command that does contain specific references to data definition records, the system will first attempt to locate each data definition record in the dictionary of the file (or in the file specified in a USING clause). If the data definition is not found in the dictionary (or the file specified in a USING clause) the system look in the files specified in the JBCDEFDICTS environment variable but only if it as been assigned. If JBCDEFDICTS has not been assigned, the system will look for the data definition in the file defined by the JEDIFILENAME_MD environment variable.

For example, if you issue the command "LIST SALES VALUE", the system will look in the dictionary of the SALES file for a data definition record named "VALUE". If it cannot find "VALUE" in the file dictionary, the system will look in the files specified in the JBCDEFDICTS environment variable (if JBCDEFDICTS is assigned) and then in the file specified by the JEDIFILENAME_MD environment variable (if JBCDEFDICTS is not assigned).

In this way, you can set up data-specific, file-specific or account-specific defaults to be used with any jQL command.

jQL OUTPUT (REPORTS)

By default, output from a jQL command will be displayed on your terminal, in columnar format, with a pause at the end of each page (screenful).

OUTPUT DEVICE

You can redirect the output to a printer (or the currently-assigned Spooler device) by using the LPTR format specifier or the P option.

REPORT LAYOUT

If the columnar report will not fit in the current page width of the output device, it will be output in "non-columnar" format where each field of each record occupies one row on the page.

PAGING

If the report is displayed at the terminal and extends over more than one screen, press RETURN to view the next screen. To exit the report without displaying any remaining screens, press <Control X> or "q".

JQL Defaults

Id Length

A default data section dictionary item can be used to specify the length of the record id field, e.g. for default data section named FRED, create a dictionary record named FRED in the dictionary section of file FRED.

Alternatively a global default record id length can be configured by the JBCIDLEN environment variable otherwise default to 14 characters.

```
export JBCIDLEN=10 ( Unix)
set JBCIDLEN=10 ( NT/Win95)
```

Alternative Dictionaries

In order to enable dictionary definition records from multiple dictionary files to be used *without* specifying the **USING** connective, an environment variable JBCDEFDICTS can be configured for multiple dictionary files.

```
export JBCDEFDICTS=UserDictFile;GeneralDictFile (Unix)
set JBCDEFDICTS=CustomDictFile:GeneralDictFile (Windows)
```

The order in which the dictionaries are searched for dictionary definitions is as follows:

1. Dictionary of the file being queried
2. MD/VOC (see JEDIFILENAME_MD)
3. Files specified by JBCDEFDICTS

JQL BASIC SUBROUTINES

jBASE jQL enables users to call Basic subroutines from within correlatives and conversions. There are two flavors of subroutine and each required a different include file.

For Advanced Pick subroutines the developer must include the following header file from the "include" subdirectory in the jBASE release directory.

qbasiccommonpick

For Sequoia subroutines the developer must include the following header file from the "include" subdirectory in the jBASE release directory.

qbasiccommonseq

FILE DEFINITIONS

This section starts with a review of file definition records and discusses how the content of these records can affect the output from jQL commands - particularly with reference to sublists.

File definition records contain information that is used when formatting reports. For example, should the key field of a file be left or right justified, and how wide a column should it occupy. You can also define sublist codes to tell the system that a particular field is multivalued.

RECORD STRUCTURE

The fields of a file definition record that affect jQL reports are:

Field 7	Conversion code for key, if required. For date, time, etc.
Field 8	V code to notify a multivalued (sublist) field, if required. See Sublists - V Code.
Field 9	Justification for key. Can be one of the following (see Data Definition Records):

L	Left justified
R	Right justified
T	Text
U	Unlimited

Field 10	Column width for key. Default is 14 characters.

SUBLISTS - V CODE

File records which contain sublists can be accessed with the COUNT and LIST commands and the WITHIN modifier. For the commands and the modifier to function properly, you must include the V processing code in field 8 of the file definition record. See the File Specifiers topic in the jQL Command Sentence Construction topic for more details.

COMMAND SYNTAX

```
V;;field-no
```

SYNTAX ELEMENTS

field-no is the number of the field which contains the sublist.

EXAMPLE

Consider the STOCK file used by a camera factory where each data record can represent either an assembly or a component part.

Take as an example the record set that defines a simple camera assembly. The data records contain the following data:

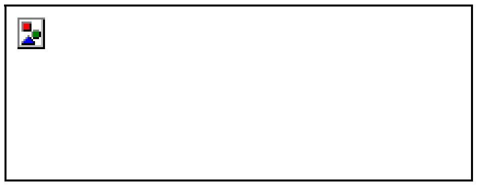
Key	A1	Key	A21
001	CAMERA ASSY	001	LENS ASSY
002	A21]A22]A23	002	A210]A211
003	10	003	15

Key	A22	Key	A23
001	BODY	001	SHUT ASSY
002		002	A230]A231
003	10	003	11

Key	A210	Key	A211
001	OPTICS	001	BARREL
002		002	
003	19	003	21

Key	A230	Key	A231
001	IRIS MECH	001	IRIS HOUSNG
002		002	
003	13	003	14

The key is the part number, field 1 contains the description, field 2 is a multivalued list of components that go to make up the part, and field 3 is the current stock level.



Record A1 represents assembled cameras. It points to the sub-assemblies (A21, A22 and A23) that are used to make each camera. The sub-assemblies in turn point to their component parts; A21 points to A210 and A211, A22 does not have any components, and A23 points to A230.

Having established the logical data relationships, we now need to ensure that the system understands that field 2 is a multivalued sublist. We do this by updating field 8 in the file definition record to read "V;;2",
like this:

```
STOCK
001 D
002
003
004
005
006
007
008 V;;2
009 L
010 10
```

Now all we need to do is to create three data definition records in the dictionary of STOCK - one for each field. We will name them DESC, COMPONENTS, and QTY.

The final step is to issue a COUNT or LIST command which uses the WITHIN modifier:

```
:LIST WITHIN STOCK "A1" DESC COMPONENTS QTY

PAGE 1 Time Date

LEVEL STOCK Description.... Components Qty

1     A1    CAMERA ASSY     A21         10
                            A22
                            A23
2     A21   LENS ASSY       A210        15
                            A211
3     A210  OPTICS                      19
3     A211  BARREL                      21
2     A22   BODY                        10
2     A23   SHUTTER ASSY    A230        11
                            A231
3     A230  IRIS MECH                   13
3     A231  FILM MECH                   14

8 RECORDS LISTED
```

DATA DEFINITION RECORDS

Data definition records (sometimes known as field definition records) define the characteristics of each field in a data file. They specify the output format and the type of processing required to generate each column of a jQL report.

Data definition records can be used to:

- Specify default output.
- Associate field names with field numbers (column headings).
- Perform output formatting.
- Calculate new values based on the source data
- Perform processing via conversion codes.

Although they are normally used to define a single physical field in a file, the data definition records can also be used for more complex operations. For example:

- to "join" or derive data from other fields or files.
- to verify the presence (or absence) of records in other files.
- to format their output in the most easily understood manner (to convert numeric 0 and 1 flags to "Yes" or "No", for example, or to output text like "Overdue" if one date field is older than another).
- to generate statistical data like record sizes or counters.

The data definition records are usually located in the dictionary of the data file (but not always - see the USING Clause and the Default Output Specification topics). You can set up any number of data definition records. Often, there are several definitions for each field, each one used by a different set of reports which have different output requirements.

You associate the data definition record with a particular field in the data file by specifying the target field"s FMC (field-mark count) in field 2 of the data definition record. The FMC refers to (points to) the field number (also known as the line number) of the data within the records of the data file.

RECORD LAYOUT

All data definition records are defined in the same way:

Field|Description

1 D/CODE|Defines the record as a data definition record. Must be one of the following codes:

A marks a normal data definition record.

S obsolete but still supported. Was like the A type, but suppressed default column headings when field 3 was blank. Replaced by the A type with a backslash in field 3 to defeat heading.

X forces the record to be ignored if selected as part of a default set of data definitions. Can only be used when explicitly named. See Default Output Specification later.

2 FMC (field-mark count)|A field number or special FMC (see Special Field-mark Counts for more details). A field number refers to the corresponding field (or line) in a record.

The special FMC codes are:

0	Refers to the record key - field number 0 (zero).
9998	Ordinal number of record at output (used for numbering or counting).
9999	Size of the record in bytes (excluding the key).

3 Column heading| Can be heading text, null, or a backslash followed by text. If more characters are entered here than the width in field 10 allows for, the width will be increased to accommodate the heading text (this field wins). Column headings are not displayed if the statement uses the COL-HDR-SUPP output modifier or the "C" option. Heading text is used as the column heading. If the text is less than the column width, it will be padded with dots. Use spaces to produce a blank heading. Value marks, (ctrl]), can be used as NEWLINE characters to place the following text on a new line. If this field is left null, the key of the data definition record will be used as the column heading. Text following a backslash "\" character will be used as the column heading. If nothing follows the backslash, the column heading will be null.

4 - 6|Not used.

7 Input/Output conversion codes|Used for processing the data after sort and selection but before output. See Conversion Codes.

Multiple conversion codes, separated by a value marks, will be processed from left to right.

8 Pre-process conversion codes|Used for processing the data before sort and selection and before field 7 codes. See Conversion Codes later.

Multiple conversion codes, separated by a value marks, will be processed from left to right.

9 Format|Specifies the layout of the data within the column. Can be any of the following:

L Left justified. If the data exceeds the column width specified in field 10, the data is broken at column width without respect to blank spaces.
R Right justify. If the data exceeds the column width specified in field 10, the data will be truncated.
T Text. Word wrap - like L (left justified) but, where possible, lines will be broken at the blank space between words.
U Unlimited. Data is output on one line ignoring column boundaries.

10 Width|Numeric value specifying the column width. If the number of characters in field 3 (the heading) is greater than the number entered here, the number of characters in field 3 will be used.

SPECIAL FIELD-MARK COUNTS

Three special FMCs (field-mark counts) are recognised: 0, 9998 and 9999.

FMC 0 - RECORD KEY

Setting field 2 of the data definition record to 0 (zero), causes the system to work with the record key. In this way, you could set up a data definition record which would allow a the record keys to be output in a column other than the first, and to use any column heading.

Typically, you would also use the ID-SUPP modifier or the I command option to suppress output of the record key in the first column.

FMC 9998 - RECORD COUNT/NI OPERAND

Setting field 2 of the data definition record to 9998, causes the system to return a record (or line) count equal to the number of records output so far in the report.

You could also use function operators within an A or F conversion code in field 7 or 8 of the data definition record to achieve the same result. Function code operand NI yields the same value as an FMC of 9998.

FMC 9999 - RECORD SIZE/NL OPERAND

Setting field 2 of the data definition record to 9999, causes the system to return the record size in bytes. The size does not include the key but does include all field marks within the record.

You could also use function operators within an A or F conversion code in field 7 or 8 of the data definition record to achieve the same result. Function code operand NL yields the same value as an FMC of 9999.

DEFAULT OUTPUT SPECIFICATION

Default output specifications work in two ways, depending on whether the default definitions are explicit or implicit.

EXPLICIT DEFAULTS

If you specify a data definition record to be used for output, the system will search for it first in the implied dictionary (or the dictionary specified in a USING clause). If the data definition is not found in the implied dictionary the system look in the files specified in the JBCDEFDICTS environment variable but only if it as been assigned. If JBCDEFDICTS has not been assigned, the system will look for the data definition in the file defined by the JEDIFILENAME_MD environment variable.

IMPLICIT DEFAULTS

If you do not explicitly specify any data definition records to be used for output, the system will search for a predefined series of records based on the search criteria outlined in the preceding section.

You can therefore set up a series of data definition records that the system will use if a jQL command sentence does not include any explicit output fields.

These "default" records must be named in a numeric sequence starting at 1 (1, 2, 3, and so on). The fields that these records define will be output in the same sequence as the keys but they do not need to follow the same sequence as the fields in the file.

When a jQL command sentence with no explicit output fields is issued, the system first looks in the dictionary for a data definition record named 1, then for a record named 2, then 3, and so on until it fails to find a record with the next number. If the record has a D/CODE of A, it will be used. If the record has a D/CODE of X, it will be ignored, but it will not break the sequence.

A record with a D/CODE of X will only be skipped if it was found as the result of a search for defaults. Under normal circumstances it can be used in the same way as any other data definition record.

This means that when you first set up a series of "default" data definition records, you should put an A in the D/CODE field of each. If you subsequently need to remove one from the sequence, you can simply change the D/CODE field to an X. This way you do not break the sequence or have to copy the remaining "default" records to new names in order to fill the gap.

A data definition record with a number for a key can still be used in the same way as any other data definition record.

PREDEFINED DATA DEFINITION RECORDS

Some predefined data definition records are automatically available so that, if appropriate data definition records are not included in a file"s dictionary, a report can still be generated. These records do not physically exist on the system but are recognised when used in a jQL command sentence.

The predefined data definition records are named *A0 to *Annn. The numeric portion of the key corresponds to the position of the field they report on and the column heading will be the same as the DDR name.

I-TYPES

The jBASE jQL processor supports I-TYPES as imported from PRIME or Universe.

The jBASE query language, jQL, has been enhanced to support I and D type attribute definition records.

Formats

I-TYPE
001 I
002 Expression
003 Conversion
004 Header
005 Format
006 - 016 Reserved

D-TYPE
001 D
002 AttributeNo
003 Conversion
004 Header
005 Format
006 - 016 Reserved

Expression
This can be one or more of the following types:

Type	Description
Dictionary Id	e.g. Attribute definition S/A/D/I type
@Variables	e.g. @RECORD, @ID, @USERNO, @DATE, @TIME
Functions	e.g. TRANS(File, Item, Attr, Code)
User Subroutines	e.g. MYSUB(param1, param2, param256)
Conditionals	e.g. IF X = Y THEN @RECORD ELSE ""
String Extraction	e.g. Expression[6,4]

Multiple expressions can be defined within the same I-TYPE. e.g.

Expression ; Expression ;

Expressions can be parenthesized, contain numeric constants, string literals, enclosed in single or double quotes, and extended operators such as EQ, NE, LE, GT, CAT, AND, OR, MATCHES.

USER SUBROUTINES

Additional functionality can be added by calling user written basic subroutines. The user subroutines should be compiled and cataloged and the library location added to the library path or JBCOBECTLIST environment variables.

The first parameter of the called routine is the result parameter. This parameter is used as the evaluated value of the subroutine e.g.

```
FRED
001 SUBROUTINE FRED(Result, Param1)
002 IF Param1 > 100 THEN Result = 1 ELSE Result = 0
003 RETURN
```

Subroutines can be called from an I-TYPE by one or other of the following formats.

```
FRED(param1 {,param2_}) or SUBR("FRED",param1 {,param2_})
```

Conversion
The Conversion attribute provides support for normal queries output conversions. E.g. D2, MT, F;, TFile etc.

Header
This attribute specifies the column heading text to be displayed.

Format
The format attribute can be specified as follows:
Length {Padding} Justification { Conversion } { Mask }
Where :
Length - The display column length.
Padding - Any character except L,R,U or T. Default space.
Justification - L Left, R Right, T Text, U Unlimited

n	Number of digits after decimal point.
$	Precede with current currency sign.
,	Insert thousandths separator every third digit.
Z	Suppress leading zeroes.
Mask	Output pattern. e.g. ##-###-##

Note: Spurious trailing spaces can give Invalid Conversion errors.

ICOMP

The first time an I-TYPE is used in a query, i.e. jQL command, the expression attribute will be "compiled", to produce internal op codes and parameter definitions. This mechanism provides greater efficiency at run time. However to ensure that all I-TYPE definitions are compiled, rather than on a ad hoc basis, a utility, ICOMP, has been provided.

Called as:
ICOMP {DICT} FileName {RecordList | * }

Where:
FileName is the name of file to convert.
RecordList is the list of Record identifiers.

Note
ICOMP will always attempt to convert the dictionary section of a file. If RecordList is omitted, all I-TYPE definitions will be compiled. ICOMP will also respect a preceding SELECT list.

jQL Conversion Codes

Conversion	**Processing**
A	Algebraic functions.
B	Subroutine call.
C	Concatenation.
D	Internal and external dates.
D1 and D2	Associates controlling and dependent fields.
F	Mathematical functions.
G	Group extract.
L	Length.
MC	Mask character.
MD	Mask decimal.
MK	Mask metric.
ML and MR	Mask with justification.
MP	Mask packed decimal.
MS	Mask Sequence.
MT	Mask time.
P	Pattern match.
R	Range check.
S	Substitution.
T	Text extraction.
TFILE	File translation.
U	User exit.
W	Timestamps.
JBCUserConversions	How to create user-defined conversion codes

A CONVERSION

A codes provide many powerful features. These include arithmetic, relational, logical, and concatenation operators, the ability to reference fields by name or FMC, the capability to use other data definition records as functions that return a value, and the ability to modify report data by using format codes.

The A code also allows you to handle the data recursively, or "nest" one A code expression inside another.

SYNTAX SUMMARY

The A code function uses an algebraic format. There are two forms of the A code:

- A uses only the integer parts of stored numbers unless a scaling factor is included.
- AE handles extended numbers. Uses both integer and fractional parts of stored numbers.

COMMAND SYNTAX

A{n}{;expression}
AE;expression

SYNTAX ELEMENTS

n is a number from 1 to 6 that specifies the required scaling factor.

expression comprises operands, operators, conditional statements, and special functions.

NOTES

The A code replaces and enhances the functionality of the F code. You will find A codes much easier to work with than F codes.

A;expression	evaluates the expression.
An	converts to a scaled integer.
An;expression	converts to a scaled integer.
AE;expression	evaluates the expression.

A;EXPRESSION FORMAT

Performs the functions specified in expression on values stored without an embedded decimal point.

AN FORMAT: EMBEDDED DECIMALS

The An format converts a value stored with an embedded decimal point to a scaled integer. The stored value's explicit or implied decimal point is moved n digits to the right with zeros added if necessary. Only the integer portion is returned.

Field 2 of the data definition record must contain the FMC of the field that contains the data to be processed.

AN;EXPRESSION FORMAT

The An;expression format performs the functions specified in expression on values stored with an embedded decimal point. The resulting value is then converted to a scaled integer.

AE;EXPRESSION FORMAT

The AE format uses both the integer and fractional parts of stored numbers. Scaling of output must be done with format codes.

EXAMPLES OF NUMERIC RESULTS

Data Record		A Code		
Field 1	**Field 2**	**A;1 + 2**	**A3;1 + 2**	**AE;1 + 2**
4	012	16	16000	16
-77	-22	-99	-99000	-99
0.12	22.09	22	22210	22.21
-1.234	-12.34	-13	-13574	-13.574
-1.234	123.45	122	122216	122.216

INPUT CONVERSION

Input conversion is not allowed.

FORMAT CODES

You can format the result of any A code operation by following the expression with a value mark, and then the required format code, like this:

```
An;expression]format
```

Format codes can also be included within the expression. See Format Codes for more information.

SUMMARY OF OPERANDS

Operands that you can use in A code expressions include: FMCs (field numbers), field names, literals, operands that return system parameters, and special functions.

You can format any operand by following it with one or more format codes enclosed in parentheses, and separated by value marks, (ctrl]), like this:

```
operand(format-code{]format-code}...)
```

See Format Codes for more information.

FIELD NUMBER (FMC) OPERAND

The field number operand returns the content of a specified field in the data record:

```
field-number{R{R}}
```

The first R specifies that any non-existent multivalues should use the previous non-null multivalue. When the second R is specified, this means that any non-existent subvalues should use the previous non-null subvalue.

FIELD NAME OPERAND

The field name operand returns the content of a specified field in the data record:

```
N(field-name){R{R}}
```

LITERAL OPERAND

The literal operand supplies a literal text string or numeric value:

```
"literal"
```

SYSTEM PARAMETER OPERANDS

Several A code operands return the value of system parameters. They are:

D	Returns the system date in internal format.
LPV	Returns the previous value transformed by a format code.
NA	Returns the number of fields in the record.
NB	Returns the current break level counter. 1 is the lowest break level, 255 is the GRAND TOTAL line.
ND	Returns the number of records (detail lines) since the last control break.
NI	Returns the record counter.
NL	Returns the record length in bytes
NS	Returns the subvalue counter
NU	Returns the date of last update
NV	Returns the value counter
T	Returns the system time in internal format.
V	Returns the previous value transformed by a format code

SPECIAL OPERANDS

Some operands allow you to use special functions. They are:

I(expression)	Returns the integer part of expression.
R(exp1, exp2)	Returns the remainder of exp1 divided by exp2.
S(expression)	Returns the sum of all values generated by expression.
string[start-char-no, len]	Returns the substring starting at character start-char-no for length len.

SUMMARY OF OPERATORS

Operators used in A code expressions include arithmetic, relational and logical operators, the concatenation operator, and the IF statement.

ARITHMETIC OPERATORS

Arithmetic operators are:

+	Sum of operands
-	Difference of operands
*	product of operands
/	Quotient (an integer value) of operands

RELATIONAL OPERATORS

Relational operators specify relational operations so that any two expressions can treated as operands and evaluated as returning true (1) or false (0). Relational operators are:

= or EQ	Equal to
< or LT	Less than
> or GT	Greater than
<= or LE	Less than or equal to
>= or GE	greater than or equal to
# or NE	Not equal

LOGICAL OPERATORS

The logical operators test two expressions for true or false and return a value of true or false. Logical operators are:

AND	Returns true if both expressions are true.
OR	Returns true if any expressions is true.

CONCATENATION OPERATOR

The concatenation operator is a colon (:)

IF STATEMENT

The IF operator gives the A code its conditional capabilities. An IF statement looks like this:

```
IF expression THEN statement ELSE statement
```

FIELD NUMBER (FMC) OPERAND

Specifies a field which contains the value to be used.

COMMAND SYNTAX

```
field-number{R{R}}
```

SYNTAX ELEMENTS

field-number is the number of the field (FMC) which contains the required value.

R specifies that the value obtained from this field is to be applied repeatedly for each multivalue not present in a corresponding part of the calculation.

RR specifies that the value obtained from this field is to be applied repeatedly for each subvalue not present in a corresponding part of the calculation.

NOTES

The following field numbers have special meanings:

0	Record key
9998	Sequential record count
9999	Record size in bytes

EXAMPLE 1

```
A;2
```

Returns the value stored in field 2 of the record.

EXAMPLE 2

```
A;9999
```

Returns the size of the record in bytes.

EXAMPLE 3

```
A;2 + 3R
```

For each multivalue in field 2 the system also obtains the (first) value in field 3 and adds it. If field 2 contains 1]7 and field 3 contains 5 the result would be two values of 6 and 12 respectively. Where 3 does not have a corresponding multivalue, the last non-null multivalue in 3 will be used.

EXAMPLE 4

```
A;2 + 3RR
```

For each subvalue in field 2 the system also obtains the corresponding subvalue in field 3 and adds it. If field 2 contains 1\2\3]7 and field 3 contains 5\4 the result would be four values of 6, 6, 7, 12 and 4 respectively.

N (FIELD NAME) OPERAND

References another field defined by name in the same dictionary or found in one of the default dictionaries.

COMMAND SYNTAX

```
N(field-name){R{R}}
```

SYNTAX ELEMENTS

field-name is the name of another field defined in the same dictionary or found in the list of default dictionaries (see the JBCDEFDICTS environment variable).

R specifies that the value obtained from this field is to be applied repeatedly for each multivalue not present in a corresponding part of the calculation.

RR specifies that the value obtained from this field is to be applied repeatedly for each subvalue not present in a corresponding part of the calculation.

NOTES

If the data definition record of the specified field contains field 8 pre-process conversion codes, these are applied before the value(s) are returned.

Any pre-process conversion codes in the specified field-name, including any further N(field-name) constructs are processed as part of the conversion code.

N(field-name) constructs can be nested to 30 levels. The number of levels is restricted to prevent infinite processing loops. For example:

```
TEST1
008 A;N(TEST2)

TEST2
008 A;N(TEST1)
```

EXAMPLE 1

```
A;N(S.CODE)
```

Returns the value stored in the field defined by S.CODE.

EXAMPLE 2

```
A;N(A.VALUE) + N(B.VALUE)R
```

For each multivalue in the field defined by A.VALUE the system also obtains the corresponding value in B.VALUE and adds it. If A.VALUE returns 1]7 and B.VALUE returns 5, the result would be two values of 6 and 12 respectively.

EXAMPLE 3

```
A;N(A.VALUE) + N(B.VALUE)RR
```

For each subvalue in the field defined by A.VALUE the system also obtains the corresponding value in B.VALUE and adds it. If A.VALUE returns 1\2\3]7 and B.VALUE returns 5 the result would be four values of 6, 7, 8 and 12 respectively.

LITERAL OPERAND

Specifies a literal string or numeric constant enclosed in double quotes.

COMMAND SYNTAX

```
"literal"
```

SYNTAX ELEMENTS

literal is a text string or a numeric constant.

NOTES

A number not enclosed in double quotes is assumed to be a field number (FMC).

EXAMPLE 1

```
A;N(S.CODE) + "100"
```

Adds 100 to each value (subvalue) in the field defined by S.CODE.

EXAMPLE 2

```
A;N(S.CODE):"SUFFIX"
```

Concatenates the string "SUFFIX" to each value (subvalue) returned by S.CODE.

SYSTEM PARAMETER OPERANDS

Reference system parameters like date, time, the current break level, or the number of the current record.

COMMAND SYNTAX

system-operand

SYNTAX ELEMENTS

system-operand can be any of the following:

D	Returns the system date in internal format.
LPV	Returns the previous value transformed by a format code.
NA	Returns the number of fields in the record.
NB	Returns the current break level counter. 1 is the lowest break level, 255 is the GRAND TOTAL line.
ND	Returns the number of records (detail lines) since the last control break.
NI	Returns the record counter.
NL	Returns the record length in bytes
NS	Returns the subvalue counter
NU	Returns the date of last update
NV	Returns the value counter
T	Returns the system time in internal format.
V	Returns the previous value transformed by a format code

SPECIAL OPERANDS

INTEGER FUNCTION

The Integer Function I(expression) returns the integer portion of expression.

EXAMPLE

```
AE;I(N(COST) * N(QTY))
```

Returns the integer portion of the result of the calculation.

REMAINDER FUNCTION

The Remainder Function R(exp1, exp2) takes two expressions as operands and returns the remainder when the first expression is divided by the second.

EXAMPLE

```
A;R(N(HOURS) / "24")
```

Returns the remainder when HOURS is divided by 24.

SUMMATION FUNCTION

The Summation Function S(expression) evaluates an expression and then adds together all the values.

EXAMPLE

```
A;S(N(HOURS) * N(RATE)R)
```

Each value in the HOURS field is multiplied by the value of RATE. The multivalued list of results is then totalled.

SUBSTRING FUNCTION

The substring function [start-char-no, len] extracts the specified number of characters from a string, starting at a specified character.

SYNTAX ELEMENTS

start-char-no is an expression that evaluates to the position of the first character of the substring.

len is an expression that evaluates to the number of characters required in the substring. Use - len (minus prefix) to specify the end point of the substring. For example, [1, -2] will return all but the last character and [-3, 3] will return the last three characters.

EXAMPLE 1

```
A;N(S.CODE)["2", "3"]
```

Extracts a sub-string from the S.CODE field, starting at character position 2 and continuing for 3 characters.

EXAMPLE 2

```
A;N(S.CODE)[2, N(SUB.CODE.LEN)]
```

Extracts a sub-string from the S.CODE field, starting at the character position defined by field 2 and continuing for the number of characters defined by SUB.CODE.LEN.

FORMAT CODES

Specifies a format code to be applied to the result of the A code or an operand.

COMMAND SYNTAX

a-code{]format-code...}
a-operand(format-code{]format-code}...)

SYNTAX ELEMENTS

a-code is a complete A Code expression.

a-operand is one of the A Code operands.

format-code is one of the codes described later - G(roup), D(ate) or M(ask).

] represents a value mark that must be used to separate each format-code.

NOTES

You can format the result of the complete A code operation by following the expression with a value mark and then the required format
code(s). (This is actually a standard feature of the data definition records.)

Format codes can also be included within A code expressions. In this case, they must be enclosed in parentheses, and separated with a value mark if more than one format code is used.

All format codes will convert values from an internal format to an output format.

EXAMPLE 1

```
A;N(COST)(MD2]G0.1) * ...
```

Shows two format code applied within an expression. Obtains the COST value and applies an MD2 format code. Then applies a group extract to acquire the integer portion of the formatted value. The integer portion can then be used in the rest of the calculation. Could also have been achieved like this:

```
A;I(N(COST)(MD2)) * ...
```

EXAMPLE 2

```
A;N(COST) * N(QTY)]MD2
```

Shows the MD2 format code applied outside the A code expression. COST is multiplied by QTY and the result formatted by the MD2 format code.

OPERATORS

Operators used in A code expressions include arithmetic, relational and logical operators, the concatenation operator, and the IF statement.

ARITHMETIC OPERATORS

Arithmetic operators are:

+	Sum of operands
-	Difference of operands
*	product of operands
/	Quotient (an integer value) of operands

RELATIONAL OPERATORS

Relational operators specify relational operations so that any two expressions can treated as operands and evaluated as returning true (1)
or false (0). Relational operators are:

= or EQ	Equal to
< or LT	Less than
> or GT	Greater than
<= or LE	Less than or equal to
>= or GE	greater than or equal to
# or NE	Not equal

LOGICAL OPERATORS

The logical operators test two expressions for true (1) or false (0) and return a value of true or false. Logical operators are:

AND|Returns true if both expressions are true.
OR|Returns true if any expressions is true.

The words AND and OR must be followed by at least one space. The AND operator takes precedence over the OR unless you specify a different order by means of parentheses. OR is the default operation.

CONCATENATION OPERATOR

A colon (:) is used to concatenate the results of two expressions.

For example, the following expression concatenates the character "Z" with the result of adding together fields 2 and 3:

```
A;"Z":2 + 3
```

IF STATEMENT

The IF statement gives the A code conditional capabilities.

COMMAND SYNTAX

```
IF expression THEN statement ELSE statement
```

SYNTAX ELEMENTS

expression must evaluate to true or false. If true, executes the THEN statement. If false, executes the ELSE statement.

statement is a string or numeric value.

NOTES

Each IF statement must have a THEN clause and a corresponding ELSE clause.

IF statements can be nested but the result of the statement must evaluate to a single value.

The words IF, THEN and ELSE must be followed by at least one space.

EXAMPLE 1

```
A;IF N(QTY) < 100 THEN N(QTY) ELSE ERROR!
```

Tests the QTY value to see if it is less than 100. If it is, output the QTY field. Otherwise, output the text "ERROR!".

EXAMPLE 2

```
A;IF N(QTY) < 100 AND N(COST) < 1000 THEN N(QTY) ELSE ERROR!
```

Same as example 1 except that QTY will only be output if it is less than 100 and the cost value is less than 1000.

EXAMPLE 3

```
A;IF 1 THEN IF 2 THEN 3 ELSE 4 ELSE 5
```

If field 1 is zero or null, follow else and use field 5. Otherwise test field 2. If field 2 is zero or null, follow else and use field 4. Otherwise use field 3. Field 3 is only used if both fields 1 and 2 contain a value.

B CONVERSION

Provides interface for jBC subroutines or C functions to manipulate data during jQL processing. Synonymous with CALL code.

See Calling Subroutines from Dictionary Items for more details.

C CONVERSION

Concatenates fields, literals, and the results of a previous operation.

COMMAND SYNTAX

C{;}n{xn}...

SYNTAX ELEMENTS

; is optional. It has no function other than to provide compatibility.

n can be one of the following:

- a field number (FMC)
- a literal enclosed in single quotes, double quotes, or backslashes
- an asterisk (*) to specify the last generated value of a previous operation.

x is the character to be inserted between the concatenated elements. If you specify a semicolon (;), no separator will be used. Any non-numeric character except a system delimiters (value, subvalue, field, start buffer, and segment marks) is valid.

NOTES

See the descriptions of the function codes (A, F, FS and their variants) for other concatenation methods.

INPUT CONVERSION

Input conversion does not invert. The concatenation is applied to the input data.

EXAMPLE 1

C1;2

Concatenates the contents of field 1 with field 2, with no intervening separator character.

EXAMPLE 2

```
C1*2
```

Concatenates the contents of field 1 with an asterisk (*) and then the content of field 2.

EXAMPLE 3

```
C1*"ABC" 2/3
```

Concatenates the contents of field 1 with an asterisk (*), the string ABC, a space, field 2 a forward slash (/) and then field 3.

D CONVERSION

Converts dates between internal and external format.

COMMAND SYNTAX

D{p}{n}{s}

SYNTAX ELEMENTS

p is the special processing operator. Can be any one of the following:

D	Returns only the day of the month as a numeric value.
I	Returns only dates stored in the external format in internal format. You can use this in field 7 or 8.
J	Returns the Julian day (1 - 365, or 1 - 366 for a leap year).
M	Returns the number of the month (1 - 12).
MA	Returns the name of the month in uppercase letters.
Q	Returns the number of the quarter (1 - 4)
W	Returns the day of the week as a numeric value (Monday is 1).
WA	Returns the day of the week in uppercase letters (MONDAY - SUNDAY).
Y	Returns the year (up to four digits).

n is a number from 0 to 4 that specifies the how many digits to use for the year field. If omitted, the year will have four-digits. If n is 0, the year will be suppressed.

s is a non-numeric character to be used as a separator between month, date, and year. Must not be one of the special processing operators.

NOTES

Dates are stored internally as integers which represent the number of days (plus or minus) from the base date of December 31, 1967. For example:

Date	Stored Value
22 September 1967	-100
30 December 1967	-1

31 December 1967	0
01 January 1968	1
09 April 1968	100
26 September 1967	1000
14 January 1995	9876
29 February 2000	11748

If you do not specify a special processing operator (see later) or an output separator, the default output format is two-digit day, a space, a three-character month, a space, and a four-digit year. If you specify just an output separator, the date format defaults either to the US numeric format "mm/dd/yyyy" or to the international numeric format "dd/mm/yyyy" (where / is the separator). You can change the numeric format for the duration of a logon session with the DATE-FORMAT command.

PRE-PROCESSOR CONVERSION

Field 8 codes are valid but, generally, it is easier to specify the D code in field 7 for input conversion. Dates in output format are difficult to use in selection processing.

If you are going to use selection processing and you want to use a code which reduces the date to one of its parts, such as DD (day of month), the D code must be specified in field 8.

INPUT/OUTPUT CONVERSION

Field 7 input and output conversions are both valid.

Generally, for selection processing, you should specify D codes in field 7. An exception is when you use a formatting code, such as DM, that reduces the date to one of its parts.

If no year is specified in the sentence, the system assumes the current year on input conversion. If only the last two digits of the year are specified, the system assumes the following years:

00-29	2000-2029
30-99	1930-1999

EXAMPLES

D Code	Internal Value	Value Returned
D	9904	11 FEB 1995

D2/	9904	11/02/95
D-	9904	11-02-1995
D0	9904	11 FEB
DD	9904	11
DI	11 FEB 1995	9904
DJ	9904	41
DM	9904	2
DMA	9904	FEB
DQ	9904	1
DW	9904	6
DWA	9904	SATURDAY
DY	9904	1995
DY2	9904	95

D1 D2 CONVERSION

Associates controlling and dependent fields.

COMMAND SYNTAX

D1;fmcd{;fmcd}...
D2;fmcc

SYNTAX ELEMENTS

fmcd is the field number (FMC) of an associated dependent field.

fmcc is the field number (FMC) of the associated controlling field.

NOTES

You can logically group multivalued fields in a record by using a controlling multivalued field and associating other fields with it. You could, for example, group the component parts of an assembly on an invoice.

The D1 code in field 8 defines the controlling field and nominates the associated dependent fields. Each dependent field will have a D2 code in field 8.

IMPORTANT: The D1 and D2 codes must be in field 8 of the data definition record and must be the first code specified. The code can be followed by other codes (separated by a value mark), but it must be the first code.

The values in the dependent associative fields are output in the order in which they are specified in the field 8 of the controlling field. The order in which the dependent fields are specified in the output specification clause is irrelevant.

Dependent fields are marked on the output by an asterisk just below the column heading.

If the output for a controlling field is suppressed, the values in the dependent fields will also be suppressed.

EXAMPLE

LIST INVOICE "ABC123" PARTS QTY PRICE

The records in data file INVOICE have 3 associated, multivalued fields. The fields are named PARTS, QTY and PRICE, and numbered 7, 2 and 5 respectively.

PARTS is the controlling field because, for each multivalue in this field there will a corresponding value in the other fields, and also because PARTS should appear first on the report. The data definition record for PARTS will have D1;2;5 in field 8.

The data definition records for QTY and PRICE will both have D2;7 in field 8.

The report generated by the command will look something like this:

```
INVOICE PART QTY PRICE
*       *
ABC123  AAA    1 10.00
BBB           11  4.00
CCC            2  3.30
```

F CONVERSION

F codes provide many facilities for arithmetic, relational, logical, and concatenation operations. All operations are expressed in Reverse Polish notation and involve the use of a "stack" to manipulate the data.

SYNTAX SUMMARY

There are three forms of the F code:

F Uses only the integer parts of stored numbers unless a scaling factor is included. If the JBCEMULATE environment variable is set to "ROS", the operands for "-", "/" and concatenate are used in the reverse order.

FS Uses only the integer parts of stored numbers.

FE Uses both the integer and fraction parts of stored numbers.

COMMAND SYNTAX

```
F{n};elem{;elem}...
FS;elem{;elem}...
FE;elem{;elem}
```

SYNTAX ELEMENTS

n is a number from 1 to 9 used to convert a stored value to a scaled integer. The stored value"s explicit or implied decimal point is moved n digits to the right with zeros added if necessary. Only the integer portion of this operation is returned.

elem is any valid operator. See later.

NOTES

F codes use the Reverse Polish notation system. Reverse Polish is a postfix notation system where the operator follows the operands. The expression for adding two elements is "a b + ". (The usual algebraic system is an infix notation where the operator is placed between the operands, for example, "a + b").

The F code has operators to push operands on the stack. Other operators perform arithmetic, relational, and logical operations on stack elements. There are also concatenation and string operators.

Operands that are pushed on the stack may be constants, field values, system parameters (such as data and time), or counters (such as record counters).

THE STACK

F codes work with a pushdown stack.

NOTE:
All possible F correlative operators push values onto the stack, perform arithmetic and other operations on the stack entries, and pop values off the stack.

The term "push" is used to indicate the placing of an entry (a value) onto the top of the stack so that existing entries are pushed down one level. "Pop" means to remove an entry from the top of the stack so that existing entries pop up by one level. Arithmetic functions typically begin by pushing two or more entries onto the stack. Each operation then pops the top two entries, and pushes the result back onto the top of the stack. After any operation is complete, the result will always be contained in entry 1.

ORDER OF OPERATION

F code operations are typically expressed as "F;stack2;stack1;operation". Under most emulations, this will be evaluated as "stack2 operation stack1". If JBCEMULATE is set to "ROS", this example is evaluated as "stack1 operation stack2", effectively reversing the order of operations.

Note that the FE and FS codes are evaluated in the same way for all emulations.

INPUT CONVERSION

Input conversion is not allowed.

EXAMPLE 1

```
F;C3;C5;-
```

Push a value of 3 onto the stack. Push a value of 5 onto the stack.

Take entry 1 from entry 2 (3 - 5) and push the result (-2) back onto the stack as entry 1. ROS emulations will subtract 3 from 5 and return a result of 2.

EXAMPLE 2

FS;C3;C5;-

Push a value of 3 onto the stack. Push a value of 5 onto the stack. Take entry 2 from entry 1 (3 - 5) and push the result (-2) back onto the stack. This works in the same way for all emulations.

EXAMPLE 3

F;C2;C11;C3;-;/

Push a value of 2 onto the stack. Push a value of 11 onto the stack. Push a value of 3 onto the stack. Subtract entry 1 from entry 2 (11 - 3) and push the result (8) back onto the stack. Now divide entry 2 by entry 1 (2 divided by 8) and push the result (0) back onto the stack.

Under ROS emulation, this would evaluate as 3 - 11 = -8, followed by -8 / 2 = -4.

PUSH OPERATORS

A push operator always pushes a single entry onto the stack. Existing entries are pushed down one position. Push operators are: "literal"Literal. Any text string enclosed in double or single quotes.

field-number{R{R}}{(format-code)}

field-number is the value of the field from the current record.

R specifies that the last non-null value obtained from this field is to be applied repeatedly for each multivalue that does not exist in a corresponding part of the calculation.

RR specifies that the last non-null value obtained from this field is to be applied repeatedly for each subvalue that does not exist in a corresponding part of the calculation.

(format-code) is one or more format codes (separated by value marks) enclosed in parentheses. Applied to the value before it is pushed onto the stack.

Cn	Constant. Where n is a constant (text or number) of any length up to the next semicolon or system delimiter.
D	Current system date.
NA	Number of fields in the record.
NB	Current break level number: -1 = during SORT or SELECT processing 0 = detail line 1 = lowest break level 255 = GRAND-TOTAL line

ND	Number of records since the last BREAK on a BREAK data line. Equal to the record counter on a GRAND-TOTAL line. Used to compute averages.
NI	Record counter. The ordinal position of the current record in the report.
NL	Length of the record, in bytes. Includes all field marks but not the key.
NS	Subvalue counter. The ordinal position of the current subvalue within the field.
NV	Value Counter. The ordinal position of the current multivalue within the field.
P	or "Duplicate of entry 1 is pushed onto the stack.
T	System time in internal format.
V or LPV	Previous Value. The value from the previous format code is to be used.

ARITHMETIC OPERATORS

The arithmetic F code operators work on just the top stack entry or the top two stack entries. They are:

+	Add the top two stack entries together and push result into entry 1.	
-	Subtract stack entries and push result into entry 1:	
	F	subtract entry 1 from entry 2
	FS, FE	subtract entry 1 from entry 2
	F	subtract entry 2 from entry 1 (ROS emulation)
*{n}	Multiply the top two stack entries and push result into entry 1. If n is specified, the result is divided by 10 raised to the power of n.	
/	Divide stack entries and push quotient into entry 1:	
	F	divide entry 2 by entry 1
	FS, FE	divide entry 2 by entry 1
	F	divide entry 1 by entry 2 (ROS emulation)

R	Compute remainder from the top two stack entries and push result into entry 1:
	F — remainder of entry 2 / entry 1
	FS, FE — remainder of entry 2 / entry 1
	F — remainder of entry 1 / entry 2
I	Return the integer part of entry 1 to the top of the stack.
S	Replace the multivalued entry 1 with the sum of the multivalues and subvalues.

MISCELLANEOUS OPERATORS

Miscellaneous operators control formatting, exchanging stack entries, popping the top entry, concatenation, and string extraction. They are:

_	Exchange the top two entries.
^	Pop last entry from the stack and discard. All other entries are pushed up.
(format-code)	Perform the specified format code on last entry and replace last entry with the result.
:	Concatenate stack entries:
	F — Concatenates Entry 1 to the end of Entry 2
	FS, FE — Concatenates Entry 1 to the end of Entry 2
	F — Concatenates Entry 2 to the end of Entry 1(ROS emulation)
[ ]	Extract a substring from stack entry 3. The starting column is specified in stack entry 2 and the number of characters is specified in entry 1

RELATIONAL OPERATORS

Relational operators compare stack entries. The result is pushed onto stack entry 1 and is either 1 (true) or 0 (false). Relational operators are:

=	equal to	
<	less than:	
	F	entry 2 < entry 1
	FS,FE	entry2 < entry 1
	F	entry 1 < entry 2 (ROS emulation)
>	Greater than:	
	F	entry 2 > entry 1
	FS,FE	entry2 > entry 1
	F	entry 1 > entry 2 (ROS emulation)
[	Less than or equal to:	
	F	entry 2 [entry 1
	FS,FE	entry2 [entry 1
	F	entry 1 [entry 2 (ROS emulation)
]	Great than or equal to:	
	F	entry 2 [entry 1
	FS,FE	entry 2 [entry 1
	F	entry 1 [entry 2 (ROS emulation)
#	Not equal.	

LOGICAL OPERATORS

Logical operators include a logical AND test and a logical inclusive-OR test.

Logical operators are:

& AND stack entries 1 and 2. If both entries contain non zero, a 1 is pushed onto stack entry 1, otherwise, a 0 is pushed.

! OR stack entries 1 and 2. If either of the entries contains non zero, a 1 is pushed onto stack entry 1; otherwise, a 0 is pushed.

MULTIVALUES

A powerful feature of F and FS code operations is their ability to manipulate multivalues. Individual multivalues can be processed, one by one, or you can use the R (or RR) modifier after a field number, to repeat a value and thus combine it with each of a series of multivalues. Field operands may be valued and subvalued. When mathematical operations are performed on two multivalued lists (vectors), the result is also multivalued. The result has an many values as the longer of the two lists. Zeros are substituted for the null values in the shorter list if the R option is not specified.

REPEAT OPERATORS

To repeat a value for combination with multivalues, follow the field number with the R operator. To repeat a value for combination with multiple subvalues, follow the FMC with the RR operator.

FORMAT CODES

Format codes can be used in three ways. One transforms the final result of the F code, another transforms the content of a field before it is pushed on the stack, and the third transforms the top entry on the stack.

COMMAND SYNTAX

```
f-code{]format-code...}
field-number(format-code{]format-code}...)
(format-code{]format-code}...)
```

SYNTAX ELEMENTS

f-code is a complete F Code expression.

field-number is the field number in the record from which the data is to be retrieved.

format-code is any valid format codes.

] represents a value mark (ctrl]) that must be used to separate each format-code.

NOTES

To process a field before it is pushed on the stack, follow the FMC with the format codes enclosed in parentheses.

To process the top entry on the stack, specify the format codes within parentheses as an operation by itself.

To specify more than one format code in one operation, separate the codes with the value mark, (ctrl]).

All format codes will convert values from an internal format to an output format.

EXAMPLE

```
F;2(MD2]G0.1);100;-
```

Obtain the value of field 2. Apply an MD2 format code. Then apply a group extract to acquire the integer portion of the formatted value, and push the result onto the stack. Subtract 100 from the field 2 formatted, group extracted value. Return this value. Note that under ROS emulation, the value returned would be the result of subtracting the integer value from the group extract, from 100. In other words:

```
100 - OCONV(OCONV(Field2, "MD2"), "G0.1" ).
```

G CONVERSION

G codes extract one or more contiguous strings (separated by a specified character), from a field value.

COMMAND SYNTAX

G{m}xn

SYNTAX ELEMENTS

m is the number of strings to skip. If omitted or zero, extraction begins with the first character.

x is the separation character.

n is the number of strings to be extracted.

NOTES

The field value can consist of any number of strings, each separated by the specified character. The separator can be any non-numeric character, except a system delimiter.

If m is zero or null and the separator x is not found, the whole field will be returned. If m is not zero or null and the separator x is not found, null will be returned.

INPUT CONVERSION

Input conversion does not invert. It simply applies the group extraction to the input data.

EXAMPLE 1

```
G0.1
```

If the field contains "123.45", 123 will be returned. You could also use "G.1" to achieve the same effect.

EXAMPLE 2

```
G2/1
```

If the field contains "ABC/DEF/GHI", GHI will be returned.

EXAMPLE 3

```
G0,3
```

If the field contains "ABC,DEF,GHI,JKL", ABC,DEF,GHI will be returned. Note that the field separators are included in the returned string.

L CONVERSION

L codes return the length of a value, or the value if it is within specified criteria.

COMMAND SYNTAX

L{{min,}max}

SYNTAX ELEMENTS

min specifies that the process is to return an element if its length is greater than or equal to the number min.

max specifies that the process is to return an element if its length is less than or equal to the number max.

NOTES

The L code by itself returns the length of an element.

When used with max, or min and max, the L code returns the element if it is within the length specified by min and/or max.

EXAMPLE 1

L

Assuming a value of ABCDEF, returns the value 6.

EXAMPLE 2

L4

If JBCEMULATE is set to ROS, L4 is translated as return the value if its length is less than or equal to 4 - the equivalent of L0,4. Assuming a value of ABCDEF, L4 will return null - the value is longer than 4 characters.

If JBCEMULATE is not set to ROS, L4 is translated as return the value if its length is exactly equal to 4 - the equivalent of L4,4. Assuming a value of ABCDEF, L4 will return null - the value is longer than 4 characters.

EXAMPLE 3

```
L4,7
```

L4,7 is translated as return the value if its length is greater than or equal to 4 and less than or equal to 7. Assuming a value of ABCDEF, L4,7 will return ABCDEF.

MC CONVERSION

MC codes include facilities for:

- changing characters to upper or lower case
- extracting a class of characters
- replacing characters
- converting ASCII character codes to their hexadecimal or binary representations and vice versa
- converting a hexadecimal values to their decimal or binary equivalents and vice versa
- converting a decimal value to its equivalent in Roman numerals and vice versa

One source of confusion when using MC codes is that input conversion does not always invert the code. For most MC codes, if they are used in field 7 of the data definition record, the code will be applied in its original (un-inverted) form to the input data. For this reason you should always try to put MC codes into field 8 of the data definition record. The exceptions to this, where input conversion is effective, are clearly indicated in the following sections.

SUMMARY

MC codes are:

MCA	Extract only alphabetic characters
MC/A	Extract only non-alphabetic characters.
MCAB{S}	Convert ASCII character codes to binary representation. Use S to suppress spaces.
MCAX or MX	Convert ASCII character codes to hexadecimal representation.
MCB	Extract only alphabetic and numeric characters.
MC/B	Extract only special characters that are neither alphabetic nor numeric.
MCBA	Convert binary representation to ASCII characters.
MCBX	Convert a binary value to its hexadecimal equivalent.
MCC;x;y	Change all occurrences of character string x to character string y.
MCDR	Convert a decimal value to its equivalent Roman numerals. Input conversion is effective.

MCDX or MCD	Convert a decimal value to its hexadecimal equivalent. Input conversion is effective.
MCL	Convert all upper case letters (A-Z) to lower case.
MCN	Extract only numeric characters (0-9).
MC/N	Extract only non-numeric characters.
MCNP{c}	Convert paired hexadecimal digits preceded by a period or character c to ASCII code.
MCP{c}	Convert each non-printable character (X"00" - X"IF", X"80" - X"FE") to a period (.) or to character c.
MCPN{c}	Same as MCP but insert the two-character hexadecimal representation of the character immediately after the period or character c.
MCRD or MCR	Convert Roman numerals to the decimal equivalent. Input conversion is effective.
MCT	Convert all upper case letters (A-Z) in the text to lower case, starting with the second character in each word. Change the first character of each word to upper case if it is a letter.
MCU	Convert all lower case letters (a-z) to upper case.
MCXA or MY	Convert hexadecimal representation to ASCII characters.
MCXB{S}	Convert a hexadecimal value to its binary equivalent. Use S to suppress spaces between each block of 8 bytes.
MCXD or MCX	Convert a hexadecimal value to its decimal equivalent. Input conversion is effective.

CHANGING CASE

The MC codes that can be used to transform text from upper to lower case and vice versa are:

MCL	Convert all upper case letters (A-Z) to lower case
MCT	Convert all upper case letters (A-Z) in the text to lower case, starting with the second character in each word. Change the first character of each word to upper case.
MCU	Convert all lower case letters (a-z) to upper case.

INPUT CONVERSION

Input conversion does not invert. The conversion code will be applied to the input data.

EXAMPLE 1

MCL

Assuming a source value of AbCdEf, MCL will return abcdef.

EXAMPLE 2

MCT

Assuming a source value of AbC dEf "ghi, MCT will return Abc Def "ghi.

EXAMPLE 3

MCU

Assuming a source value of AbCdEf, MCU will return ABCDEF.

EXTRACTING CHARACTERS

The MC codes that can be used to extract characters from a string are:

MCA	Extract only alphabetic characters.
MC/A	Extract only non-alphabetic characters.
MCB	Extract only alphanumeric characters.
MC/B	Extract only special characters that are neither alphabetic nor numeric.
MCN	Extract only numeric characters (0-9).
MC/N	Extract only non-numeric characters.

INPUT CONVERSION

Input conversion does not invert. The original code will be applied to input data.

EXAMPLE 1

MCA

Assuming a source value of ABC*123!DEF, MCA will return ABCDEF.

EXAMPLE 2

MC/A

Assuming a source value of ABC*123!DEF, MC/A will return *123!.

EXAMPLE 3

MCB

Assuming a source value of ABC*123!DEF, MCB will return ABC123DEF.

EXAMPLE 4

MC/B

Assuming a source value of ABC*123!DEF, MC/B will return *!.

EXAMPLE 5

MCN

Assuming a source value of ABC*123!DEF, MCN will return 123.

EXAMPLE 6

MC/N

Assuming a source value of ABC*123!DEF, MC/N will return ABC*!DEF.

REPLACING CHARACTERS

Some MC codes replace one set of characters with other characters. These codes can:

- exchange one character string for another
- replace non-printable characters with a marker character

- replace non-printable characters with a marker character and the character"s hexadecimal representation
- replace the marker and hexadecimal representation with the ASCII code

MCC;x;y	Change all occurrences of character string x to character string y.
MCP{c}	Convert each non-printable character (X"00" - X"IF", X"80" - X"FE") to character c, or period (.) if c is not specified.
MCPN{c}	Same as MCP but insert the two-character hexadecimal representation of the character immediately after character c, or tilde (~) if c is not specified.
MCNP{c}	Convert paired hexadecimal digits preceded by a tilde or character c to ASCII code. The opposite of the MCPN code.

INPUT CONVERSION

Input conversion does not invert. The original code will be applied to input data.

EXAMPLE 1

MCC;X5X;YYY

Assuming a source value of ABC*X5X!DEF, MCC will return ABC*YYY!DEF.

EXAMPLE 2

MCPN.

Assuming a source value of ABC]]DEF where] represents a value mark, MCPN will return ABC.FC.FCDEF.

CONVERTING CHARACTERS

The MC codes that convert ASCII character codes to their binary or hexadecimal representations or vice versa are:

MCAB{S}	Convert ASCII character codes to binary representation (Use S to suppress spaces).

MCAX or MY	Convert ASCII character codes to hexadecimal representation
MCBA	Convert binary representation to ASCII characters.
MCXA or MX	Convert hexadecimal representation to ASCII characters.

NOTES

The MCAB and MCABS codes convert each ASCII character to its binary equivalent as an eight-digit number. If there is more than one character, MCAB puts a blank space between each pair of eight-digit numbers. MCABS suppresses the spaces.

When converting from binary to ASCII characters, MCBA uses blank spaces as dividers, if they are present. MCBA scans from the right-hand end of the data searching for elements of "eight-bit" binary strings. If it encounters a space and the element is not eight binary digits long, it prepends zeros to the front of the number until it contains eight digits. This continues until the leftmost digit is reached. Zeros are again prepended, if necessary. Each eight-digit element is then converted to its ASCII character equivalent.

INPUT CONVERSION

Input conversion does not invert. The original code will be applied to input data.

EXAMPLE 1

MCAX

Assuming a source value of ABC, MCAX will return 414243.

EXAMPLE 2

MCXA

Assuming a source value of 414243, MCXA will return ABC.

EXAMPLE 3

MCAB

Assuming a source value of AB, MCAB will return 01000001 01000010.

EXAMPLE 4

MCABS

Assuming a source value of AB, MCABS will return 0100000101000010.

EXAMPLE 5

MCBA

Assuming a source value of 01000001 1000010, MCBA will return AB. Note the missing binary digit at the start of the second element of the source value.

EXAMPLE 6

MCBA

Assuming a source value of 0100000101000010, MCBA will return AB.

CONVERTING NUMERIC VALUES

The MC codes that convert numeric values (as opposed to characters), to equivalent values in other number schemes are:

MCBX{S}	Convert a binary value to its hexadecimal equivalent. Use S to suppress spaces.
MCDR	Convert a decimal value to its equivalent Roman numerals. Input conversion is effective.
MCDX or MCD	Convert a decimal value to its hexadecimal equivalent. Input conversion is effective.
MCRD or MCR	Convert Roman numerals to the decimal equivalent. Input conversion is effective.
MCXB{S}	Convert a hexadecimal value to its binary equivalent. Use S to suppress spaces.
MCXD or MCX	Convert a hexadecimal value to its decimal equivalent. Input conversion is effective.

NOTES

These codes convert numeric values rather than individual characters. For example, a decimal value of 60 would be converted to X"3C" in hexadecimal, or LX in Roman numerals. The value 60 is converted, not the characters "6" and "0".

With the exception of MCBX{S} which will handle spaces, all these codes will stop if they encounter a character that is not a digit of the source number system. The value up to the invalid character will be converted.

With the exception of MCDR, if the conversion fails to find any valid digits, a zero will be returned. MCDR will return null.

If you submit an odd number of hexadecimal digits to the MCXB code, it will add a leading zero (to arrive at an even number of characters) before converting the value.

The MCXB and MCXBS codes convert each pair of hexadecimal digits to its binary equivalent as a eight-digit number. If there is more than one pair of hexadecimal digit, MCXB puts a blank space between each paid of eight-digit numbers. MCXBS suppresses the spaces.

When converting from binary to hexadecimal digits, MCBX uses blank spaces as dividers if they are present. MCBX effectively scans from the right-hand end of the data searching for elements of eight-bit binary digits. If it encounters a space and the element is not a multiple of eight binary digits, it prepends zeros to the front of the number until contains eight digits. This continues until the leftmost digit is reached. Zeros are again prepended, if necessary. Each eight-digit element is then converted to a hexadecimal character pair.

INPUT CONVERSION

Input conversion is effective for MCDR, MCDX, MCRD and MCXD. Input conversion is not inverted for the other codes. The original code will be applied to input data.

EXAMPLE 1

MCBX

Assuming a source value of 01000001 1000010, MCBX will return 4142. Would return the same value if there was no space between the binary source elements.

EXAMPLE 2

MCRD

Assuming a source value of MLXVI, MCRD will return 1066.

EXAMPLE 3

MCDX

Assuming a source value of 1066, MCDX will return 42A.

MD CONVERSION

The MD code transforms integers by scaling them and inserting symbols, such as a currency sign, thousands separators, and a decimal point. The ML and MR codes are similar to MD but have greater functionality.

COMMAND SYNTAX

MDn{m}{Z}{,}{$}{ix}{c}

SYNTAX ELEMENTS

n is a number from 0 to 9 that specifies how many digits are to be output after the decimal point. Trailing zeros are inserted as necessary. If n is omitted or 0, the decimal point is not output.

m is a number from 0 to 9 which represents the number of digits that the source value contains to the right of the implied decimal point. m is used as a scaling factor and 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 to be equal to n (the decimal precision). If m is greater than n, the source value is rounded up or down to n digits. The m option must be present if the ix option is used and both the Z and $ options are omitted. This to remove ambiguity with the ix option.

Z suppresses leading zeros. Note that fractional values which have no integer will have a zero before the decimal point. If the value is zero, a null will be output.

, specifies insertion of the thousands separator symbol every three digits to the left of the decimal point. The type of separator (comma or period) is specified through the SET-THOU command. (Use the SET-DEC command to specify the decimal separator.)

$ appends an appropriate currency symbol to the number. The currency symbol is specified through the SET-MONEY command.

ix aligns the currency symbol by creating a blank field of "i" number of columns. The value to be output overwrites the blanks. The "x" parameter specifies a filler character that can be any non-numeric character, including a space.

c appends a credit character or encloses the value in angle brackets (>). Can be any one of the following:

- Appends a minus sign to negative values. Positive or zero values are followed by a blank.
C Appends the characters CR to negative values. Positive or zero values are followed by two blanks.

INPUT CONVERSION

Input conversion works with a number that has only thousands separators and a decimal point.

EXAMPLES

MD Code	Source Value	Returned Value
MD2	1234567	12345.67
MD2,	1234567	12,345.67
MD2,$	1234567	$12,345.67
MD2,$12*	1234567	$**12,345.67
MD2,$12-	-1234567	$12,345.67-
MD42Z,$	001234567	$12,345.6700
MD42Z,$15 <	-001234567	<$ 12,345.6700>

MK CONVERSION

The MK code allows you to display large numbers in a minimum of columns by automatically descaling the numbers and appending a letter to represent the power of 10 used. The letters and their meanings are:

K 10 /3 (Kilo)
M 10 /6(Mega)
G 10 /9 (Giga)

COMMAND SYNTAX

MKn

SYNTAX ELEMENTS

n is a number that represents the field width. This will include the letter and a minus sign, if present.

NOTES

If a number will fit into the specified field width, it will not be changed.

If the number is too long but includes a decimal fraction, the MK code first attempts to round the fractional part so that the number will fit the field. If the number is still too long, the code rounds off the three low-order integer digits, replacing them with a K. If the number is still too long, the code rounds off the next three digits, replacing them with an M. If that is still too long, the code rounds off three more digits, replacing them with a G. If the number still does not fit the specified field, the code displays an asterisk.

If the field size is not specified or is zero, the code outputs null.

INPUT CONVERSION

Input conversion does not invert. It simply applies the metric processing to the input data.

EXAMPLES

Source Data	MK3	MK4	MK5	MK7
1234	1K	1234	1234	1234
123456789	*	123M	123M	123457K
1234567890 12345	*	*	*	123457G
999.9	1K	1000	999.9	999.9
-12.343567	-12	-12	-12.3	-12.344
-1234.5678	-1K	-1K	-1235	-1234.6
-0.1234	-.1	-.12	-.123	-0.1234

ML/MR CONVERSION

ML and MR codes format numbers and justify the result to the left or right respectively. The codes provide the following capabilities:

- Decimal precision and scaling
- Zero suppression
- Thousands separator
- Credit codes
- Currency symbol
- Inclusion of literal character strings.

COMMAND SYNTAX

```
ML{n{m}}{Z}{,}{c}{$}{fm}
MR{n{m}}{Z}{,}{c}{$}{fm}
```

SYNTAX ELEMENTS

ML provides left justification of the result.

MR provides right justification of the result.

n is a number from 0 to 9 that defines the decimal precision. It specifies the number of digits to be 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 is 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 to be equal to n (the decimal precision).

Z suppresses leading zeros. Note that fractional values which have no integer will have a zero before the decimal point. If the value is zero, a null will be output.

, is 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	Print the literal CR after negative values.
D	Print the literal DB after positive values.
E	Enclose negative values in angle brackets < >

M	Print a minus sign **after** negative values.
N	Suppresses embedded minus sign.

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.

$ specifies that a currency symbol is to be included. A floating currency symbol is placed in front of the value. The currency symbol is specified through the SET-MONEY command.

fm specifies a format mask. A format mask can include literal characters as well as format codes. The format codes are as follows:

Code	**Function**
#{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 may be enclosed in parentheses "( )".

NOTES

The justification specified by the ML or MR code is applied at a different stage from that specified in field 9 of the data definition record. The sequence of events starts with the data being formatted with the symbols, filler characters and justification (left or right) specified by the ML or MR code. The formatted data is then justified according to field 9 of the definition record and overlaid on the output field - which initially comprises the number of spaces specified in field 10 of the data definition record.

INPUT CONVERSION

Input conversion works with a number that has only thousands separators and a decimal point.

EXAMPLES

Conversion Code	Source Value	Dict Fields 9 and 10	Returned Value (columns) 12345678901234567890
MR2#10	1234	L 15	`     12.34`
MR2#10	1234	R 15	`          12.34`
ML2%10	1234	L 15	`12.3400000`
MR2%10	1234	R 15	`     0000012.34`
ML2*10	1234	L 15	`12.34*****`
MR2*10	1234	R 15	`     *****12.34`
MR2,$#15	12345678	L 20	`    #123,456.78`
MR2,&$#15	12345678	L 20	`#####123,456.78`
ML2,&*$#15	12345678	L 20	`#123,456.78*****`
MR2,& $#15	12345678	L 20	`# 123,456.78`
MR2,C&*$#15	-12345678	L 20	`#***123,456.78CR`
ML& ###-##-###	123456789	L 12	123-45-6789
ML& #3-#2-#4	123456789	L 12	123-45-6789
ML& Text #2-#3	12345		Text 12-345
ML& ((Text#2) #3)	12345		(Text12) 345

In the last example, the leading and trailing parenthesis are ignored.

MP CONVERSION

MP codes convert packed decimals to unpacked decimal representation for output or decimal values to packed decimals for input.

COMMAND SYNTAX

```
MP
```

NOTES

The MP code most often used as an output conversion. On input, the MP processor combines pairs of 8-bit ASCII digits into single 8-bit digits as follows:

- The high order four bits of each ASCII digit are stripped off.
- The low order four bits are moved into successive halves of the stored byte.
- A leading zero will be added (after the minus sign if present) if the result would otherwise yield an uneven number of halves.
- Leading plus signs (+) are ignored.
- Leading minus (-) signs are stored as a four-bit code (D) in the upper half of the first internal digit.

When displaying packed decimal data, you should always use an MP or MX code. Raw packed data is almost certain to contain control codes that will upset the operation of most terminals and printers.

INPUT CONVERSION

Input conversion is valid. Generally, for selection processing you should specify MP codes in field 7 of the data definition record.

EXAMPLES

```
OCONV -1234 "MP"
```

yields 0x D01234

```
ICONV 0x D01234 "MP"
```

yields -01234

MS CONVERSION

The MS code allows an alternate sort sequence to be defined for sort fields.

COMMAND SYNTAX

```
MS
```

NOTES

Use of the MS code is only relevant when used in a field 8 pre-process code and applied to a field that is specified in a sort clause. In all other cases it will be ignored.

The sort sequence to be used is defined in a special record named SEQ that you must create in the ERRMSG file. Field 1 of this record contains a sequence of ASCII characters that define the order for sorting.

EXAMPLE

```
SEQ (defined in ERRMSG file)
001 aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyY
    zZ9876543210 ,.?!"";:+-*/^=()[]{}<>@#$%&"~\|

INV.CODE (data definition record)
001 A
.
.
008 MS

SORT SALES BY INV.CODE INV.CODE ID-SUPP

SALES....

AbC789
ABC789
ABC788
dEF123
```

MT CONVERSION

The MT code is used to convert time notations such as 01:40:30 or 1:30 AM between internal and external format.

COMMAND SYNTAX

MT{H}{S}

SYNTAX ELEMENTS

H specifies 12-hour format. If omitted, 24-hour format will be used.

S specifies that seconds are to be included.

NOTES

Time is stored internally as the number of seconds since midnight. The stored value can be output in 12 hour or 24 hour (international) format.

12:00PM is defined as noon and 12:00AM is defined as midnight.

For output conversions, AM and PM designators are automatically displayed. For example: 09:40AM and 06:30PM.

INPUT CONVERSION

Input conversion is valid. Generally, for selection processing you should specify MT codes in field 7 of the data definition record.

AM or PM designators are taken into account. The result of the input conversion for certain values can be affected by the time_is_hours emulation setting.

EXAMPLES

INPUT CONVERSION

Code	**Input**	**Result**
MT	00:00	0
MTH	12:00AM	0
MT	01:00AM	3600
MT	01:00	3600

Code	**Input**	**Result**
MTH	01:00	3600
MTH	01:00AM	3600
MT	01:00PM	46800
MTH	01:00PM	46800
MTS	01:00:30	3630

OUTPUT CONVERSION

Code	**Source Value**	**Result**
MTS	0	00:00:00
MTHS	0	12:00:00AM
MT	3600	01:00
MTH	3600	01:00AM
MT	46800	13:00
MTS	46800	13:00:00
MTH	46800	01:00PM
MTHS	46800	01:00:00PM

P CONVERSION

The P code returns a value if it matches one of the specified patterns. Patterns can be combinations of numeric and alphabetic characters and literal strings.

COMMAND SYNTAX

P{#}(element){;(element)}...

SYNTAX ELEMENTS

returns null if the source matches the pattern, or the source if it does not.

element is one or more of the following:

nA	tests for n alphabetic characters
nC	tests for n alphabetic or numeric characters
nN	tests for n numeric characters
nP	tests for n printable characters
nX	tests for n characters
"literal"	tests for the presence of the literal

NOTES

If the value does not match any of the patterns, a null will be returned.

INPUT CONVERSION

Input conversion does not invert. It simply applies the pattern matching to the input data.

EXAMPLE 1

```
P(2A"*"3N"/"2A)
```

Will match and return AA*123/BB or xy*999/zz. Will fail to match AAA*123/BB or A1*123/BB, and will return null.

EXAMPLE 2

```
P(2A"*"3N"/"2A);(2N"-"2A)
```

Will match and return AA*123/BB, xy*999/zz, 99-AA or 10-xx. Will fail to match AA&123/BB, A1*123/BB, 9A-AA or 101-xx, and will return null.

R CONVERSION

The R code returns a value that falls within one or more specified ranges.

COMMAND SYNTAX

```
Rn,m{;n,m}...
```

SYNTAX ELEMENTS

n is the starting integer of the range. Can be positive or negative.

m is the ending integer of the range. Can be positive or negative, but must be equal to or greater than n.

NOTES

If the value does not fall within the range(s), a null will be returned.

INPUT CONVERSION

Input conversion does not invert. It simply applies the range check to the input data.

EXAMPLE 1

```
R1,10
```

Will return any value that is greater than or equal to 1 and less than or equal to 10.

EXAMPLE 2

```
R-10,10
```

Will return any value that is greater than or equal to -10 and less than or equal to 10.

EXAMPLE 3

```
R-100,-10
```

Will return any value that is greater than or equal to -100 and less than or equal to -10.

S CONVERSION

The S code substitutes one value for another.

COMMAND SYNTAX

S;Var1;Var2

SYNTAX ELEMENTS

Var1 specifies the value to be substituted if the referenced value is not null or zero. Can be a quoted string, an FMC (field number), or an asterisk. An asterisk indicates that the value of the referenced field should be used.

Var2 specifies the value to be substituted if the referenced value is null or zero. Can be a quoted string, an FMC (field number), or an asterisk.

EXAMPLE 1

```
S;*;"NULL VALUE!"
```

If the referenced field is null, this example will return the string "NULL VALUE!". Otherwise it will return the referenced value.

EXAMPLE 2

```
S;*;3
```

If the referenced field is null, this example will return the content of field 3 of the data record. Otherwise it will return the referenced value.

EXAMPLE 3

```
S;4;5
```

If the referenced field is null, this example will return the content of field 5 of the data record. Otherwise it will return the content of field 4.

T CONVERSION

The T code extracts a character substring from a field value.

COMMAND SYNTAX

T{m,}n

SYNTAX ELEMENTS

m specifies the starting column number.

n is the number of characters to be extracted.

NOTES

If m is specified, the content of field 9 of the data definition record has no effect - characters are counted and extracted from left to right, for n characters.

If m is not specified, the content of field 9 of the data definition record will control whether n characters are extracted from the left or the right-hand end of the value. If field 9 does not contain an R, the first n characters will be extracted from the value. If field 9 does contain an R (right justify), the last n characters will be extracted from the value.

INPUT CONVERSION

Input conversion does not invert. It simply applies the text extraction to the input data.

EXAMPLES

Code	Source Value	Field 9	Result
T3,4	ABCDEFG	L	CDEF
T3,4	ABCDEFG	R	CDEF
T2	ABCDEFG	L	AB
T3	ABCDEFG	R	EFG
T3	ABCDEFG	T	ABC

JQL

TFILE CONVERSION

Tfile codes provide a method for retrieving data fields from any other file to which the user has access.

COMMAND SYNTAX

T[*|DICT]file-specifier;c{n};{i-fmc};{o-fmc}

SYNTAX ELEMENTS

***** or **DICT** indicates that the dictionary of the specified file is to be used, rather than the data section.

file-specifier identifies the reference file by name in the format file-name{,data-section-name}.

c specifies a translation code. Can be any one of the following:

C	If reference record does not exist or the specified FMC is null, output the value unchanged.
I	Input verify. Functions as a C code for output and as a V code for input.
O	Output verify. Functions as a C code for input and as a V code for output.
V	Reference record must exist and the specified FMC must contain a translatable value. If the record does not exist or the FMC contains a null, an error message will be output.
X	If reference record does not exist or the specified FMC is null, return a null.

n specifies a value mark count to return one specific value from a multivalued field. If omitted, all values are returned.

i-fmc is the field number for input translation. If omitted or the value is null, no input translation takes place.

o-fmc is the field number for output translation. If the value is null, no output translation takes place.

NOTES

The current data value is used as the record key for searching the specified reference file.

A data field, or a single value from a data field, is returned from the record.

Tfile codes can be used in fields 7 or 8 of the data definition record. Use field 8 if translation of a multivalued field or comparisons and sorts are required.

If selection criteria might be applied, you can either use field 8, or you can use field 7 and set up special records in the reference file to perform any input translation you require.

The special records in the reference file have as record keys values that the field subject to translation may be compared with in a jQL sentence. Field i-fmc within these records contains the translate value that will be compared to values on file. Typically, values in a jQL sentence are output values, so that the special input translation records are effectively the inverse of the output translation records.

Tfile codes can be "embedded" in other conversion codes but you must still follow the syntactical conventions of the "host" code. For example, if you include a Tfile code in an F code conversion, the Tfile code must be enclosed in parentheses.

INPUT/OUTPUT CONVERSION

Output conversion is valid. The Tfile code has a parameter (o-fmc) that specifies the field in the translation record to use for output conversion.

Input conversion is valid. The Tfile code has a parameter (i-fmc) that specifies the field in the translation record to use for input conversion.

EXAMPLE 1

TSALES;X;;2

Using this Tfile code in field 8 of a data definition record, which also has a 0 in field 2, will cause the key of the current record to be used as the key when accessing the reference file SALES. If the record cannot be found, a null will be returned. If the record is found, the value of field 2 will be returned.

EXAMPLE 2

TSALES;C;;2

Using this Tfile code in field 8 of a data definition record, which also has a 6 in field 2, will cause the content of field 6 from the current record to be used as the key when accessing the reference file SALES. If the record cannot be found, or if found, field 2 is null, the content of field 6 of the current record will be returned. If the record is found, and field 2 contains a value, that value will be returned.

EXAMPLE 3

```
A;3(TSALES;X;;2)
```

Using this embedded Tfile code in field 8 of a data definition record will cause field 3 of the current record to be used as the key when accessing field 2 of the reference file SALES. If the record cannot be found a null will be returned. If the record is found, the value of field 2 will be returned.

U CONVERSION

The U code is used to execute a system subroutine to process values.

COMMAND SYNTAX

Uxxxx

SYNTAX ELEMENTS

xxxx is the hexadecimal identity of the routine.

NOTES

jBASE user exits are customized routines specially produced to perform extraordinary processing.

INPUT CONVERSION

Routine dependent.

BSELECT

Retrieves selected records and generates a list composed of data fields from those records as specified by any explicit or default output specifications. Each subvalue within a field becomes a separate entry within the list.

COMMAND SYNTAX

BSELECT file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier}{output-specification} {(options}

NOTES

When the command terminates, the total number of entries in the generated list is displayed and the list is made available as if it had been generated by a SELECT, GET-LIST or other list-providing command.

If you do not specify a sort-criteria clause, the record list will be unsorted.

If you do not specify an output-specification, the default data definitions "1", "2" etc. will be used.

EXAMPLE

```
BSELECT SALES WITH S.CODE = "ABC]" P.CODE
```

Creates a list containing all P.CODE values from all the records in the SALES file which have an S.CODE that starts with ABC.

COUNT

Reports the total number of records found in a file that match the selection criteria specified.

COMMAND SYNTAX

COUNT file-specifier {record-list} {selection-criteria} {USING file- specifier} {(options}

SYNTAX ELEMENTS

options can be one or more of the following:

Option	Description
B	Suppress initial line-feed.
C{n}	Display running c ounters of the number of records selected and records processed. Unless modified by n, the counter increments after every 500 records processed or the total number of records if less than 500. The n specifies a number other than 500 by which to increment. For example, (C25) increments the counter after every 25 records processed.
P	Send the report to the printer.

EXAMPLES

```
:COUNT SALES WITH VALUE > "1000"

91 Records counted
```

Count the number of records in the SALES file which have a value greater than 1000.

```
:COUNT SALES WITH VALUE > "1000" (C50

91 Records selected 240 Records processed

91 Records counted
```

Count the number of records in the SALES file which have a VALUE greater than 1000, and display a running total of selected and processed records after each group of 50 records are processed.

EDELETE

Deletes selected records from a file according to record list or selection criteria clauses.

COMMAND SYNTAX

EDELETE file-specifier [record-list | selection-criteria]

NOTES

EDELETE requires an implicit or explicit record list, or selection criteria. An implicit list can be provided by preceding the command with a SELECT, GET-LIST or other list-providing command.

EDELETE will immediately delete the specified records.

To clear all the records in a file, use the CLEAR-FILE command.

EXAMPLES

```
:EDELETE SALES "ABC" "DEF"

2 Records deleted
```

Delete the records ABC and DEF based on the explicit list of records.

```
:EDELETE SALES IF P.CODE = "GHI]"

n Records deleted
```

Delete all records in the SALES file in which the P.CODE field starts with GHI.

```
:SELECT SALES WITH S.CODE = "ABC"

n Records selected

>EDELETE SALES

n Records deleted
```

Selects all records in the SALES file in which the S.CODE field contains ABC, and deletes them.

ESEARCH

Generates an implicit list of records in a file if they contain (or do not contain) one or more occurrences of specified character strings.

COMMAND SYNTAX

ESEARCH file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier} {(options}

SYNTAX ELEMENTS

options can be one or more of the following:

Option	**Description**
A	ANDs prompted strings together. Records must contain all specified strings.
I	Displays the keys of selected records.
L	Saves the field numbers in which the specified strings were found. The resulting list contains the record keys followed by multivalued line numbers. Ignores the A and N options if either or both are specified.
N	Selects only those records that do not contain the specified string(s).
S	Suppresses the list but displays the record keys that would have been selected.

PROMPT

You will be prompted to supply one or more search strings:

STRING:

Enter the required character string and press RETURN. This prompt is repeated until only RETURN is pressed. There is no limit on the number of characters that be entered.

Do not enter double quotes unless they are part of the string to search.

NOTES

When the command terminates (unless the "S" option is used), the total number of entries in the generated list is displayed and the list is made available as if it had been generated by a SELECT, GET-LIST or other list-providing command.

If you do not specify a sort criteria clause, the record list will be unsorted.

EXAMPLE

```
:ESEARCH SALES (I
STRING: ABC
STRING: DEF
KEY1
KEY2

2 Records selected

>
```

Generates a list of all records in the SALES file which contain the strings ABC or DEF.

I-DUMP / S-DUMP

Displays the entire contents of items in a file, including the system delimiters.

COMMAND SYNTAX

I-DUMP file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier} {(options}

NOTES

The **S-DUMP** command can be used to produce sorted output.

System delimiters are denoted as follows:

Attribute mark	^
Value mark	]
Sub value mark	\

EXAMPLE 1

```
I-DUMP EMPLOYEE WITH EMP.JOB = "SALESREP"
```

The following output is generated:

```
8499^HARRIS^TAMMY^SALESREP^8698^5400^250000^30000^90030^^^^^]1158
8^
8654^MCBRIDE^KEVIN^SALESREP^8698^5620^215000^140000^90030^^^^^]11
639\^
8521^TAYLOR^MAVIS^SALESREP^8698^5402^219000^50000^90030^^^^^3]115
00^
```

EXAMPLE 2

```
S-DUMP EMPLOYEE BY EMP.HIREDATE WITH EMP.TITLE "SALESREP"
```

The following output is generated:

```
8499^HARRIS^TAMMY^SALESREP^8698^5400^250000^30000^90030^^^^^]1158
8^
8521^TAYLOR^MAVIS^SALESREP^8698^5402^219000^50000^90030^^^^^3]115
00^
8654^MCBRIDE^KEVIN^SALESREP^8698^5620^215000^140000^90030^^^^^]11
639\^
```

LIST

Generates a formatted report of records and fields from a specified file.

COMMAND SYNTAX

LIST file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier} {output-specification} {format-specification} {(options}

NOTES

If an output specification clause is not provided, the system will search for default data definition records (named 1, 2 and so on) in the file dictionary and then in the file specified in the JEDIFILENAME_MD environment variable. If default data definition records cannot be found, the only the record keys will be listed.

The records will not be sorted unless you specify a sort criteria clause.

EXAMPLE 1

```
LIST SALES
```

List all the records in the SALES file and use the default data definition records (if found) to format the output.

EXAMPLE 2

```
LIST SALES "ABC" "DEF" "GHI"
```

List the records from the SALES file with key values of ABC, DEF or GHI. Use the default data definition records (if found) to format the output.

EXAMPLE 3

```
GET-LIST SALES.Q4
>LIST SALES GT "DEF"
```

Get the previously saved list called SALES.Q4 and, using the list, report on the records in the SALES file which have a key greater than DEF. Use the default data definition records (if found) to format the output.

EXAMPLE 4

```
LIST SALES WITH S.CODE = "ABC]" OR "[DEF"
```

List the records in the SALES file in which the S.CODE field contains values which start with ABC or end with DEF. Use the default data definition records (if found) to format the output.

EXAMPLE 5

```
LIST SALES WITH NO S.CODE = "ABC]" OR "[DEF" (P
```

List the records in the SALES file in which the S.CODE field does not contain values which start with ABC or end with DEF. Output the report to the printer. Use the default data definition records (if found) to format the output.

EXAMPLE 6

```
LIST SALES BY S.CODE BREAK-ON S.CODE ""BL" P.CODE TOTAL VALUE
GRAND-TOTAL "Total" HEADING "Sales Code : "B" "DL" FOOTING " Page
"CPP" LPTR
```

Sort the SALES file by S.CODE. Output the S.CODE, P.CODE and VALUE fields.

Control break on a change in S.CODE and suppress the LINE FEED before the break. Reserve the break value for use in the heading ("B").

Maintain a running total of the VALUE field and output it at each control break.

Put the word "Total" on the grand-total line.

Set up a heading for each page which comprises the words "Sales Code : ", the sales code (from the break), a date and a LINE FEED. Set up a footing which contains the text "Page " and a page number, centered on the line.

Produce the report on the currently assigned printer.

LIST-LABEL

Outputs data in a format suitable for producing labels.

COMMAND SYNTAX

LIST-LABEL file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier}{output-specification} {format-specification} {(options}

PROMPTS

You will be prompted to supply formatting criteria as follows:

COL,ROW,SKIP,INDENT,SIZE,SPACE{,"C"}:

COL	Number of columns required to list the data across the page.
ROW	Number of lines for each record. Each element of the output specification will be output on a separate line, if more elements exist in the output specification than there are rows specified, the extra elements will be ignored. If more rows are specified than elements, the output specification for these rows will be blank.
SKIP	Number of blank lines between each record.
INDENT	Number of spaces for left margin.
SIZE	Number of spaces required for the data under each column.
SPACE	Number of horizontal spaces to skip between columns.
C	Optional. Suppresses null or missing data. If absent, null or missing values are output as blanks. If present the C must be upper case and not in quotes.

NOTES

The total number of columns specified must not exceed the page width, based on the calculation:

COLs * (SIZE + SPACE) + INDENT <= page width

ROW must be a minimum of one for each field, plus one for the record key (if not suppressed). If record keys are not suppressed, the first row of each label will contain the record key.

If INDENT is not zero, you will be prompted to supply a series of HEADERs that will appear in the left margin for each field. If a heading is not required for a particular line, press RETURN.

Multivalued fields appear as separate labels.

If COL-HDR-SUPP or HDR-SUPP, or the C or H options are specified, the page number, date, and time will not be output and the report will be generated without page breaks. The records will not be sorted unless you specify a sort criteria clause.

See also the SORT-LABEL command.

EXAMPLE

```
LIST-LABEL SALES NAME ADDRESS STREET TOWN POSTCODE ID-SUPP (C

COL,ROW,SKIP,INDENT,SIZE,SPACE(,C): 2,5,2,0,25,4,C

NAME1                            NAME2

ADDRESS1                         ADDRESS2

STREET1                          STREET2

TOWN1                            TOWN2

POSTCODE1                        POSTCODE2


NAME3                            NAME4

ADDRESS3                         ADDRESS4

STREET3                          STREET4

TOWN3                            TOWN4

POSTCODE3                        POSTCODE4
```

LISTDICT

Generates a report of all data definition records in the first MD/VOC file found, or the specified file.

COMMAND SYNTAX

LISTDICT {file-specifier}

SYNTAX ELEMENTS

file-specifier specifies a dictionary file other than a file named MD/VOC in the JEDIFILEPATH.

NOTES

If you do not specify a file-name, LISTDICT will work with the first file named MD which is found in your JEDIFILEPATH.

REFORMAT

REFORMAT is similar to the LIST command in that it generates a formatted list of fields, but its output is directed to another file or the magnetic tape rather than to the terminal or printer.

COMMAND SYNTAX

REFORMAT file-specifier {record-list} {selection-criteria} {USING file-specifier} {output-specification} {format-specification} {(options}

PROMPT

You will be prompted to supply the destination file:

FILE:

Enter a file name, or the word "TAPE" for output to a magnetic tape.

NOTES

Records that already exist in the destination file will be overwritten.

When you reformat one file into another, each record selected becomes a record in the new file. The first value specified in the output specification clause is used as the key for the new records. The remaining values in the output specification clause become fields in the new records.

When you reformat a file to tape, the values specified in the output specification clause are concatenated together to form one tape record for each record that is selected. The record output is either truncated or padded at the end with nulls (X"00") to obtain a record the same length as specified when the tape was assigned by the T-ATT command.

Unless you specify HDR-SUPP or COL-HDR-SUPP, or a C or H option., a tape label containing the file name, tape record length (in hexadecimal), the time, and date will be written to the tape first. If a HEADING clause is specified, this will form the data for the tape label.

Record keys are displayed as the records are written to tape unless the ID-SUPP modifier or the I option is specified.

Two EOF marks terminate the file on tape.

See also the SREFORMAT command.

EXAMPLE

```
:REFORMAT SALES C.CODE NAME ADDRESS
FILE: ADDRESS
```

Creates new records in the ADDRESS file, keyed on C.CODE from the SALES file. Each record contains two fields, one with the values from the NAME field and one with the values from the ADDRESS field.

SELECT

Generates an implicit list of record keys or specified fields based on the selection criteria specified.

COMMAND SYNTAX

SELECT file-specifier {record-list} {selection-criteria} {sort-criteria} {output-criteria} {USING file-specifier} {(options}

SYNTAX ELEMENTS

options are:

C{n} Display running counters of the number of records selected and records processed. Unless modified by n, the counter increments after every 500 records processed or the total number of records if less than 500.

n n specifies a number other than 500 by which to increment. For example, C25 increments the counter after every 25 records processed.

NOTES

The records will not be sorted unless you specify a sort criteria clause.

See also the SSELECT command.

If you specify an output-criteria clause, the generated list will comprise the data (field) values defined by the clause, rather than the selected record keys.

If you are in jSHELL when the command terminates, the total number of entries in the generated list is displayed and the list is made available to the next command, as indicated by the **>** prompt.

If you use the BY-EXP or BY-EXP-DSND connectives on a multivalued field, the list will have the format:

record-key]multivalue#

where multivalue# is the position of the multivalue within the field specified by BY-EXP or BY-EXP-DSND. multivalue# can be accessed by a READNEXT Var,n statement in a jBC program.

EXAMPLE 1

SELECT SALES WITH S.CODE = "ABC]"

```
23 Records selected
>LIST SALES WITH VALUE > "1000"
```

Select all the records in SALES file with an S.CODE value that starts with ABC. Then, using the list, report on the records in the SALES file which have a VALUE field greater than 1000.

EXAMPLE 2

```
SELECT SALES WITH S.CODE = "ABC]"
23 Records selected
>SAVE-LIST SALES.ABC
```

Select all the records in SALES file with an S.CODE value that starts with ABC. Then save the list as SALES.ABC.

SORT

Generates a sorted and formatted report of records and fields from a specified file.

COMMAND SYNTAX

SORT file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier} {output-specification} {format-specification} {(options}

NOTES

Unless a different sort order is specified in the sort criteria, the records will be presented in an ascending order based on the record key.

The data definition record (or the file definition record in the case of keys) determines whether a left or right sort will be applied to the data.

If the field is left-justified, the data will be compared on a character-by-character basis from left to right, using ASCII values. For example:

```
01
100
21
A
ABC
BA
```

If the field is right-justified and the data is numeric, a numeric comparison will be performed and the values ordered by magnitude.

If the field is right-justified and the data is alphanumeric, the data will be collated into an alphanumeric sequence. For example:

```
A
01
123
ABCD
```

If a descending sequence is required, the BY-DSND modifier should be used in the sort criteria. A descending sequence of record keys can be obtained by using the BY-DSND modifier with a data definition record that points to field 0 (the key). See "Sort Criteria Clause" earlier for a full explanation of the sorting process.

EXAMPLE 1

```
SORT SALES
```

Sort all the records in the SALES file into key order and use the default data definition records (if found) to format the output.

EXAMPLE 2

```
SORT SALES WITH S.CODE = "ABC" "DEF" "GHI"
```

Select the records in the SALES file in which the S.CODE field contains the values ABC, DEF or GHI. Sort the records into key order.

EXAMPLE 3

```
GET-LIST SALES.Q4
SORT SALES GT "DEF" BY S.CODE
```

Get the implicit list called SALES.Q4 and, using the list, report on the records in the SALES file which have a key greater than DEF. Sort the report by S.CODE.

EXAMPLE 4

```
SORT SALES WITH S.CODE = "ABC]" OR "[DEF" BY-DSND S.KEY LPTR
```

Select the records in the SALES file in which the S.CODE field contains values which start with ABC or end with DEF. Sort the report in descending order of S.KEY (a data definition record which points to field 0 - the key) and output the report to the printer

EXAMPLE 5

```
SORT SALES BY S.CODE BREAK-ON S.CODE ""BL" P.CODE TOTAL VALUE
GRAND-TOTAL "Total" HEADING "Sales Code : "B" "DL" FOOTING " Page
"CPP" LPTR
```

Sort the SALES file by S.CODE. Output the S.CODE, P.CODE and VALUE fields.

Control break on a change in S.CODE and suppress the LINE FEED before the break. Reserve the break value for use in the heading ("B"). Maintain a running total of the VALUE field and output it at each control break. Put the word "Total" on the grand-total line.

Set up a heading for each page which comprises the words "Sales Code : ", the sales code (from the break), a date and a LINE FEED. Set up a footing which contains the text "Page " and a page number, centered on the line.

Produce the report on the currently assigned printer.

SORT-LABEL

Outputs data in a format suitable for producing labels.

COMMAND SYNTAX

SORT-LABEL file-specifier {record-list} {selection-criteria} {sort-criteria} {USING file-specifier}{output-specification} {format-specification} {(options}

PROMPTS

You will be prompted to supply formatting criteria as follows:

COL,ROW,SKIP,INDENT,SIZE,SPACE(,C):

COL	Number of columns required to list the data across the page.
ROW	Number of lines for each record. Each element of the output specification will be output on a separate line, if more elements exist in the output specification than there are rows specified, the extra elements will be ignored. If more rows are specified than elements, the output specification for these rows will be blank.
SKIP	Number of blank lines between each record.
INDENT	Number of spaces for left margin.
SIZE	Number of spaces required for the data under each column.
SPACE	Number of horizontal spaces to skip between columns.
C	Optional. Suppresses null or missing data. If absent, null or missing values are output as blanks. If present the C must be upper case and not in quotes.

NOTES

The total number of columns specified must not exceed the page width, based on the calculation:

COLs * (SIZE + SPACE) + INDENT <= page width

ROW must be a minimum of one for each field, plus one for the record key (if not suppressed). If record keys are not suppressed, the first row of each label will contain the record key. The records will be sorted in key order unless you specify a sort criteria clause.

If INDENT is not zero, you will be prompted to supply a series of HEADERs that will appear in the left margin for each field. If a heading is not required for a particular line, press RETURN.

Multivalued fields appear on separate lines.

If COL-HDR-SUPP or HDR-SUPP, or the C or H options are specified, the page number, date, and time will not be output and the report will be generated without page breaks.

See also the LIST-LABEL command.

SREFORMAT

SREFORMAT is similar to the SORT command in that it generates a formatted list of fields, but its output is directed to another file or the magnetic tape rather than to the terminal or printer.

COMMAND SYNTAX

SREFORMAT file-specifier {record-list} {selection-criteria} {USING file-specifier} {output-specification} {format-specification} {(options)}

PROMPT

You will be prompted to supply the destination file:

FILE:

Enter a file name, or the word "TAPE" for output to a magnetic tape.

NOTES

Records that already exist in the destination file will be overwritten.

When you reformat one file into another, each record selected becomes a record in the new file. The first value specified in the output specification clause is used as the key for the new records. The remaining values in the output specification clause become fields in the new records.

When you reformat a file to tape, the values specified in the output specification clause are concatenated together to form one tape record for each record that is selected. The record output is either truncated or padded at the end with nulls (X"00") to obtain a record the same length as specified when the tape was assigned by the T-ATT command.

Unless you specify HDR-SUPP or COL-HDR-SUPP, or a C or H option., a tape label containing the file name, tape record length (in hexadecimal), the time, and date will be written to the tape first. If a HEADING clause is specified, this will form the data for the tape label.

Record keys are displayed as the records are written to tape unless the ID-SUPP modifier or the I option is specified.

Two EOF marks terminate the file on tape.

See the REFORMAT command for examples.

SSELECT

SSELECT generates an implicit list of record keys or specified fields, based on the selection criteria specified.

COMMAND SYNTAX

SSELECT file-specifier {record-list} {selection-criteria} {sort-criteria} {output-criteria} {USING file-specifier} {(options}

SYNTAX ELEMENTS

options are:

C{n} Display running counters of the number of records selected and records processed. Unless modified by n, the counter increments after every 500 records processed or the total number of records if less than 500.

n specifies a number other than 500 by which to increment. For example, C25 increments the counter after every 25 records processed.

NOTES

The records will be sorted in key order unless you specify a sort criteria clause.

See also the SELECT command.

If you specify an output-criteria clause, the generated list will comprise the data (field) values defined by the clause, rather than the selected record keys.

When the command terminates, the total number of entries in the generated list is displayed and the list is made available to the next command. This is indicated by the ">" prompt if you are in jSHELL.

If you use the BY-EXP or BY-EXP-DSND connectives on a multivalued field, the list will have the format:

record-key]multivalue#

where multivalue# is the position of the multivalue within the field specified by BY-EXP or BY-EXP-DSND. multivalue# can be accessed by a READNEXT Var,n statement in a jBC program.

EXAMPLE 1

```
SSELECT SALES WITH S.CODE = "ABC]"
```

```
23 Records selected
>LIST SALES WITH VALUE > "1000"
```

Select all the records in SALES file with an S.CODE value that starts with ABC. Sort the list into key order. Then, using the list, report on the records in the SALES file which have a VALUE field greater than 1000.

EXAMPLE 2

```
SSELECT SALES WITH S.CODE = "ABC]" BY P.CODE
23 Records selected
>SAVE-LIST SALES.ABC
```

Select all the records in SALES file with an S.CODE value that starts with ABC. Sort the list into P.CODE order and then save the list as SALES.ABC.

List Processing

jBASE includes an extensive range of list processing facilities. Most are provided in the form of specialised commands which can be used to build and maintain lists of record keys (selection lists). These commands are available directly from the command level, others are integral parts of the various jBASE languages.

A select list can either be active or saved. A list is said to be active if it will be used to modify the effect of the next command you issue. In jSHELL, an active list is indicated by a special system prompt (>) on the command line.

A saved list is a list that has been written away to a file for repeated use later. Typically, when a command is processed under an active list, the list will no longer be active when the command has been completed. You would then need to reprocess the potentially lengthy selection again to recreate the list. By saving the list while it is still active, you can tell the system to make it the active list as often as you need.

If you do not want to use a specific file to store the lists, the system will maintain a special area for you. In the same way, if you do not want to use a specifically named list, the system will maintain a default list name for you.

Lists are generally stored in a common file but are private to the generating account. This is achieved by automatically suffixing the list names with the name of the account from which they were created.

Saved lists are static data - they will not reflect changes to the data files after they are saved. Also, saved lists can take up significant space on the system. Regular housekeeping routines are required to ensure that you recover the space when the lists are no longer required.

Command Level List Processing

The list processing facilities available from the command level include the SEARCH facility which looks for specified text strings, and various Select List facilities such as FORM-LIST, GET-LIST and SORT-LIST.

Select lists are held in a special area of the database and are used to hold lists of record keys or other information for later use by a program or command.

COPY-LIST

Copies a saved list to another list or to another file.

COMMAND SYNTAX

COPY-LIST {from_listname} {from_accountname} {(options} TO: {to_spec}

SYNTAX ELEMENTS

from_listname specifies the source list. If you do not specify from_listname, the default list will be used.

from_accountname specifies the source account if different to the current account.

to_spec specifies the destination list. Can be:

{to_listname} {to_accountname}
or
({DICT }to_filename {to_record_key}

Use the first variant if you want to copy the list or change the account it's attached to. Use the second variant if you want to copy the list to a data file Each key becomes a separate field in the list record. Note use of the left parenthesis before **to_filename**.

If you do not specify **to_spec**, the list will be copied to the default list.

Option	**Description**
O	overwrite destination list or record if it already exists
D	delete source list after successfully copying it
L	synonymous with S option
N	suppresses auto paging. Only used with T
P	sends the list to the printer
S	suppresses line numbers. Only used with T or P.
T	sends the list to the screen.
X	outputs in hexadecimal notation; only used with T or P

NOTES

Also see the information on list storage.

EXAMPLE 1

```
:COPY-LIST A.SALES (O
TO: B.SALES
List "A.SALES" copied to "B.SALES"
```

Copies A.SALES (a previously saved list) to B.SALES, and overwrites if
necessary.

EXAMPLE 2

```
:COPY-LIST A.SALES
TO: A.SALES ACCOUNTS
List "A.SALES" copied to "A.SALES ACCOUNTS"
```

Copies A.SALES (a previously saved list belonging to the current account) to A.SALES, and marks it as belonging to the ACCOUNTS account.

EXAMPLE 3

```
:COPY-LIST A.SALES
TO: (SALES.LISTS APRIL.SALES
List "A.SALES" written to record "APRIL.SALES" in file
"SALES.LISTS"
```

Copies A.SALES (a previously saved list) to record APRIL.SALES, in file SALES.LISTS.

DELETE-LIST

Deletes a saved list.

COMMAND SYNTAX

DELETE-LIST {listname {account-name}}

SYNTAX ELEMENTS

listname specifies the name of the list to be deleted. If you do not specify listname, the default list will be deleted.

account-name is the name of the account from which the list was saved. If you do not specify account-name, the current account is assumed.

EXAMPLE

```
:DELETE-LIST A.SALES

List "A.SALES" deleted
```

DIFF-LIST

Creates a list from two stored lists. The new list contains all the items in the first list less any like items from the second list.

COMMAND SYNTAX

DIFF-LIST {DICT} File1 List1 TO {TargetList}
Less: {({DICT} File2} List2

SYNTAX ELEMENTS

List1 and **List2** are the names of stored lists.

File1 and **File2** represent the files in which the lists are stored. If **File2** is not specified then **List2** is assumed to be in **File1**.

TargetList is the name of the resultant list. If it is not specified then an active list is created otherwise it is written to file defined to hold stored lists.

EXAMPLE

File PROFILES holds the items LISTA and LISTB:

LISTA	**LISTB**
Jennifer	Michelle
Carrie	Sheila
Michelle	Mary
Renee	Carrie
Maryanne	
Cindy	

The command:

```
DIFF-LIST PROFILES LISTA TO LISTV
Less:LISTB
```

generates a stored list called LISTV that contains the elements Jennifer, Renee, Maryanne and Cindy.

The command:

```
DIFF-LIST PROFILES LISTB
Less:(PROFILES LISTA
```

generates an active list that contains the elements Sheila and Mary.

EDIT-LIST

Invokes an editor session to allow you to create, modify, merge or delete a select list.

COMMAND SYNTAX

EDIT-LIST {list-name {account-name}} {(options}

SYNTAX ELEMENTS

list-name specifies the name of the list to be edited. If the list does not already exist it will be created. If you do not specify a list name, the default list will be used.

account-name is the name of the account from which the list was saved. If you do not specify account-name, the current account name will be used.

options can be:
ED=editor editor is the name of the editor you want to use. Default is ED.

NOTES

The EDIT-LIST command is used when you want to edit the contents of a new or existing saved list. The command will convert the list into an editable record and pass it to the editor. When the edit session is finished, the record will be reconverted to a list and stored under its original name.

You can specify an editor other than ED by naming the editor in the ED option.

Also see the information on list storage.

EXAMPLE

```
EDIT-LIST A.SALES ED=NOTEPAD
```

Edits the list called A.SALES. The editor used is Windows Notepad.

FORM-LIST

Creates a select list from a record in a file.

COMMAND SYNTAX

FORM-LIST filename record-key {(n}

SYNTAX ELEMENTS

filename is the name of the file which contains record-key.

record-key is the name of the record to be converted into a select list.

n is the field number in the record from which the list is to start. If **n** is omitted the list will start from field 1.

NOTES

If you issue the FORM-LIST command without specifying a record key, you will be prompted to supply one.

The command will open file filename, read the record record-key and create a select list using each field in the record as a separate element for the list. The list formed becomes the active select list and will be inherited by the next jBASE command or program.

EXAMPLE

```
:FORM-LIST SALES.LISTS APRIL.SALES
200 records SELECTED
>LIST CUSTOMERS
```

A record named APRIL.SALES in file SALES.LISTS contains a list of customer keys. Each key occupies a separate field. The FORM-LIST command generates an active list of customers. You can then issue a subsequent command such as LIST CUSTOMERS which will use the active list.

GET-LIST

Retrieves a previously stored list from the common area and makes it the current active (default) list.

COMMAND SYNTAX

GET-LIST {list-name {account-name}}

SYNTAX ELEMENTS

list-name specifies the name of the list to be retrieved. If you do not specify a list name, the default list will be used.

account-name is the name of the account from which the list was saved. If you do not specify account-name, the system will use the current account name.

NOTES

The GET-LIST command is used to retrieve a list previously stored using the SAVE-LIST command. If the list is retrieved successfully, it will become the current active (default) list and will be inherited by the next jBASE command or program.

Also see the information on list storage.

EXAMPLE

```
:GET-LIST A.SALES
200 Records selected
>LIST SALES
```

Retrieves a list named A.SALES and generates an active list. You can then issue a subsequent command such as LIST SALES which will use the active list.

LIST-LISTS

Displays a list of all select lists currently available in the common area.

COMMAND SYNTAX

```
LIST-LISTS
```

QSELECT

Generates a select list from the fields of specified items and makes it the current active (default) list.

COMMAND SYNTAX

QSELECT {{file-name} record-list} {(n}

SYNTAX ELEMENTS

file-name specifies the source file name.

record-list is a list of record keys, or an asterisk (*) to signify all records.

n is the field number in each record, from which the list is to be created. If n is not specified, all the fields in the source records will be used to generate the list elements.

NOTES

If you issue the QSELECT command without specifying an record list, you will be prompted to supply one.

QSELECT is similar to the FORM-LIST command but has the additional capability of creating a list from more than one source record. Also, FORM-LIST creates a list from all the attributes of a single record, QSELECT can create a list from multiple records, and you can specify that each record is only to contribute one field.

EXAMPLE 1

```
:QSELECT SALES * (3
4000 records selected
>SAVE-LIST CUSTOMERS
```

Creates a list containing the values from field 3 of all the records in the SALES file. The list is then saved as CUSTOMERS.

EXAMPLE 2

```
:QSELECT SALES "ABC" "DEF" "GHI" (3
3 records selected
>SAVE-LIST CUSTOMERS
```

Creates a list containing the values from field 3 of records ABC, DEF and GHI in the SALES file. The list s then saved as

CUSTOMERS.

SAVE-LIST

Saves the currently active select list to a named list record.

COMMAND SYNTAX

SAVE-LIST {list-name}

SYNTAX ELEMENTS

list-name specifies the name under which the list is to be stored. If you do not specify a list name, the default list will be used

NOTES

The SAVE-LIST command is used to store the currently active (default) select list under a permanent list name. If the specified list name already exists, it will be overwritten.

Also see the information on list storage.

EXAMPLE

```
:SSELECT CUSTOMERS WITH POST.CODE = "MK]"
40 records selected
>SAVE-LIST CUSTOMERS.MK
:
```

Sorts and selects records from the CUSTOMER file which have a POST.CODE value that starts with MK. Saves the list as CUSTOMERS.MK.

SEARCH

The SEARCH command is used to create a list of all records in a file which contain one or more text sequences.

COMMAND SYNTAX

SEARCH filename {(N)}
Fist String to search for : string
Second String to search for : string

SYNTAX ELEMENTS

filename is the name of the file to be searched.

string is the text string to search for. You will be prompted for any number of strings to search for. Press the <Enter> key only to finish entering search strings.

If the (N) option is used then only items **not** containing the specified search strings are selected.

NOTES

The SEARCH command will scan the file specified by filename looking for records which contain the specified text string(s). If it finds any records that match the criteria, it will create a select list containing the record keys. This list then becomes the default select list and will be inherited by the next jBASE command or program.

SORT-LIST

Sorts a saved select list.

COMMAND SYNTAX

SORT-LIST {list-name{account-name}} {(options}

SYNTAX ELEMENTS

list-name specifies the name under which the list is to be stored.

account-name is the name of the account from which the list was saved. If you do not specify account-name, the system will use the current account name.

options can be
U - remove duplicate entries.

NOTES

The SORT-LIST command will sort a list previously saved with the SAVE-LIST command. SORT-LIST is useful for sorting lists which have modified by EDIT-LIST, or for avoiding an extra SSELECT when working with a large file.

EXAMPLE

```
jsh ~ -->SELECT CUSTOMERS LAST.NAME
4380 records selected
>SAVE-LIST CUSTOMERNAMES
jsh ~ -->SORT-LIST CUSTOMERNAMES (U

List 'CUSTOMERNAMES' sorted with 3872 records
```

Generates a sorted list of unique customer names.

XSELECT

Generates a select list (or display) of all keys in a file which do not match given selection criteria.

COMMAND SYNTAX

XSELECT file-name

SYNTAX ELEMENTS

file-name specifies the name of the file to be searched.

NOTES

If you issue the XSELECT command while a select list is active, the process will return with a list of all the records in the file which are not in the active list.

If you issue the XSELECT command without an active select list, you will be prompted initially for the name of a saved list to use. If you supply a list name the process will return with a list of all the records in the file which are not in the named list. If you do not supply a list name but press <Enter> to this prompt, you will be asked to supply selection criteria (in normal jQL format) which will identify all the records in the file that you do not want in the returned list.

In all cases, providing the returned list contains one or more record keys, you will be asked for a new name to save the list to. If you do not supply a list name but only press <Enter> to this prompt, the list of record keys will be displayed on the terminal screen.

EXAMPLE 1

```
:GET-LIST CUSTOMERS.MK
>XSELECT CUSTOMERS
Input save list name: CUSTOMERS.NOT.MK
```

GET-LIST makes the CUSTOMERS.MK list active. XSELECT then creates a list of all the records in the CUSTOMERS file which are not in the active list, and save it as CUSTOMERS.NOT.MK.

EXAMPLE 2

```
:XSELECT CUSTOMERS
Input list name: CUSTOMERS.MK
Input save list name: CUSTOMERS.NOT.MK
```

XSELECT first prompts for the name of a list to use. It then creates a list of all the records in the CUSTOMERS file which are not in the specified list, and save the new list as CUSTOMERS.NOT.MK.

EXAMPLE 3

```
:XSELECT CUSTOMERS
Input list name: <Enter>
Input selection criteria: WITH POST.CODE = "MK]"
Input save list name: CUSTOMERS.NOT.MK
```

XSELECT first prompts for the name of a list to use. Because a list name is not supplied, it prompts for the anti-selection criteria to use. A list is then created which contains all the keys of the CUSTOMERS file where the records do not match the selection criteria. The new list is saved as CUSTOMERS.NOT.MK.

jBC List Processing

The two main jBC list processing statements are READLIST and WRITELIST. However, if there is a currently active (default) list outstanding when you start a jBC program, you can also use the READNEXT command to extract elements from the external list.

jCL List Processing

The two list processing commands which are used with jCL programs are PQ-SELECT and PQ-RESELECT.

jQL List Processing

The four list processing commands which follow jQL command syntax are BSELECT, ESEARCH, SELECT and SSELECT.

List Storage

The jBASEWORK file is a temporary file which contains information specific to jBASE processes and ports while executing applications. This file should not be accessed while any jBASE processes are active.

The jBASEWORK file also contains default or numbered select lists. These select lists are temporary and so do not need to be deleted. Named select list may also be stored in the jBASEWORK file dependent upon configuration. See later.

The jBASEWORK can be safely deleted when no users are executing jBASE programs.

The jBASEWORK files can be configured to another directory by using the JBCBASETMP environment variable.

Named Select Lists

Depending upon the JBCLISTFILE environment variable and existence of a POINTER-FILE select lists are stored in one of three possible places.

First, if the environment file JBCLISTFILE is configured and is valid then the save list is stored with an identifier of the list name. If JBCLISTID is set then the identifier is **SEL*AccountName*ListName**.

If the JBCLISTFILE is not valid and a POINTER-FILE exists then the list is stored as named in the POINTER-FILE.

If JBCLISTFILE is not set and the POINTER-FILE does not exist, then the list is saved in the jBASEWORK file as **SEL*AccountName*ListName**. To store lists in jBASEWORK with an id of just the list name, set a Q-pointer or F-pointer called POINTER-FILE to jBASEWORK.

jBASE Editors

jBASE is supplied with its own fully featured screen editor that can be used for creating, modifying, or deleting records. The jED editor has been designed for ease of use, easy personal configuration and is especially suited to the editing of jBC programs.

jBASE also provides a limited version of an ED editor to enable the easy transition of users who are only familiar with the old style line editors.

JED

jED is a robust, terminal independent screen editing tool.

JED Command syntax

JED Filename Item (Options
JED Item (Options

Option	Description
Bnn{,mm}	Indent "nn" spaces, with "mm" as multiple for initial indent.
L	Skip setting default item lock on record.
R	Allow read only

Execution Commands

Command	Description
HOME/Ctrl A	Move to start of current line
END/Ctrl E	Move to end of current line
Ctrl W	Delete word
Ctrl K	Clear to end of line or join.
Ctrl D	Delete current line
Ctrl G	Mark block. 1st Start Blk, 2nd End Blk, 3rd Remove Mark
Ctrl L	Insert line below current line
Ctrl N	Locate next occurrence
Ctrl O	Toggle overwrite and insert. default insert
Ctrl R	Redisplay screen
Ctrl T	Copy the character from the corresponding cursor position on the line above
Ctrl V	Indent for BASIC
Ctrl]	Insert]
Ctrl \	Insert \

Command Line

From Edit mode, press <Esc> to invoke the command line.

Command	**Description**
CBn	Copy Marked block before current line, n times
Can	Copy Marked block after current line. n times
/string	Locate the next occurrence of "string"
MB	Move Marked block before current line
MA	Move Marked block after current line
BI	Format BASIC code
BION	Turn on Format indentation
! Cmd	Execute command
!!	Re-execute last ! Cmd
HX or HEX	Toggle the display of the record in Hexadecimal

jED

The jED editor is a full screen, context sensitive, screen editor. It has been specifically designed so that users will find it easy to learn and use and is the preferred editing tool for the jBASE operating environment.

The features provided in jED will be familiar to users of many other editors. It incorporates many powerful facilities for manipulating text and data, and contains all the features that programmers have come to expect of a good editor.

jED has full screen access to jBASE file records and UNIX files - a feature which is not provided by other editors.

The command keystrokes are fully configurable by each user, as is the keyboard. jED can therefore be customized to mimic many operations of other editors and provide a familiar environment for new users. Keyboard and command independence make jED the most versatile and powerful editing tool available for all jBASE editing requirements

EDITOR SCREEN

A typical jED editor session might look like this:

```
*File PROGS, Record cust_rep.b Insert 10:45:17

Command->

001

002

003

-------------------------------- End Of Record ------------------
--------------
```

The screen is divided into three sections; the editor status line at the top, followed by the command line and then the data editing area which fills the rest of the screen.

INVOKING jED

Call the jED editor from the UNIX or Windows command line.

COMMAND SYNTAX

jed pathname {pathname..}

jed {DICT} filename{,filesection} {record-list} {(options)}}

If you simply issue the command jed, the editor will open the last file that was used, and the cursor will be positioned wherever it was when the last edit session was closed. In other words, you can carry on from exactly where you left off. The command, jed pathname, will either open an existing file or create a new one if the file referenced by pathname does not exist. The will contents of the file will be displayed in the edit window. If you specify a list files, the editor will present the next file as you finish with each one.

When the editor is supplied with the name of a file resident in a database (such as a j-file), it scans the rest of the command line looking for a list of records keys. If no record keys were specified, the jED editor will prompt for a list. Otherwise the list of record keys will be edited one after the other.

Note that because the editor uses the jEDI interface to access the records, it can be used to edit records in any file system that jEDI recognizes.

COMMAND ELEMENTS FOR DATABASE RESIDENT FILES

DICT This modifier is only required if you wish to edit records in the DICTionary of a j-file.

filename This is the name of the "file" containing the records.

filesection This is the file section name, as used in a j-file.

record-list It is possible to furnish a list of records to be successively edited. This can be a list of records separated by a space, or "*" to indicate all records in the file. Note that the \ is the shell escape character to stop the * being treated as a wild card that would otherwise be expanded. Additionally, the record-list can be fed to this command by preceding the jed command with a jBASE list generating command such as SELECT or SSELECT. In this case, the record-list is ignored.

COMMAND LINE OPTIONS

Options available when executing the jed command are as follows:

Option	**Explanation**
Bnn{,mm}	performs automatic indentation on the record to be edited. This will be of use when creating jBC programs.
nn	parameter specifies the number of spaces to indent by for each indentation level (default is 4 spaces).
mm	this optional parameter is the number of times the nn indent value should be applied at the start of each line. If mm is 2 and nn is 3, each line will be indented initially by 6 spaces and each subsequent indent level will be 3 further spaces.
E	uses the default keyboard command set-up at installation, rather than that which may have been set up exclusively for the port.
L	does not lock the file or record being edited. This allows simultaneous edit access from elsewhere.
R	allows READ ONLY access to the record or file.
S	space characters not trimmed from end of line
Tnn	sets tab stops every nn spaces for use within the editor.

EXAMPLES

jed test.b
Opens the test.b file for editing, initially in insert mode with automatic indentation turned on. If the file does not exist, it is created and the text New Record is shown at the top of the screen.

jed test.b (B5,2
The jBC program test.b is edited with automatic indentation set. The initial indent is set at 10 spaces for all lines, and each additional indentation level is set at 5 spaces.

jed invoices.b subs.c
The jBC program invoices.b will be edited, followed by the "C" program subs.c.

jed BP menu1.b menu1.1.b
The jBASE file records menu1.b and menu1.1.b are successively edited. Record locks are taken on the records as they are edited to prevent multiple edits on the same record.

JED ORDERS 0012753 0032779 (R

The records 0012753 and 0032779 from the file ORDERS will be successively edited in read-only mode.

:SSELECT ORDERS WITH CUST.NAME = "UPA"
>JED ORDERS
The orders of the customer UPA will be successively edited in sorted order. Record locks will be automatically set during the editing period to prevent simultaneous updates by other users.

jed -F BP *
All the records in the jBASE file BP are set up to be edited one after the other. Note the use of the shell escape character (\) before the *.

jed -F BP STXFER.b \(T10
The record STXFER.b in file BP is opened for editing. A tab stop is set at column 10 for use in this session.

USING THE JED EDITOR

The jED editor is used in two different modes:

Command mode for entering editor commands, and
Edit mode for entering or modifying data.

The current mode is displayed at the top of the screen.

COMMAND MODE

When the editor is invoked, the record or text file is displayed, and the user is placed in input mode with the cursor at the input position.

To change to command mode simply press the <Esc> key on the keyboard. The cursor now moves to the top portion of the screen and the editor awaits input of a command. Once a valid command has been executed, control passes back to the Edit mode if appropriate.

EDIT MODE

Edit mode is used when entering or modify data. This is the default mode for an editor session.

Keyboard control sequences are available to perform a variety of functions such as cursor positioning, scrolling and marking text for a subsequent action.

Some command line operations are also available from keyboard control sequences.

KEYBOARD PERSONALIZATION

The jED editor allows a considerable number of functions and commands to be performed whilst in edit mode, mostly by combining the <Ctrl> key and one other key.

Most keys have a default value (which can be reset using the E option when invoking jED). These can be reconfigured for each command. The keystroke sequence can be chosen to suit the keyboard, the installation environment or personal preference.

The keystroke environment is usually be set up by modifying the UNIX terminfo file parameters. The default editor commands can also be overriden by configuring the .jedrc file.

jED DEFAULT KEY COMMANDS

The default keystroke sequences available from jED are shown below. If the system administrator has reconfigured these for a particular port, they can be re-assigned by using the E option when starting a jED session. The execution of a command is relative to the current cursor position.

Key	**Function**
<F1>	scrolls the screen up one line.
<F2>	scrolls the screen down one line.
<F3>	scrolls the screen up half a page.
<F4>	scrolls the screen down half a page.
<F5>	scrolls the screen up one page.
<F6>	scrolls the screen down one page.
<F7>	displays the first page of the record or file.
<F8>	displays the last page of the record or file.
<F9>	pressing <F9> when the cursor is positioned on a line of source code that begins a structured statement (IF, BEGIN CASE etc.), will cause the editor to locate the closing statement for the structure. If the cursor line is an IF statement then the editor will attempt to find the END statement that closes this structure. If there is no matching END statement then the editor will display a message to this effect.

Key	**Function**
<F10>	the <F10> key is complement of the <F9> key. Therefore if the cursor is positioned on an END statement, then the editor will attempt to find the start of the structure that it is currently terminating. If the END has been orphaned (it matches no structure), then the editor will display a message to this effect.
<Ctrl A>/<Home>	moves cursor to start of the current line.
<Ctrl E>/<End>	moves the cursor to the end of the current line.
Left Arrow	moves the cursor one character position to the left.
Right arrow	moves the cursor one character position to the right.
Up arrow	moves the cursor to the previous line.
Down arrow	moves the cursor to the following line.
<Tab>	moves the cursor to the start of the next tab position on the line.
<Shift Tab>	moves the cursor to the previous tab position on the line.
<Esc>	moves the cursor to the COMMAND LINE.
<Ctrl W>	deletes from the cursor to the end of the word, including the following whitespace characters.
<Ctrl K>	clears text to the end of the line. If the cursor is situated at the end of the text line, then this command will join the following line with the current line.
<Back Space>	performs a destructive backspace.
<Delete>	deletes the character under the current cursor position.
<Ctrl D>	deletes the current line. By default, this key must be pressed twice to delete the line. This is to avoid accidental deletion by users familiar with vi. To override, place "set delete-line = ^D" in the .jedrc file.
<Ctrl G>	sets the start or end position for marking a block of text. The first <Ctrl G> will mark the start of a block or mark a single line. The second <Ctrl G> with the cursor on a different line will mark a complete block. The block can be unmarked by pressing <Ctrl G> a third time.
<Ctrl L>	inserts a blank line below the current line and positions the cursor on it.

Key	**Function**
<Ctrl N>	locates the next occurrence of a earlier located string.
<Ctrl O>/<Insert>	toggles between the Overwrite and Insert data entry modes.
<Ctrl P>	locates the previous occurrence of a earlier located string.
<Ctrl R>	redisplays the screen and discards the most recent updates (since the last carriage return).
<Ctrl T>	copies the character at the corresponding cursor position on the line above the current line.
<Ctrl V>	performs jBC program indentations on the current screen window.
<Ctrl X>	exits the current record without writing away any updates. If the record has been changed within the current editing session then the editor will ask for confirmation to exit the modified record.
<Ctrl]>	inserts the field value delimiter character.
<Ctrl \>	inserts the field sub-value delimiter character.
<Enter>	opens a new line. Any characters on the current line after the current cursor position are moved to the start of the new line.

COMMAND LINE OPERATIONS

To enter the command line from the jED edit mode, press the <Esc> key, or one that has been reconfigured to perform the same action.

LEAVING THE EDITOR

There are several options available for exiting a file or record being edited. It can be deleted, stored in its latest form, keeping all the changes made in the current editing session, or it can be stored as it existed before the edit session began.

ABANDON EDIT AND START NEW SESSION

The **E** command will abandon the current edit (you will be asked to verify leaving a changed record) and edit the specified record(s).

The command syntax is as follows:

E unixfile
E filename record

If the form filename record is used, then the filename should be the name of a jBASE file. You can also specify the pathname of a standard UNIX file with the unixfile form. Note that wildcard characters such as asterisk (*) are not expanded by the **E** command, and that you must the jBASE file name again, even if you are currently editing a record from within that file.

DELETE FILE OR RECORD

The command syntax is as follows:

FD {options}

options can be **K**, **T** and/or **O**. See the Command Options topic for details.

This command deletes the file or record and releases any lock set. Before it does so, the user is prompted for confirmation. The edit session then terminates, or continues with the next record if this choice is in effect.

EXIT AND UPDATE

The command syntax is as follows:

FI {options} {unixcommand}

FI writes the updated version of the file or record back to disk and releases any lock set. The edit session then terminates, or continues with the next record, if this choice is in effect.

options are **B, K, R** and **T**. See Command Options for details.

unixcommand specifies a UNIX command to ,be executed on exiting the editor.

EXIT AND DISCARD

The command syntax is as follows:

EX {options}

EX leaves the file or record as it was at the start of the session, and releases any lock set. If updates have been made you will be prompted for confirmation before the updates are discarded. The edit session then terminates, or continues with the next record, if this choice is in effect.

options are **K**, **T** and **O**. See Command Options for details.

UPDATE WITHOUT EXIT

The command syntax is as follows:

FS {options} {unixcommand}

This command writes the updated file or record to disk, then returns to the editing session at the point where it left off.

options are **B** and **R**. See Command Options for details.

unixcommand specifies a UNIX command to be executed on exiting the editor.

DISPLAY RECORD IN HEXADECIMAL

The command syntax is as follows:

HX
HEX

This command acts as a toggle such that each iteration of the command turns the hexadecimal display on or off depending on its previous state. The HX (or HEX) command is only used for display, the record is not stored as it appears in hexadecimal.

COMMAND OPTIONS

R specifies that, after the file has been written to disk, it should be executed. Additional parameters can be added to this option and passed to the program. The editor issues the command filename {parameters} to execute the program. Note that the .b suffix is removed.

K or **T** option specifies that if the editor was working from a list of records, the list should be discarded and that the editor should exit directly to the shell (or to the calling process).

O specifies that the confirmation request normally issued with the FD and EX commands should be suppressed.

The R option is particularly useful to jBC programmers.

EXAMPLES

FIK
Exits the record and writes it to disk. If in the middle of a list of records being edited, the list is abandoned and the editing session is terminated.

FDO
Delete the current record being edited. The normal confirmation of this action is not given.

LOCATING STRINGS

The editor allows the user to search and locate any string held in the body of the text being edited.

There is also a keystroke command sequence (default <Ctrl N>) to allow the user to find the next occurrence of the string used in the previous locate command.

The locate command syntax is as follows:
L{nnn}dstring{doption}

nnn is the numeric value of the number of lines to search from the cursor position. If omitted, the search continues to the end of the file or record. If this optional parameter has been specified then all occurrences of the string will be located over the specified number of lines. If only a single occurrence is found then the cursor is placed at this point in the file. If multiple occurrences of the string are found then each one is listed below the editing screen.

d is the delimiter used to enclose the string to be located. It can be any character that does not form part of the string.

string is the string to locate.

option can be one or more of the following:
F specifies that the search is to begin at the start of the file or record.
C performs a case insensitive search. Otherwise the search defaults to match the cases as provided in the string.

EXAMPLES

L/report
Searches the record from the current position for the string "report" and halts at the first occurrence found, with the cursor at the start.

L9 FORM
Search the next 9 lines and locate all occurrences of the string "FORM".

L/STARS/F
Searches from the first line of the file to find the first

occurrence of the string "STARS". This line is placed at the top of the screen.

L/acropolis/C
Locates the first occurrence of the string "acropolis" with the letters in upper or lower case.

REPLACING STRINGS

The editor allows the user to replace any occurrence of a string on any line with another from the command line. This is in addition to the overwrite mode.

The command syntax is as follows:
R{U}{nnn}dstring1dstring2{doption}

U replaces ALL occurrences of string1 with string2 on the current line only.

nnn is a numeric value for the number of lines, starting from the current one, over which to perform the replace operation. If this optional parameter is specified and more than a single occurrence of string1 is found then all replacements are listed beneath the current editing screen. d is the delimiter character used to separate the string values. It can be any character not in either of the strings.

string1 is the string that is to be replaced.

string2 is the replacement string. This can be shorter or longer than the original.

option can be one or more of the following:
F executes the replace command from the first line of the file or record.
* replaces ALL occurrences of string1 with string2 on the current line.
nnnnumeric value for the number of times to repeat the replace operation on the current line.

EXAMPLES

R/ABC/DEF
Replaces the first occurrence (reading from the left) of ABC in the current line with DEF.

R9/*/!
Replace on the next 9 lines, the first occurrence on the line of

"*" with a "!". The changed lines are displayed before moving on.

RU9/*/!
Replace any occurrence of "*" with "!" over 9 lines (the current line and the next 8).

R999//*/F
Place a "*" character at the start of every line starting from the first. All lines changed are shown before returning to the original line.

R/^/AM/*
All occurrences of the "^" character on the line are replaced with "AM".

R9/*//
Removes (replaces with null) the first occurrence of "*" on the next 9 lines.

R/x//10
Removes the first 10 "x" characters on the current line.

COPYING, PASTING AND CUTTING BLOCKS OF TEXT

The editor allows the user to copy or move blocks of text from one location to another within the current record being edited. It is also possible to copy from another UNIX file or jBASE record. Before a block can be moved or copied, it has to be marked or highlighted. Marked lines have their line numbers replaced by the characters **++++**.

MARKING TEXT

Text can be marked whilst in edit mode by using the appropriate keystroke command (default <Ctrl G>) to mark the start and end of the block.

To highlight a block, move the cursor to the first line to be highlighted and press <Ctrl G> (or the reassigned ones). Then move the cursor to the last line to be highlighted and again press the <Ctrl G> The start and end lines can be marked in any order.

To cancel the marked text, simply press <Ctrl G> again, which will remove the markers.

Once the text is marked the cursor should be positioned on the line to which the text is to be copied or moved before invoking the command line or key sequence required.

COPYING MARKED TEXT

Once marked, text can be copied by moving the cursor to the target line, entering command mode, then using the copy commands provided. To copy text to the line before the current one, use the **CB** command. To copy to the line following the current one, use the CA command.

The syntax for both commands is the same:
CB{nn}
CA{nn}

The optional **nn** parameter is a numeric value that gives the number of copies of the marked text to transfer. This is particularly useful for the creation of a large quantity of repetitive text.

MOVING HIGHLIGHTED TEXT

Commands used to move highlighted text are MB to move to the line before the current one, and MA to move to the line following the current one.

The syntax for both commands is the same:

MB
MA

The text will be physically deleted from the original position. It is not valid to move text within the highlighted block.

MERGING TEXT FROM ANOTHER RECORD

Using the jED editor it is possible to merge data from any file or record into the current one.

The command to achieve this is:
MERGE

This is achieved by the following command sequence:
Position the cursor one line above the desired position of the merged text.
Spawn a new editor session using the ! command (detailed later). For example, "!jed record", or any other valid jed syntax. This will execute another editing session, placing the current one in the background.

Mark the block of text you wish to merge, then from the command line, issue the MERGE command.

The newly spawned editing session will be exited and control will be passed back to the original edit session. The merged text will then be copied into the record before the current line.

DELETING MARKED TEXT

The command **DB** deletes the marked text. The position of the cursor or portion of the record being displayed has no effect on the action.

jBC LINE INDENTATION

The jED editor has the capability of formatting lines of jBC program code with appropriate indentation and so make it more readable. The commands available and their syntax are described below.

BI{nn}
Formats the entire record as jBC code by adding indentations in appropriate places. The value nn gives the number of space characters per indentation (maximum 20), and defaults to 3 if omitted.

BION{nn}
Turns on the automatic indentation for jBC source code. Equivalent to using the B option with the jed command. The value nn gives the number of space characters per indentation, and defaults to the value used with the **B** option, or the value used in the last **BI** command.

BIONA{nn}
This command is the same as the BION command, except that an alternative form of indentation is used for the CASE statement. It is equivalent to using the **A** option with the jed command when opening an editing session.

BIOF{F}
Turns off the automatic indentation for jBC source code. It is equivalent to not using an indent option when opening an editing session.

MISCELLANEOUS COMMANDS

DE{nnn}
Deletes the number of lines specified by nnn, starting from the current cursor position. If nnn is omitted it defaults to a value of one line.

S?
Displays the size of the record being edited in bytes. It includes field delimiter marks in the body of the record.

!{command}
Executes command. Can be any valid UNIX or jBASE command.

!!
Re-executes the command specified in the most recent ! command executed.

U{nn}
Scrolls the screen up by nn lines. If omitted, nn defaults to one line.

D{nn}
Scrolls the screen down by nn lines. If omitted, nn defaults to one line.

I{nn}
Inserts nn blank lines after the line holding the cursor. If omitted, nn defaults to one line.

nn
Positions the cursor on line nn, which is positioned at the top of the screen if the number of remaining lines still allows a screen"s worth of data to be displayed.

IN
Equivalent to the <F10> key.

IP
Equivalent to the <F9> key.

?
Displays the main help screen menu.

CHANGING jED COMMAND KEYS

The keystrokes used for jED editor commands are configured using the UNIX terminfo terminal characteristic database.

TERMINFO FOR ALTERING JED KEYSTROKES

Terminfo is a UNIX database that describes the capabilities of terminals and their keyboards. Terminal capabilities are defined for how operations are performed, any padding requirements, and the initialization sequences required for each function. The terminfo system is comprehensively documented within the standard UNIX documentation.

The terminfo data is used by utilities such as vi and jED to allow them to work on entirely different terminals without needing to set up or change parameters for each one.

The terminfo data can usually be found in the /usr/lib/terminfo directory.

Terminfo entries consist of a number of fields delimited by a comma.

Embedded whitespace characters are ignored.

The first line of each description gives one or more names (separated by a | character) by which the terminal is known. Names should not contain space characters and at least the first 14 characters should be unique. The first name in the list is normally the shortest and is used when defining the terminal to UNIX, e.g. in setting the TERM environment variable.

The terminal name is followed by a list of capabilities that describe the functionality available with it.

There are three types of terminfo definitions:

Booleans to indicate what features of the terminfo system the particular terminal supports such as: margin; color; erase; tabs.
Numerics to indicate magnitudes such as numbers of columns per line,numbers of lines in the display.
Strings For example, cursor, italics, carriage return, keyboard definitions.

The jED editor is affected mainly by the definitions of key strokes in the strings section. If the terminfo definition for your terminal does not define the keyboard sequences for the jED editor (F1 - F10 keys, Cursor keys, etc.), you may change the definition yourself like this:

```
# TERM=myterm ; export TERM

# infocmp >termdef.myterm

# vi termdef.myterm
........add the new keystrokes and write back the new record

# tic termdef.myterm
```

Note that on many systems you will need to have super user permissions to execute the tic command.

Although terminfo is documented extensively it can become quite complex. jBASE Software would be pleased to help you to define terminfo entries for terminals. However, in most circumstances, the jkeys program will provide all functionality required.

ED

The ED editor provided with jBASE is a limited version of the familiar (but now superseded) ED editors.

You are strongly advised to use the jED editor in preference to ED - so strongly that we have not included any operating instructions here for the ED editor.

If you are only familiar with the old style ED editor, by all means continue to use it in your accustomed fashion - but please find a few minutes to review the operation of jED. You will find that the transition from ED to jED is very simple and you will be amply rewarded by adopting the much more efficient and friendly environment provided by jED.

jBASE OBjEX

Overview

The majority of computer systems are now connected to some kind of network, which enables that machine to "talk" in some kind of way to another machine. The amount of "talking" possible between the two machines depends on either proprietary mechanisms, whereby the two systems specifically know how to talk to each other, or by standard mechanisms, defacto or otherwise, whereby any machine should be able to talk to another using the "standard" component.

However even when data can be transferred between systems being able to exchange data between applications can still be a problem. One of the primary reasons behind jBASE OBjEX was to be enable data from Multivalued applications to be provided directly to common desktop applications like Excel and Word or any application coded in Visual Basic or Delphi.

The jBASE OBjEX interface is provided via an additional dynamic linked library, OBjEX.dll, and a type library, OBjEX.tlb. The OBjEX interface provides the developer with a mechanism whereby data can be retrieved or updated directly from an OLE, Object Linking and Embedding, compliant application. For example, an Excel macro can be coded using jBASE OBjEX to extract data from a jBASE hash file directly into an Excel spreadsheet. As jBASE OBjEX accesses data via the jBASE jEDI layer any database, for which a jEDI driver exists, can be accessed using jBASE OBjEX. For example jBASE Hash files, NT/95 directories or SQL databases.

jBASE OBJECT

The jBASE object represents the jBASE engine. As the top-level object, it contains and controls all other objects in the hierarchy of objects.

The jBASE object is used to open files and to perform certain jBASE operations on the built-in types. For ease of maintainability you are advised to use the name jB for all references to the jBASE object. This documentation will use jB whenever a reference is made to the jBASE object

CREATION

To create the jBase object from an early binding controller such as Visual Basic 4.0, use the following VB statement:

Public jB As New jBase

Alternately, when using a late binding controller such as VBA then use:

Public jB As Object
Set jB = CreateObject("OBjEX.jBaseObject.3")

NOTES

There can only be one jBASE Object in existence in a process at any one time, any attempt to create an additional one will return a pointer to the original object.

When creating a jBase object using CreateObject or the equivalent in other languages you can either default to using the latest installed version of OBjEX or specify a specific version by appending a period and the major version number as a single digit to the object name e.g.:

```
Set jB = CreateObject("OBjEX.jBaseObject.3")
```

METHODS

CHANGE METHOD

Returns a string with all occurences of a substring replaced with another.

COMMAND SYNTAX

jB.Change(Source, FromString,ToString)

Source	A string expression that evaluates to the string to be modified.
FromString	A string expression that evaluates to the substring to be replaced
ToString	A string expression that evaluates to the substring to use as a replacement

REMARKS

This method returns a modified string, it does not update the source string.

CONNECT METHOD

Connects to a jBASE Database.

COMMAND SYNTAX

Set jConnectionobject = jB.Connect([Target[,Options]])

jConnectionobject	A variable that will be set to refer to the target database.
Target	A string expression that represents the connection to a machine on the same network. The expression can be the name of the machine or an IP address. To connect to the local machine omit this parameter. Note that it should not be necessary to open remote files by connecting to a remote machine. Remote files should be opened from the local machine either through jRFS, F-pointers, mapped network drives or UNC paths. Remote connections are only practical when there is some subroutine or EXECUTE statement which must be run on the remote system.
Options	A string expression that is reserved. This optional value should be omitted (or set up as an empty parameter) in this release of OBjEX.

CONVERT METHOD

Returns a string where all occurrences of a set of characters have replaced with the equivalent character in the replacement set.

COMMAND SYNTAX

jB.Convert(Source, FromString,ToString)

Source	A string expression that evaluates to the string to be modified.
FromString	A string expression that evaluates to the set of characters to be replaced.
ToString	A string expression that evaluates to the set of replacement characters.

REMARKS

Each character in the Source string is checked to see if it occurs in the FromString, if it does then it is replaced with the character in the same position in the ToString If there is no corresponding character in the ToString then the character is deleted from the string. If there are duplicate characters in the FromString then only the first character will be used. This method returns a modified string, it does not update the source string.

COUNT METHOD

Returns the number of times the delimiter string occurs in the source string.

COMMAND SYNTAX

jB.Count(Source, Delimiter)

Source	A string expression that evaluates to the string to be examined.
Delimiter	A string expression that evaluates to the desired delimiter.

REMARKS

If a multi-character delimiter is specified then they are only counted when they start after the end of the prior delimiter occurrence. ie: delimiters
cannot overlap.

DATE PROPERTY

Returns the current date in jBase internal format.

COMMAND SYNTAX

Object.Date()

REMARKS

The date is returned as an integer containing the number of days since December 31 1967.

DCOUNT METHOD

Returns the number elements of a string that are separated by a delimiter string.

COMMAND SYNTAX

jB.DCount(Source, Delimiter)

Source	A string expression that evaluates to the string to be examined.
Delimiter	A string expression that evaluates to the desired delimiter.

REMARKS

If a multi-character delimiter is specified then they are only counted when they start after the end of the prior delimiter occurrence. ie: delimiters cannot overlap. If the source string is NULL then the function will return a value of 0, in all other cases this method will return one greater than the Count method would.

FIELD METHOD

Returns multi-character delimited substrings from within a string.

COMMAND SYNTAX

jB.Field(Source,Delimiter,Occurrence,Count,BeforeCol,AfterCol)

Part Description

Source The string to search for the substring.
Delimiter The string that delimits substrings.
Occurrence The optional starting substring number. (default 1)
Count The optional count of substrings to extract. (default 1)
BeforeCol An optional long variable that returns the character

position 1 before the extracted substring(s).
AfterCol An optional long variable that returns the character position 1 after the extracted substring(s).

REMARKS

This method encompasses the functionality provided by the jBC statements "Field()", "Col1()", "Col2()" and Group()"

ICONV METHOD

Applies a jBase input conversion code to the supplied data.

COMMAND SYNTAX

```
destvar = Object.IConv( sourcevar, convcode )
```

Part Description

destvar The name of a Variant that receives the data from the Conversion.
Object The jBase Object or a jConnection Object.
sourcevar A variant that contains the data to be converted.
Convcode A string expression that evaluates to the desired conversion code.

NOTES

For a list of available conversion codes, see OConv Method

INDEX METHOD

Returns the position of a character or characters within a string

COMMAND SYNTAX

Position = jB.Index(Source,SubString,Occurrence)

Source	The string that you wish to search.
SearchString	The sub string to find.
Occurrence	A numeric expression that is the occurrence to be found.

REMARKS

The function will return the character position of the first character of the specified occurrence of the sub string or 0 if it is not found.

MATBUILD METHOD

Creates a jDynArray from an Array of Variants.

COMMAND SYNTAX

Set jDynArrayVar = jB.Matbuild(Source)

Part Description

jDynArrayVar The name of a jDynArray variable that receives a reference
to the created object.
Source A Variant that contains the source data.

REMARKS

Matbuild takes a variant containing a string or an array and creates a
jDynArray object from it.
If the variant is an array then each element of the array becomes a field in
the created object. If any element of the array is itself an array, then each
element in that array becomes a value in that field. The process is repeated
for subvalues.
If the variant is a string , or can be coerced to a string, then a jDynArray
object containing a single field is created.

OCONV METHOD

Applies a jBase conversion code to the supplied data.

COMMAND SYNTAX

destvar = Object.OConv(sourcevar, convcode)

Part Description

destvar The name of a Variant that receives the data from the Conversion.
Object The jBase Object or a jConnection Object.
sourceexp A variant that contains the data to be converted.
Convcode A string expression that evaluates to the desired conversion code.

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 returned in the form ddcmmcyyyy. If it is not specified, the month name is returned in abbreviated form.

DI Allows 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 Windows in which jBASE is running.

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.

MCB Returns just the alphabetic and numeric characters from input string.

MC/B Removes the alphabetic and numeric characters from 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 chars.

MCU Converts all lower case characters in the string to upper case chars.

MCT Capitalizes each word in the input string; i.e.: JIM converts to Jim.

MCP{c} Converts all non printable characters to a tilde character
"~"
in the input string. If the character "c" is supplied then this
character
is used instead of the tilde.

MCPN{n} In the same manner as the MCP conversion, all non
printable characters are replaced. However, the replacing
character is
followed by the ASCII hexadecimal value of the character that was
replaced.

MCNP{n} Performs the opposite conversion to MCPN. The ASCII
hexadecimal value following the tilde character is converted back
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 character

TIME PROPERTY

Returns the current time in jBase internal format.

COMMAND SYNTAX

Object.Time()

REMARKS

The time is returned as an integer containing the number of
seconds since midnight.

TRIM METHOD

Returns a string with certain occurrences of a character removed.

COMMAND SYNTAX

Result = jB.Trim(Source, Flags,Character)

Source	A string expression that evaluates to the string to be trimmed.
Flags	An optional expression that defines the type of trim to perform.
Character	An optional character to specify the character to remove.

REMARKS

The default operation is to Trim leading, trailing and redundant blanks. The optional parameters can be used to modify the type of trimming performed. The possible values for the Flags parameter are:

Symbol Action

TRIM_LEADING Remove leading blanks.

TRIM_TRAILING Remove trailing blanks.

TRIM_REDUNDANT Replace consecutive blanks with single blank

TRIM_ALL Remove all blanks.

These values can be added together to achieve a combination of actions. If the optional Character is specified then the actions will be performed on occurrences of that character rather than blank. Note : a possible enhancement is to default to all white space rather than just blank so specify the blank character if this would be a problem. This method returns a modified string, it does not update the source string.

For Compatibility with OBjEX 1.0 the following jConnection methods are supported on jBase Objects

CREATEFILE METHOD

Creates a new file.

COMMAND SYNTAX

Object.CreateFile filename,options

Part Description

Object The jBase Object or a jConnection Object.
jBaseobject A variable of jBase object data type that represents the jBase object. filename A string expression that is the name of a file and may include the path to the file.
options An optional string expression that supplies any desired modifiers to the file create processor. See jBase documentation for details.

OPEN METHOD

Opens a specific file and, if successful, returns a reference to it.

COMMAND SYNTAX

Set jEDIobject = Object.Open(filename)

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are opening.
Object The jBase Object or a jConnection Object.
filename A string expression that is the name of a file and may include the path to the file.

JCONNECTION.OBJECT

The jConnection Object represents a connection to a specific machine on the network.

CALL METHOD

Calls a cataloged jBC subroutine.

COMMAND SYNTAX

jConnectionObject.Call subroutine, parameters.

Part Description

subroutine The name of a catalogued jBC subroutine.
parameters Up to 20 comma separated parameters to the called subroutine.

REMARKS

The number of parameters specified must match the number of parameters
expected by the called routine.

Parameters

- The number of parameters specified must match the number of parameters
  expected by the called routine.
- All parameters are passed by reference and can by modified by the jBC
  subroutine unless they are explicitly passed by value by the caller.
- To pass by value in Visual Basic, enclosing the parameter in parentheses
  will create a temporary variable to hold the result of the expression in
  parentheses.
- Parameters can be any of the valid OLE types including references to objects. OBjEX will perform the necessary conversions

  between OLE and jBC variable types.

CREATEFILE METHOD

Creates a new file.

COMMAND SYNTAX

Object.CreateFile filename,options

Part Description

Object The jBase Object or a jConnection Object.
jBaseobject A variable of jBase object data type that represents the
jBase object.
filename A string expression that is the name of a file and may include the path to the file.
options An optional string expression that supplies any desired modifiers
to the file create processor. See jBase documentation for details.

DATE PROPERTY

Returns the current date in jBASE internal format.

COMMAND SYNTAX

Object.Date()

REMARKS

The date is returned as an integer containing the number of days since
December 31 1967.

DELETELIST METHOD

Deletes a previously stored list.

COMMAND SYNTAX

jConnectionObject.DeleteList Name

Part Description

Name A string expression that evaluates to the name of the list to be
deleted.

REMARKS

Lists are saved in the jBase work file.

EXECUTE METHOD

Executes any other program from the command line.

COMMAND SYNTAX

```
jConnectionObject.Execute
CommandLine,Options,[,Setting[,Capturing[,RtnList[,PassList]]]]
```

Part Description

CommandLine A string expression that evaluates to the command to be
executed
Options A numeric expression available for future options - Use zero for now.
Setting A variable that will capture the output of any error messages
generated by the command
Capturing A variable that will capture the terminal output generated
by the command.
RtnList A variable that will reference a jSelectList Object created
by the command.
PassList The name of a jSelectListobject to pass to the command.

GETLIST METHOD

Retrieves a previously stored list.

COMMAND SYNTAX

```
Set jSelectList object = jConnectionObject.GetList( Name)
```

Part Description

Name A string expression that evaluates to the name of the list to be retrieved.
jSelectList object A variable of jSelectList data type that will be set up to reference the retrieved select list.

REMARKS

Lists are saved in the jBase work file.

ICONV METHOD

Applies a jBase input conversion code to the supplied data.

COMMAND SYNTAX

destvar = Object.IConv(sourcevar, convcode)

Part Description

destvar The name of a Variant that receives the data from the Conversion.
Object The jBase Object or a jConnection Object.
sourcevar A variant that contains the data to be converted.
Convcode A string expression that evaluates to the desired conversion code.

NOTES

For a list of available conversion codes, see OConv Method for jBASE
OBJECT.

OConv Method

Applies a jBase conversion code to the supplied data.

COMMAND SYNTAX

destvar = Object.OConv(sourcevar, convcode)

See Oconv Method for jBASE OBJECT

Open Method

Opens a specific file and, if successful, returns a reference to it.

COMMAND SYNTAX

Set jEDIobject = Object.Open(filename)

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are opening.

Object The jBase Object or a jConnection Object.
filename A string expression that is the name of a file and may include the path to the file.

PRECISION PROPERTY

Returns or sets the current precision used for the Call method

COMMAND SYNTAX

```
precision =jConnectionObject.Precision()
jConnectionObject.Precision = precision
```

REMARKS

The precision is an integer between zero and nine. The default is four.

TIME PROPERTY

Returns the current time in jBase internal format.

COMMAND SYNTAX

```
Object.Time()
```

REMARKS

The time is returned as an integer containing the number of seconds since
midnight.

JEDI OBJECT

The jEDI Object represents an open file.

jEDI Objects are created by using the Open method of the jConnection object

They are closed when all references to them have been deleted.

METHODS

CLEARFILE METHOD

Deletes all records from the file.

COMMAND SYNTAX

jEDIobject.ClearFile

Part Description

jEDIobject A variable of a jEDI object data type. This represents the file to which you are referring.

Copy Method

Creates a copy of an object.

COMMAND SYNTAX

Set NewObject = Object.Copy()

Part Description

NewObject A variable to receive a reference to the newly created object.
Object A reference to the object to be copied.

NOTES

jEDI objects are handles to open files. Copying the jEDI object creates a new handle to the same file.

DELETE METHOD

Deletes specific record from a file.

COMMAND SYNTAX

jEDIobject.Delete(recordKey)

Part Description

jEDIobject A variable of a jEDI object data type that represents the file
that you are deleting from.
recordkey A string expression that is the name of the record to be deleted.

REMARKS

If the record cannot be deleted then an error status is returned.

READ METHOD

Reads a specific record from a file into a new jDynArray object.

COMMAND SYNTAX

Set jDynArrayobject = jEDIobject.Read(recordkey)

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are opening. jDynArrayobject A variable of jDynArray type that will be receive a reference to the resulting jDynArray object. recordkey A string expression that is the name of the record to be read.

REMARKS

If the record cannot be read then an error status is returned.

READU METHOD

Reads a specific record from a file into a new jDynArray object. If successful, the record is locked.

COMMAND SYNTAX

Set jDynArrayobject = jEDIobject.ReadU(recordkey)

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are opening.
jDynArrayobject A variable of jDynArray type that will be receive a reference to the resultant jDynArray object.
recordkey A string expression that is the name of the record to be read.

REMARKS

If the record cannot be read then an error status is returned.

READV METHOD

Reads a specific field from a file into a variable.

COMMAND SYNTAX

destvar = jEDIobject.ReadV(recordkey, field number)

Part Description

destvar A variable of type VARIANT.
jEDIobject A variable of a jEDI object data type that represents

the file that you are opening.
recordkey A string expression that is the name of the record to be read.
field number A numeric expression that is the field number you want to read.

REMARKS

If the record cannot be read then an error status is returned.

READVU METHOD

Reads a specific field from a file into a variable. If successful, the record is locked.

COMMAND SYNTAX

destvar = jEDIobject.ReadVU(recordkey, field number)

Part Description

destvar A variable of type VARIANT.
jEDIobject A variable of a jEDI object data type that represents the file that you are opening.
recordkey A string expression that is the name of the record to be read.
field number A numeric expression that is the field number you want to read.

REMARKS

If the record cannot be read then an error status is returned.

ReleaseAllLocks

Release all locks on records in a file.

COMMAND SYNTAX

jEDIobject.ReleaseAllLocks()

Part Description

jEDIobject A variable of a jEDI object data type that represents the file.

RELEASELOCK METHOD

Release the lock on a specific record in a file.

COMMAND SYNTAX

```
jEDIobject.ReleaseLock(recordkey)
```

Part Description

jEDIobject A variable of a jEDI object data type that represents the file.

recordkey A string expression that is the name of the record to be unlocked.

SELECT METHOD

Returns a jSelectList object that represents all the record keys in the file.

COMMAND SYNTAX

```
Set jSelectList object = jEDIobject.Select()
Set jSelectList object = jDynArrayobject.Select()
```

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are opening.
jSelectList object A variable of jSelectList data type that will be set up to reference the created select list.

WRITE METHOD

Write a jDynArray object into a file, releasing any locks held on the record.

COMMAND SYNTAX

```
jEDIobject.Write recordkey, jDynArrayobject
```

Part Description

jEDIobject A variable of a jEDI object data type that represents the file to which you are writing.
jDynArrayobject A variable of jDynArray data type that will be written to the file.
recordkey A string expression that is the name of the record to be written.

REMARKS

If the record cannot be written then an error status is returned.

WRITEU METHOD

Write a jDynArray object into a file retaining any locks on the record.

COMMAND SYNTAX

```
jEDIobject.WriteU recordkey, jDynArrayobject
```

Part Description

jEDIobject A variable of a jEDI object data type that represents the file to which you are writing.
jDynArrayobject A variable of jDynArray data type that will be written to the file.
recordkey A string expression that is the name of the record to be written.

REMARKS

If the record cannot be written then an error status is returned.

WRITEV METHOD

Update a specific field in a record, releasing any locks held on the record.

COMMAND SYNTAX

```
jEDIobject.WriteV recordkey, fieldnumber, replacementvalue
```

Part Description

jEDIobject A variable of a jEDI object data type that represents the file that you are updating.

recordkey A string expression that is the name of the record to be updated.

fieldnumber A numeric expression that specifies the field to be updated.

replacementvalue An expression that is the value to replace the specified field.

REMARKS

If the record cannot be read or written then an error status is returned.

WRITEVU METHOD

Update a specific field in a file, retaining any locks held on the record.

COMMAND SYNTAX

jEDIobject.WriteVU recordkey, fieldnumber, Replacement Value

Part	Description
jEDIobject	A variable of a jEDI object data type that represents the file that you are updating.
recordkey	A string expression that is the name of the record to be updated.
fieldnumber	A numeric expression that specifies the field to be updated.
Replacement Value	An expression that is the value to replace the specified field.

REMARKS

If the record cannot be read or written then an error status is returned.

JDYNARRAY OBJECT

The jDynArray Object is a variable length string that uses embedded delimiter characters to implement a 3 dimensional multi valued record structure.

Each record comprises any number of Fields. Each field can contain any number of Values. Each value can contain any number of SubValues.

jDynArray Objects are either the result of methods on the jEDI Object or can be created dynamically.

METHODS

CHANGE METHOD

Returns a string with all occurrences of a sub string replaced with another.

COMMAND SYNTAX

jDynArrayobject.Change(Source, FromString,ToString)

jDynArrayobject The name of a jDynArray object that you wish to update.

DEL METHOD

Deletes an element from a jDynArray object.

COMMAND SYNTAX

jDynArrayobject.Del Fieldno[,ValueNo[,SubValueNo]]

jDynArrayobject The name of a jDynArray object that you wish to update.
Fieldno is a numeric expression that is the field number to be deleted.
ValueNo is an optional numeric expression that is the value number to be deleted.
SubValueNo is ann optional numeric expression that is the subvalue number to be deleted.

REMARKS

Invalid numeric values for the expressions are ignored without warning. The command operates within the scope specified, i.e. if only a field is specified, then the entire field (including its multivalues and subvalues) is deleted. If a subvalue is specified, then only the subvalue is deleted, leaving its parent multivalue and field intact.

EXTRACT METHOD

Returns the value of a specific field, value or subvalue.

COMMAND SYNTAX

DestVar = jDynArrayVar.Extract(Fieldno,[ValueNo,[SubValueNo]])

DestVar is the name of a String or Variant that receives the extracted data.
JDynArrayVar is the name of a jDynArray you wish to extract data from.
Fieldno is a numeric expression that is the field number to extract.
ValueNo is an optional numeric expression that is the value number to extract.
SubValueNo is an optional numeric expression that is the subvalue number to extract.

REMARKS

If the extracted field or value comprises multiple values or subvalues, then the Dest Var will contain embedded delimiter characters.

EXTRACTDYNARRAY METHOD

Returns the value of a specific field or value as a jDynArray object.

COMMAND SYNTAX

Set jDynArrayVar = jDynArrayobject.ExtractDynArray(Fieldno,[ValueNo])

jDynArrayVar is the name of a jDynArray variable that receives the extracted data.
jDynArrayobject is the name of a jDynArray object that you wish to extract the data from.
Fieldno is a numeric expression that is the field number to extract.
ValueNo is an optional numeric expression that is the value number to extract.

FIND METHOD

Allows the location of a specified string within a jDynArray object.

COMMAND SYNTAX

jDynArrayobject.Find(SearchString,OccurenceNo,fieldVar,[ValueVar,[SubValueVar]])

jDynArrayobject is the name of a jDynArray object that you wish to search.
SearchString is the string to find.
OccurrenceNo is a numeric expression that is the occurrence to be found.
FieldVar is the name of a numeric or Variant that receives the field position in which the string was found.
ValueVar is an optional name of a numeric or Variant that receives the value position in which the string was found.
SubValueVar is an optional name of a numeric or Variant that receives the sub-value position in which the string was found.

REMARKS

The function has a boolean return type that is set true if the string is found. This allows the method to be used in an IF statement.

FINDSTR METHOD

Allows the location of a specified substring within a jDynArray object.

COMMAND SYNTAX

jDynArrayobject.FindStr(SearchString,OccurenceNo,FieldVar,[ValueVar,[SubValueVar]])

jDynArrayobject is the name of a jDynArray object that you wish to search.
SearchString is the substring to find.
OccurenceNo A numeric expression that is the occurrence to be found.
FieldVar The name of a numeric or Variant that receives the field **position** is the position in which the substring was found.
ValueVar is an optional name of a numeric or Variant that receives the value position in which the substring was found.
SubValueVar is an optional name of a numeric or Variant that receives the sub-value position in which the substring was found.

REMARKS

The function has a boolean return type that is set true if the substring is found. This allows the method to be used in an IF statement.

INS METHOD

Inserts elements into a jDynArray.

COMMAND SYNTAX

jDynArrayobject.Ins ValueToInsert, Fieldno[,ValueNo[,SubValueNo]]

jDynArrayobject is the name of a jDynArray object into which you wish to insert the data.
ValueToInsert is the variable containing the value to be inserted.
Fieldno is a numeric expression that is the field number to insert before.
ValueNo is an optional numeric expression that is the value number to insert before.
SubValueNo is an optional numeric expression that is the subvalue number to insert before.

REMARKS

The ValueToInsert can be either a jDynArray object or any other type of variable that can be coerced to a string . Specifying a negative value to any of Fieldno, ValueNo, or SubValueNo will cause the element to appended as the last Field, Value, or Sub-Value rather than at a specific position. Only one of the expressions may be negative, otherwise the first negative value is used correctly but 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.

LOCATE METHOD

Finds the position of an element within a specified dimension of a jDynArray object.

COMMAND SYNTAX

jDynArrayobject.Locate(searchstring, positionvar [, ordercode[,startposition[, Fieldno[, Valueno]]]])

jDynArrayobject is the name of a jDynArray object that you wish

to search.
searchstring is the string to find.
positionvar is the name of a long variable that receives the position.
ordercode is an optional string expression that specifies the ordering.
startposition an optional numeric expression that specifies the field, value or subvalue position from which the search will begin.
Fieldno is an optional numeric expression that specifies the field position to search for a value.
Valueno is an optional numeric expression that specifies the value position to search for a sub value.

REMARKS

Specifing ordercode 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 ordercode 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

Positionvar will be set to the position of the Field, Value or Sub-Value in which searchstring was found if indeed, it was found. If it was not found and ordercode was not specified then positionvar will be set to one position past the end of the searched dimension. If ordercode did specify the order of the elements then positionvar will be set to the position before which the element should be inserted to retain the specified order.

The function has a boolean return type that is set true if the string is found. This allows the method to be used in an IF statement.Matparse Method

MATPARSE METHOD

Builds an Array from a jDynArray.

COMMAND SYNTAX

```
DestVar = jDynArrayVar.Matparse
```

DestVar is the name of a Variant that receives the Array.
JDynArrayVar is the name of a jDynArray the you wish to create an Array from.

REMARKS

Matparse returns an array of variants with one element for each field in the jDynArray object. If any field contains multiple values then the corresponding field will contain an array of variants with one element for each value. The process is repeated for sub values. When the contents of a jDynArray object is a single field without any
embedded values,the DestVar is returned as a variant of type string . The Visual Basic function IsArray() is useful for determining the presence of embedded values.

REPLACE METHOD

Replaces a specified Field, Value, or Sub-Value in a jDynArray.

COMMAND SYNTAX

```
jDynArrayobject.Replace
ReplacementValue,Fieldno[,ValueNo[,SubValueNo]]
```

jDynArrayobject is the name of a jDynArray object that you wish to update.
ReplacementValue is the variable containing the data that is to replace existing data.
Fieldno is a numeric expression that is the field number to be replaced.
ValueNo is an optional numeric expression that is the value number to be replaced.
SubValueNo is an optional numeric expression that is the subvalue number to be replaced.

REMARKS

The ReplacementValue can be either a jDynArray object or any other type of variable that can be coerced to a string .

SELECT METHOD

Returns a jSelectList object that represents all the record keys in the file.

COMMAND SYNTAX

Set jSelectList object = jEDIobject.Select()
Set jSelectList object = jDynArrayobject.Select()

jEDIobject is a variable of a jEDI object data type that represents the file that you are opening.
jSelectList object is a variable of jSelectList data type that will be set up to reference the created select list.

SORT METHOD

Sorts the elements of a jDynArray.

COMMAND SYNTAX

jDynArrayobject.Sort

jDynArrayobject is the name of a jDynArray object that you wish to sort.

REMARKS

The SORT method will sort a dynamic array by the highest delimiter found in the array. This means that if the jDynArrayobject contains multiple fields, it will sort by field; but if it contains only sub-values, it will sort by sub-values.

TEXT METHOD

Converts jDynArray objects into text strings suitable for display.

COMMAND SYNTAX

StringVar = jDynArrayObject.Text()

NOTES

The text method is especially useful for populating list boxes since each attribute is separated by a newline sequence.

JSELECTLIST OBJECT

The jSelectList Object is created by the Select method of the jEDI Object. It holds a list of record keys that are used by the Readnext method to supply the next record key.

ReadNext Method

Retrieves the next element from a jSelectList object.

COMMAND SYNTAX

destvar = jSelectListobject.ReadNext

Part	Description
Destvar	The name of a string that receives the element.
jSelectListobject	The name of a jSelectListobject to operate on.

REMARKS

If there are no more elements in the list then ReadNext will return an error status.

WRITELIST METHOD

Stores a jSelectList in the jBase work file.

COMMAND SYNTAX

jSelectList object .WriteList Name

Part	Description
Name	A string expression that evaluates to the name of the list to be written.
jSelectListobject	The name of a jSelectListobject to operate on.

REMARKS

By default, lists are saved in the jBASEWORK file.

ERROR HANDLING

Errors are reported using the standard OLE / ActiveX reporting technique.

REMARKS

All jBase OBjEX methods return a standard result code to the calling program that indicates the success or failure of the method and the cause any errors.

VISUAL BASIC

In Visual Basic the result code is used to update the Err object and will cause certain actions to be taken according to the use of the On Error statement.

If you don"t use an On Error statement, any run-time error that occurs is fatal; that is, an error message is displayed and execution stops.

An "enabled" error handler is one that has been turned on by an On Error statement; an "active" error handler is an enabled handler that is in the process of handling an error. If an error occurs while an error handler is active (between the occurrence of the error and a Resume, Exit Sub, Exit Function, or Exit Property statement), the current procedure"s error handler can't handle the error. Control returns to the calling procedure; if the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no inactive, enabled error handler is found, the error is fatal at the point at which it actually occurred. Each time the error handler passes control back to the calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the Resume statement.

JBC AND VISUAL BASIC GUIDE

This table is a cross reference between jBC Basic commands and the equivalent functionality in Visual Basic using OBjEX. For each command that is applicable there is a reference to either a Visual Basic Intrinsic or to an OBjEX method.

JBC	Visual Basic with OBjEX
ABORT	Stop/End
ABS()	Abs()
ALPHA()	Like
ASCII	n/a
ASSIGNED()	IsEmpty() / IsObject
BITCHANGE	n/a
BITCHECK	n/a
BITLOAD	n/a
BITRESET	n/a
BITSET	n/a
BREAK	Exit Do
CALL	Call or to call a jBC Subroutine, Call Method
CASE	Select Case
CHAIN	n/a
CHANGE	Change Method
CHAR()	Chr()
CHDIR()	ChDir
CHECKSUM()	n/a
CLEAR	n/a
CLEARFILE	Clearfile Method
CLOSE	Set = Nothing
COMMON	jCommon
COL1() & COL2()	Part of Field Method
COLLECTDATA	n/a
CONTINUE	n/a
CONVERT	Convert Method
COS	Cos()
COUNT()	Count Method

CRT	n/a
DATA	n/a
DATE()	Date Property
DCOUNT()	DCount Method
DEBUG	n/a
DEFC	n/a
DEL	Del Method
DELETE	Delete Method
DELETELIST	Deletelist Method
DIMENSION	Dim
DTX()	Hex()
EBCDIC()	n/a
ECHO	n/a
ENTER	n/a
EQUATE	#Const
EXECUTE	Execute Method
EXIT()	End
EXP()	Exp()
EXTRACT	Extract Method
FIELD()	Field Method
FIND	Find Method
FINDSTR	FindStr Method
FOOTING	n/a
FOR	For
GETCWD()	CurDir()
GETENV()	Environ()
GETLIST	Getlist Method
GOSUB	Call()
GO(TO)	GoTo
GROUP()	Field Method
HEADING	n/a
ICONV()	IConv Method
IF	If
IN	n/a
INDEX()	Index Method

INPUT	n/a
INPUTNULL	n/a
INS	Ins Method
INT()	Int() or Fix()
LEN()	Len()
LN()	Log()
LOCATE	Locate Method
LOCK	n/a
LOOP	Do or While
MAT	For Each or Erase
MATCH(ES)	Like
MATBUILD	Matbuild Method
MATPARSE	MatParseDynArrays Method
MATREAD	Read Method followed by MatparseDynArrays Method
MATREADU	ReadU Method followed by MatparseDynArrays Method
MATWRITE	Matbuild Method followed by Write Method
MATWRITEU	Matbuild Method followed by Write Method
MOD() and REM()	Mod()
NOT()	Not
NULL	n/a
NUM()	Is Numeric()
OCONV()	Oconv Method
ON . . . GOSUB	On . . . GoSub
ON . . . GOTO	On . . . GoTo
OPEN	Open Method
OUT	n/a
PAGE	n/a
PERFORM	Execute Method
PRECISION	Precision Property
PRINT	n/a
PRINTER ON/OFF/CLOSE	n/a
PRINTERR	n/a

PROCREAD	n/a
PROCWRITE	n/a
PROGRAM	n/a
PROMPT	n/a
PUTENV()	n/a
PWR()	^ Operator
READ	Read Method
READLIST	Getlist Method
READNEXT	Readnext Method
READT	n/a
READU	ReadU Method
READV	ReadV Method
READVU	ReadVU Method
REGEXP	Like
RELEASE	ReleaseLock Method and ReleaseAllLocks Method
REMOVE	use Extract Method
REPLACE	Replace Method
RETURN	End Sub/Exit Sub
REWIND	n/a
RND	Rnd()
RQM	n/a
RTNDATA	n/a
SELECT	Select Method
SENTENCE	Command()
SEQ()	Asc()
SIN()	Sin()
SLEEP	Call Sleep
SORT()	Sort Method
SOUNDEX()	n/a
SPACE()	Space()
SQRT	Sqr()
STOP	Exit
STR()	String()/(1st character only)
SUBROUTINE	Sub
SYSTEM	n/a

TAN()	Tan()
TIME()	Time Property
TIMEDATE()	Now()
TRANSABORT	TransAbort Method
TRANSEND	TransEnd Method
TRANSQUERY	TransQuery Method
TRANSTART	TransStart Method
TRIM()	Trim Method
TRIMB()	Trim Method
TRIMF()	Trim Method
UNASSIGNED	IsEmpty()/IsObject
UNLOCK	n/a
WEOF	n/a
WRITE	Write Method
WRITELIST	Writelist Method
WRITET	n/a
WRITEU	WriteU Method
WRITEV	WriteV Method
WRITEVU	WriteVU Method
XTD()	n/a

TRAPS AND PITFALLS

Some of the common problems encountered by Basic programmers migrating to VB and OBjEX

ERROR HANDLING

It is important to read the VB documentation and understand the way that errors are handled in Visual Basic. In particular when you disable the default error handler (which aborts the program) ensure that you can handle all possible errors that can occur until you re-enable the default handler. One error to be particularly aware of is that any attempt to access an uninitialized object reference will cause an error rather than a familiar message of the form "Uninitialized variable - Zero Used"

Object References

Object variables that are declared in a VB program are just variables that can contain references to objects. They are effectively uninitialized variables until they are set to reference an object either as the result of a method on another object or as the result of a new operation. So make sure you do not attempt to use them as source operands until they refer to an object that actually exists.

THE SET STATEMENT

All references to objects have to be stored by the Set statement. If you omit the Set and create just an assignment statement then VB will attempt to assign the current default value of the object to the target variable.

EXAMPLE

```
Set MyDynArray = MyFile.Read(MyItemID)
```

Will make MyDynArray a reference to the jDynArray object that is created by the Read method. If MyDynArray was referencing an object
prior to the Set statement then that object would be released if there were
no other references to it.

```
MyDynArray = MyFile.Read(MyItemID)
```

Will assign the string value of the dynamic array to the MyDynArray variable. If the MyDynArray actually references a existing dynamic array then the existing data will be overwritten by the string data converted back into a dynamic array. In effect the following will occur:

Set TempjDynArray = MyFile.Read(MyItemID)	Creates a temporary object
TempString = TempjDynArray.Value()	Takes string value of object
MyDynArray.Value = TempString	Overwrites value with string

This may appear to be achieving the desired result but it will fail if the reference is null and is doing far more work than it needs to.

NULL REFERENCES VS NULL DYNAMIC ARRAYS

A Basic program would typically initialize dynamic arrays to, and test for, the null value in many circumstances. Using OBjEX , dynamic arrays are accessed through "references" that introduce a level of indirection creating two possible ways of representing null dynamic arrays.

1. As a null reference.
Methods that return jDynArray Object references will return a null reference when they fail to obtain the desired jDynArray (and cause an error). Code that tests for the existence of a record, for example, should use the IsObject() function to check for null references rather than looking for null dynamic arrays. The Set jDynArrayreference = Nothing statement should be used to set a reference to the null state.

2. As a reference to a jDynArray object with the value ".
Code that builds dynamic arrays from scratch should create the array using the new operator before accessing it.

```
Set jDynArrayreference = New jDynArray
```

This statement will release the current object (if any), create a jDynArray object and initialize it to the null state.

COMPONENTS

File	**Description**
OBjEX30.dll	OBjEX dynamic link library
OBjEX30.tlb	OBjEX type library
OBjEX30.gid	OBjEX unique identifier
OBjEX30.hlp	OBjEX help pages
OBjEX30.cnt	OBjEX help contents