jBASE Tools

Contents

Documentation Conventions

This manual uses the following conventions:

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

Syntax definitions and examples are indented for ease in reading.

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

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

Preface

This manual is one of a set of jBASE manuals, which are intended for use by System Administrators, developers and other end users who might need guidance or instruction on various languages, applications and/or front end systems which jBASE provides. If you have never used or are not familiar with jBASE, we recommend that you read the Overview Guide to familiarize yourself with the system.

Organization of This Manual

This manual is designed to help the end-users familiarize them selves with the functionality of the jBASE Editor. It contains details of how to edit and manipulate on screen data the use of the keyboard to move around the screen, jQL I-type support and the necessary syntax to enable use of jQL.
List processing commands appear in alphabetical order, each beginning on a new page. At the top of each page is the syntax for the Command List followed by a detailed description of its use, often including references to other command lists, which can be used with it or are helpful to know about. Examples illustrate the application of the command list in a program.

jBASE Editor (jed)

The jED editor is a full screen, context sensitive, screen editor designed specifically for ease of use and learning and is the preferred editing tool for the jBASE-operating environment and suited to the editing of jBASIC programs. 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 not provided by other editors. The Command keystrokes are fully configurable by each user, as is the keyboard. Therefore, jED is customizable to mimic many of the 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

jED

jED is a robust, terminal independent screen-editing tool.

JED COMMAND SYNTAX

```
JED Filename Item (Options JED Item )
```

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
- the Command line
- The data editing area, which fills the rest of the screen.

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 overwrites 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

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 at the last file in use with the cursor positioned wherever it was when the last edit session was closed. In other words, you can continue from where you last left off. If the file referenced by pathname does not exist the Command, jed

pathname, will either open an existing file or create a new one. The edit window displays the contents of the file. If you specify a list file, 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 record keys. If no record keys were specified, the jED editor will prompt for a list, else it will edit the list of record keys one after the other.
NOTE: that because the editor uses the jEDI interface to access the records, use it 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 jBASIC 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.

EXAMPLE

```
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 jBASIC 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 five spaces.

```
jed invoices.b subs.c
```

The jBASIC 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 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 has two different modes:

1. Command mode for entering editor Commands, and
2. 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

Use edit mode when entering or modifying 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.
<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.
<Ctrl N>	locates the next occurrence of an earlier located string.
<Ctrl O>/<Insert>	toggles between the Overwrite and Insert data entry modes.
<Ctrl P>	locates the previous occurrence of an 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 jBASIC 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 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 to exit a file or a current editing session. It can be:

1. Deleted
2. Stored in its latest form, under the current editing session
3. Stored, as it existed before the edit session began.

Abandon, Edit and Start a 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).

COMMAND SYNTAX

```
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 use 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.
Unix Command specifies a UNIX Command to be executed on exiting the editor.

Exit and Discard

COMMAND SYNTAX

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

COMMAND SYNTAX

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 upon 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. This option is particularly useful to jBASIC programmers.
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.

EXAMPLES

`FIK`

Exits the record and writes it to disk. If in the middle of editing a list of records, it abandons the list and terminates the editing session.

`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	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	The delimiter used to enclose the string to be located, which can be any character that does not form part of the string.
String	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, else 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 nine 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	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	The string that is to be replaced.

string2	The replacement string, which can be shorter or longer than the original.

options 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 nine lines (the current line and the next 8).

R999//*/F

Starting at the first line place a "*" character on every line; all modified lines 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 nine 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. You must mark or highlight a block before moving or copying it. 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 and press <Ctrl G> (or the reassigned ones). Move the cursor to the last line to 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, position the cursor 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. Use the CB Command to copy text to the line before the current line. To copy to the line following the current line, 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
```

The text is deleted from the original position. It is not valid to move text within the highlighted block.

Merging Text From Another Record

It is possible to merge current data using jED from any file or record by using the Command:

MERGE

Use 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 session into the background.
Mark the block of text you wish to merge, and 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.

jBASIC Line Indentation

The jED editor has the capability of formatting lines of jBASIC program code with appropriate indentation and so makes it more readable. The Commands available and their Syntax are described below.

BI{nn}

Formats the entire record as jBASIC code by adding indentations in appropriate places. The value nn gives the number of space characters per indentation (maximum 20), and defaults to three if omitted.

BION{nn}

Turns on the automatic indentation for jBASIC 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 jBASIC 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 in bytes the size of the record being edited; 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:

1. Booleans
Indicate what features of the terminfo system the particular terminal supports such as: margin; color; erase; tabs.
2. Numerics
Indicate magnitudes such as numbers of columns per line, numbers of lines in the display.
3. Strings

FOR EXAMPLE, cursor, italics, carriage return, and keyboard definitions
The jED editor is affected mainly by the definitions of keystrokes 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 consultants 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 - we have not included any operating instructions for the ED editor.

If you are only familiar with the old style ED editor, 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.

jSHELL

The jsh command invokes jSHELL - the jBASE shell. It can be invoked as your login shell by using the normal system administration software supplied with the platform, either via *.bat files (Windows) or *.profiles (UNIX).
jSHELL has been designed to ease migration from older systems, and to overcome some of the differences between various platform command line environments. The more primitive features seen on some older platforms (such as the “dot” command stacker) have been replaced with easier to use and equivalents that are more functional.
The most noticeable difference between jSHELL and other command line shells, such as the UNIX Korn shell (ksh), is that command line arguments such as “*” and “?” are not expanded by the shell but passed directly to the command that has been invoked. In same manner, quoted strings (such as “quoted string”) are passed directly to the command with quotes intact. This enables query language statements such as:

```
SSELECT file = “[SPROUT]” BY *A1 –
```

to be issued directly from jSHELL. If the same command were issued from the ksh Prompt, it would have to be issued as:

```
SSELECT file = \”[SPROUT]\” BY \*A1 –
```

to avoid the quotes being removed and the “*” being expanded by the Korn shell.
Beyond this convenient feature, jSHELL also offers many significant advantages over traditional shells and is easier to use. Some of the main features of the jsh are:

Easily customised command line prompt
If required commands can be passed to other shells
Easy command recall.
Easily identified special prompt when a select list is active
Long program names are automatically translated to their new names on SVR3.2 systems.
Run jCL programs directly from jBASE hashed files.
Command type-ahead is supported.

JSH COMMAND SYNTAX

```
jsh - -c command -s shell -p Prompt
```

Option	Description
-	Execute proc from MD/VOC file with same name as user login. (on

	UNIX the .profile and .jshrc files are processed)
-c command	Specifies that a jsh process should be spawned to execute command. When the command terminates, the jsh process will also terminate.
-s shell	Specifies which shell emulation to use when executing jsh. The jsh will default to the previous emulation used by the current port.
-p Prompt	Specifies the Prompt to be used while executing jsh.
-t	Opens the tty device and accepts commands from the keyboard when the jSHELL has been invoked to process a command input file. The default action is to exit the shell once the processing of the input file has been completed.
-z foreground, background	Select foreground and background screen colors (e.g. jsh -z foreground,background). Colors can be WHITE, YELLOW, MAGENTA, RED, CYAN, GREEN, BLUE or BLACK. On Windows 95/98, the defaults are BLUE foreground and WHITE background (i.e. jsh -z will be blue on white). On Windows NT/2000, colors can be globally set using the Console setup from the Control Panel or by selecting the Properties of a jShell shortcut.

NOTE: If the jsh command is issued without arguments, a jsh process is spawned and this process becomes your command shell. The jsh process will replace the current shell if it is invoked through the UNIX exec command.

USING JSH

To use this tutorial you will need to be logged in to your UNIX system and positioned at the shell prompt. If your user account has not been configured to run jSHELL by default, execute it and complete the following:

```
exec jsh –
```

NOTE: Some UNIX SVR4.x systems have there own shell called jsh. If the PATH environment variable includes the directory containing the SVR4x native jsh before the jBASE release directory path, you may execute the native version rather than the jBASE version. You can either change the PATH list or use the absolute path name to the jsh executable.
The default jSHELL Prompt should now appear:

```
jsh user cwd -->
```

User is your login name and cwd is your current working directory. For this exercise, we will assume that your login name was jBASE and that your current working directory is the home directory for jBASE. In this case, your Prompt will look like this:

```
jsh jBASE ~ -->
```

The tilde character (~) is a shorthand method of referring to your home directory. The shell expands this character to the full path name of your login home directory before executing commands. If you had changed to a sub-directory called “source” your Prompt would now look like this:

```
jsh jBASE ~ --> cd source
jsh jBASE ~/source -->
```

You can change the primary and/or secondary default Prompts by using the following commands:

```
jsh jBASE ~ -->set jps1 newPrompt
jsh jBASE ~ -->set jps2 newPrompt
```

NOTE: The secondary prompt is only displayed for an active select-list, i.e., after a SELECT, GET-LIST, QSELECT, BSELECT, etc.

newPrompt is a string defining the new Prompts. The string can contain terminal control characters such as a bell character by specifying special character sequences in the newPrompt string. The character sequences allowed are:

Sequence	Replaced With
$EnvVar	The value of the specified environment variable
$%a	The user account name
$%m	The phrase “(Cmd)” if the shell is in command mode
$%n	The new line sequence
$%C	The current working directory
$%c	The current working directory with any portion matching the home directory replaced with ~
$%p	The port number
$%e	The entry number in the stack currently being edited
$%d	The current date in dd mmm yyyy format
$%t	The time of day in hh:mm:ss format
$%u	The host name as defined by the UNIX command uname (UNIX only)
$%y	The tty name (UNIX only)
$%s	The name of the jshelltype that will execute the commands at the Prompt
Chars	all other characters are taken as literals and included in the Prompt

The shell operates in two distinct modes each having a separate function.
command mode is used to issue all commands to jsh itself
operating mode is used to issue all commands to the system.

There is only one command available in the current implementation of jsh - the / command. This character introduces a search string to jsh. The search string is compared against every command in your command history and if a match is found, the command is recalled as the current command, just as if you had typed it in again. Pressing the escape key on your keyboard normally enters command mode.
If you include the $%m sequence when you configure the Prompt, the Prompt will change to indicate whether you are in the shell command mode. For example, if the Prompt has otherwise been left in its default state, the following sequence will locate the last cd command in your command history.
NOTE: the appearance of the “(Cmd)” string as part of the Prompt on the middle line:

```
jsh ~ --><Esc>
jsh ~ (Cmnd) -->/cd
jsh ~ -->cd source
```

Two other keystrokes within jsh allow you to recall up to 50 previous commands. They are:
<Ctrl P> Goto previous command

<Ctrl N> Goto next command

Using these two keystrokes, you can retrace your commands by stepping backwards or forwards one command at a time. If you need to change any part of a command line, it is very easy as the jsh supports command line editing by using a subset of the jED editor keys.
In particular, you can use the right and left arrow keys to move the cursor to any position in the current command string. The jsh is configured for editing in insert mode by default. This means that any characters you type will be inserted just before the current cursor position. Use the backspace key to delete the previous character and the <Delete> key to delete the character directly under the cursor. Try recalling a previous command and experiment with the editing keys.
The jsh can be placed into overwrite editing mode by pressing <Ctrl O>. In this mode, all typed characters will replace the character under the cursor.

Shown in the following table are all the editing commands:

Keystroke	Command
<Right>	Move the cursor right by one character
<Left>	Move the cursor left by one character
<Home> or <Ctrl A>	Move the cursor to the start of the command line
<End> or <Ctrl E>	Move the cursor to the end of the command line
<Down> or <Ctrl N>	Recall the next command in your history
<Insert> or <Ctrl O>	Toggle Overwrite/Insert mode, default is Insert
<Up> or <Ctrl P>	Recall the previous command in your command history
<Ctrl L>	List the command history maintained by the shell
<Ctrl K>	Delete from the cursor to the end of the command line
<Ctrl W>	Delete from the cursor to the end of the current word
<Tab>	Move to the start of the next word
<Backtab>	Move to the start of the previous word

jSH Emulation Modes

If you are already familiar with operating UNIX under other shells, jsh will allow you to work in the environment with which you are most comfortable. You can switch between the various emulation modes in jsh by using the function keys:

F1	jSHELL (jsh)
F2	Native Platform Shell. (CMD.exe, ksh, csh, etc)
F3	Mixed shell (msh)

NOTE: that some terminals may not support these function keys. If your terminal does not support the F1, F2 and F3 function keys, the emulation mode can be modified by the jsh internal command, jshelltype. The command syntax is as follows:

jshelltype shell
jshell can be one of the following:
jsh pre-processes Meta characters like the asterisk (*), as expected by legacy systems.
sh native system shell On UNIX depends of SHELL on NT/Win95 CMD.exe.
msh mixed shell. pre-processes Meta characters as a combination of jsh and sh.
msh Mapped Sequences
Options specified on the command line will have the leading bracket escaped. For example:

```
CT File1 Record1 (X becomes CT File1 Record1 \(X
```

Any asterisk used as part of a record specification is escaped. For example:

```
CT File1 * becomes CT File1 \*
```

Quotes used in a selection criteria specification are escaped

```
LIST File1 WITH A1 = “XYZ]” becomes LIST File1 WITH \*A1 = \”XYZ]\”
```

Other Meta characters are untouched so that pipes, etc. can be invoked:

jBASE Profiling

The following describes the new jBASE 4.1 mechanisms for controlling certain profiling tools:
This profiling mechanism allows versatility in where the output logs are stored and hot spot profiling in C and jBASIC code. The functionality can be applied dynamically to currently executing applications. It may be used to trace branch functions (INPUT, EXECUTE, CALL and RETURN instruction), to trace memory allocations (subsequently using jfatty to look for memory leaks), and general profiling activity.

Invoking Profiling

The basic way to invoke the functionality is through the JDIAG environment variable. Here follows a quick example as run from the Unix shell.

```
% export JDIAG=branch=on:output=filename.out
% MYPROGRAM
```

In this example, we have specified two commands delimited by a colon (There can be any number of commands delimited). In this example, 'branch=on' means to turn on the tracing of branch instructions and write the trace output to the file 'filename.out'.

It is also possible to dynamically run these commands against running programs. The same example as above but applied dynamically to port 23 would be:

```
% jprof 23 branch=on:output=filename.out
```

On most platforms the effect should be immediate but due to a bug in RedHat Linux and the limitation of Windows, the effect might take up to 60 seconds before it starts working.

You can obtain a short help screen as a quick reference reminder by the following:

```
 % JDIAG=help WHO
JDIAG=option{:option{:option ...}}
option can be one of ...
        profile={off|short|long|user|jcover}
        output={stdout|stderr|tmp|filename{,refresh_mins}
        memory={off|on|verify}
        branch={off|on}
        help
1 greg
```

Although the JDIAG environment variable name is case sensitive, the commands contained in its value are not e.g. profile=ON is the same as ProFILE=On
A list of all the available commands follow, below which is listed an example of dynamic usage to test for memory leaks in jBASE.

profile=off	Turns off the general profiling tools.
profile=short	Turns ON the short form of profiling.
profile=long	Turns ON the long form of profiling.
jprofile=user	Turns ON the user-profiling output
profile=jcover	Turns ON the form of profiling required for jcover
output=stdout	All the trace logs will be displayed to stdout
output=stderr	All the trace logs will be displayed to stderr
output=tmp{,nn}	The trace logs will be written to $JBASICRELEASEDIR/tmp/jprof_nn where nn is the process ID. You can optionally specify a cycle time in minutes. If the cycle time is specified, then 5 files are created with the suffixes _0 through _4 and the output logs will cycle through all 5 files in intervals of nn minutes. (At the time of writing this manual this

	feature is under development, future utilities will use this feature to display snapshots of the running system).
output=filename{,nn}	The trace logs will be written to the file 'filename' which can be any valid operating system file name, or even device name. Again, the optional cycle time can be used to create 5 files.
memory=off	Turns OFF all the traces of memory allocations
memory=on	Turns ON the trace of memory allocations.
memory=verify	Turns on the verification of memory allocations, looks for over-runs and under-run memory allocations, duplicate released spaces etc. This can only be executed by using the JDIAG environment variable and cannot be set dynamically.
branch=off	Turns OFF all the branch traces.
branch=on	Turns ON the trace of branch instructions such as INPUT , CALL, RETURN and EXECUTE/PERFORM
help	Displays the short help messages.

As an example, let us assume that a program is running and gradually consuming more and more memory. You want to investigate where the memory is being leaked.

Find the port number from where it is running

```
jsh --> WHERE
 Port   Device     Account      PID      Command
 0      6          greg         10941    MYPROG
*1      8          greg         10949    WHERE
```

Start dynamically tracing the memory allocations.

```
jsh->jprof 0 memory=on:output=MYPROG.output
Message sent successfully to port 0
```

The above command could take up to 60 seconds to invoke. Run MYPROG for 3-4 minutes to accumulate some statistics. The MYPROG.file output will increase in size as the trace begins.

Stop the trace

```
jsh->jprof 0 memory=off
Message sent successfully to port 0
```

Wait 60 seconds to take effect

Run the jfatty command against file created file

```
jsh->jfatty MYPROG.output
Remaining: 00005785: 100 bytes at 0x08115108 at
test9C.c,6(test9.b,6)
Remaining: 00005795: 100 bytes at 0x08115148 at
test9C.c,6(test9.b,6)
Remaining: 00005805: 100 bytes at 0x08115188 at
test9C.c,6(test9.b,6)
```

We can see where the memory is leaking.
This is an example of hot-spot user profiling, the idea of which is that you place some function calls in either your jBASIC code or your C code and let the profiling tools measure the CPU used between these calls. You call function JBASEUserProfile(n) where **n** is an integer between 0 and 10. A value of 0 means profiling ticks are discarded. A value of 1 through 10 means each CPU clock tick is stored in a 10 element array so we can ascertain how many CPU ticks arrived for that period. Consider the following example jBASIC code ...

```
INCLUDE JBASIC.h
OPEN "FB1" TO DSCB ELSE DEBUG
S1 = SYSTEM(12)
FOR I = 1 TO 100000
    JBASEUserProfile(1)
    READU rec FROM DSCB,"x" ELSE NULL
    JBASEUserProfile(2)
    RELEASE
    JBASEUserProfile(0)
NEXT I
S2 = SYSTEM(12)
PRINT "ELAPSED = ":(S2-S1)/1000
```

Compile normally with BASIC and CATALOG, or use the jcompile command.
When it is run set the JDIAG variable as follows (assuming Unix):

```
JDIAG=profile=user PROGRAMNAME
ELAPSED = 3.66
PROFILE:User Profile 1 , 30 ticks 83.33 %
PROFILE:User Profile 2 , 5 ticks 13.89 %
```

The output shows you that 83.33% of the CPU time is spent between the JBASEUserProfile(1) and JBASEUserProfile(2) function calls, and that 18.39% of the CPU time is spent between the JBASEUserProfile(2) and the JBASEUserProfile(0) function calls, thus proving that the RELEASE statement is consuming large amounts of CPU.

The function calls can equally be used in C code.

To ensure more accuracy, the program should be run for at least a few seconds in order to accumulate as many CPU ticks as possible. Anything less than a second will increase the margin of error

In particular, developers might like to notice the hot-spot user defined profiling which allows C code to be modified and easy verification of where CPU time is being spent within a function.

Simply add checkpoints into your jBASIC or C code. At the end of the test program, it tells you how many ticks and what percentage of CPU was spent between these checkpoints something very valuable when trying to determine, for example, " There is a big JediOpen function that takes a huge amount of CPU, but where inside the JediOpen() does it take the most?" Using the profiler, in this example, you can tell it is not JediOpen, but rather JediClose.

jBASE Tools

CHAR

The char utility displays a character conversion table, providing decimal, hexadecimal and octal representations of the ANSI character

Encrypt

The Encrypt utility provides simple encryption/decryption of files using a user specified key.

```
Encrypt –k key File > EncryptedFile
Encrypt –d –k key EncryptedFile > File
```

ERRMSG

The errmsg tool displays the platform related error message text when used with an error message number.

```
errmsg 13
```

HAD

The had utility displays any file in a hexadecimal format. A hexadecimal offset can be specified from which to start the display.

```
had –offset
```

JCOMP

The jcomp utility provides a mechanism to compare two Hash files or directories

```
jcomp -Options FileName (Options
```

Option	Description
-L or (L	Restrict error display to single line.
-N or (N	No paging.
-P or (P	Output to printer.
-S or (S	Suppress matching records.
-T or (T	Trim source before compare.
-V or (V	Verbose display of non-matching records.

jCOVER

The jcover programs provide a mechanism to generate statistical information on the coverage of an application during a test run. It can be used to calculate the percentage of code that was executed, percentage not executed, what sources were not used in files and so on.
The usage of jcover depends upon much of the existing profiling tools. Therefore this document should be in conjunction with existing documentation available for profiling tools. The jcover utility can be used to great advantage with the jkeyauto utility. Using both of them together it is possible to build up highly automated scripts that will automatically test new releases of software and ensure that a known percentage of your entire application gets executed as well as ensuring that all paths through your application are executed and tested.
Using jcover involves these 3 key steps:

1. Run the application and record the information.
2. Collating the recorded information.
3. Reporting the jcover information.

Recording the information

This phase involves running your application in a certain manner and allowing jBASE to record the statistics about the lines of code that were executed whilst the application was run. There are two ways of telling jBASE to record this information

Use the -JC option to the program. For example:

```
% MYAPPLICATION -JC COMMANDARG ETC ETC
```

In the above example, when the application 'MYAPPLICATION' terminates a file called 'jprof' will be written to the current working directory which contains all the profiling information for this application.

Set the JBCPROFILE environment variable to 3.

For example:

```
% JBCPROFILE=3
% export JBCPROFILE
% MYAPPLICATION COMMANDARG ETC ETC
```

In this second invocation jBASE will create a file in the format jprof_mmmmm_nnn, one file for each program that is executed where mmmm is the process id and nnn is the PERFORM/EXECUTE level. In this way you can generate multiple copies of profiling information which will automatically include all executed procs and programs. Each file will be stored in the current working directory as of when the application was loaded.
Remember to unset the JBCPROFILE environment variable when you are finished otherwise you will find a very large number of the profiling files cluttering up your directory. It might be useful to run your application through a jkeyauto script. In this manner you can accurately repeat a test case through your application and repeatedly use the jcover tools to ensure as much as possible percentage of your application is executed.
At the end of the recording phase you will have one or more profiling files. These profiling files can then be used in the second stage of jcover profiling. Remember that as UNIX files they can be easily moved and renamed. For example you could run a number of test cases using the -JC option and each time rename the generated 'jprof' file to something else, such as jprof_a, jprof_b and so on. Then in the next phase you can supply multiply file name to be manipulated.

Collating the information

Once you have recorded one or more profiling files as shown in the previous section, you are ready to collate all the information. This is the function of the jcover program. The jcover program is run like this:

```
jcover {-fFilename {-fFileName ...}} {-oOutputFile} {-ddelim} {-e}
{-h|?} {-u} {-x} {jprof {jprof ...}}
```

-ddelim	Change the delimiter in output records from '*'
-e	Write out verbose details of all lines of code executed
-fName	Name of file to extract source from
-h | -?	Help screen output
-oOutputFile	Output file name
-u	Write out details of unused source code in the supplied input files
-x	Write out verbose details of all lines of code NOT executed
prof	Profile name

By default it will take the input Unix file ‘jprof’, extract some statistical information about it, and write it back out to a newly-created file called jcover_nnn where nnn is your port number. It will also create suitable dictionary items.
The default information stored will be a summary of the percentages of all source items executed.
You can use the -e, -u and -x options to create additional statistical information.
The -oOutputFile option allows you to specify an alternative output file to jcover_nnn (where nnn is your port number). If this option is not used, we default to jcover_nnn and create the file if necessary.
Whatever file name you choose, both the DICT and the DATA section will be cleared and suitable DICT items written away for later usage.
Once this phase is complete, you will have an output file ready for use with the final phase.

Reporting the information

This third phase requires the use of the file created in the second phase. For convenience we will call it jcover_23, but in fact it can be any file name used as the output of the jcover program.
There are basically 3 ways of using the information created in file jcover_23
Using jQL with the default DICT macros
To show a summary of percentages of coverages use:

```
LIST jcover_23 STATS
```

To show a summary of general information, such as times, dates, files used and so on. Use:

```
LIST jcover_23 GENINFO
```

If the -e option was used; you can display a list of all the lines of source code executed with:

```
LIST jcover_23 EXEC
```

If the -x option was used, you can display a list of all the lines of source code not executed with:

```
LIST jcover_23 NOTEXEC
```

If the -u option was used; you can display a list of all source items that didn’t have any code executed during the application execution

```
LIST jcover_23 NOTUSED
```

Using jQL with your own command line
This involves simply using a jQL statement with something other than the supplied macros. For example you could:

```
LIST jcover_23 WITH S.ITEMID EQ “MAR]” S.ITEMID
```

Using a jBASE BASIC program
Refer to the layout of the output file and the dictionaries to write your own jBC program to further report on the information available.

Caveats

At present, jcover is available on UNIX platforms only.
When you are running profiling for jcover your application will run at 2-20 times slower, so this type of profiling isn't suitable for performance measurement at all. If you use this against batch jobs, you may find this performance degradation is too much. You will additionally use around 0.5 to 5 Mb extra memory per application while running this jcover profiling.
The profiling can only detect whole lines of code executed, or not executed. In the following line of code:

READ record FROM filevar, itemid ELSE record = "xxx" ; GOSUB subname
The jcover profiling will only record the fact that the READ statement was executed -- it cannot be used to determine if the code following the ELSE was executed or not. If you want to test for this, then the application will need to be changed so that the ELSE code is on a separate line.

Layout of the JCOVER output file

The records written to the jcover output file have different formats depending upon what they are showing. The formats are identified by a prefix on the record key. Note that in the following examples a * character is used to delimit certain fields. This is the default delimiter character. Remember if the -d option was used to jcover, then this delimiter will be different.
E*itemid*filename*lineno
These items show which lines of code have been executed. They are only generated if the -e option is used on jcover. There will be one item for each different line of code executed during the application execution.

Item Id The 4 fields show the record type (E), the item id, the source file name and the line number that were executed.

<1> The actual line of source code taken from the source 'itemid' in file 'filename'

F*itemid*filename

Used internally to keep track of what items in what files have been accessed as part of the jcover execution

Item Id The 3 fields show the record type (F) , the item id and the source file name that were executed.

G*descr

These are general information items.

Item Id The 2 fields show the record type (G) and then a unique identifier, not really of any use.

<1> Textual description of the general information

<2> The actual general information.

<3> the order in which the fields are to be displayed using 'LIST filename GENINFO'

S*itemid

These items show what item has been executed and are used to build up a list of statistical analysis of the executed item.

Item Id The 2 fields show the record type (S) and the item id that were executed.

<1> Item ID of the source executed

<2> File name containing the executed source

<3> Number of lines of source. This attribute will be the sum of attributes <4> and <5>.

<4> Total number of comment lines in the source code.

<5> Number of lines of code in the source that are executable. Blank lines and comment lines are not included in this count.

<6> Number of lines of code executed by the application.

<7> Number of lines of code executed by the application as a percentage of the total number of executable lines.

<8> Number of lines of code not executed by the application

<9> Number of lines of code not executed by the application as a percentage of the total number of executable lines.

<10>-<20> Reserved for future use

<21> Details of line 1 of the source code; a character 'C' shows line 1 is a comment line. A numeric value shows line 1 is executable code and has been executed this many times. A blank shows line 1 is executable code, but was not executed during the application execution.

<22> Ditto as <21> but for line 2 of the source code.

<23> Ditto as <21> but for line 3 of the source code.

etc. for all lines in the source code.

U*itemid*filename

For every file name specified with the -ffilename parameter of jcover, there will be one of these items for every item in every file that was not used during the application execution. These items are only generated if the -u option is used on jcover. Item id's that show a non-source item will not be included and these are items that are appended with .o , .a , .so , .sl , .obj , .d or .d or items that begin with ! , $ or *.

Item Id The 3 fields show the record type (U) , the item id and the source file name that were not executed in any way whatsoever.

X*itemid*filename*lineno

These items show which lines of code have not been executed. They are only generated if the -x option is used on jcover. There will be one item for each different line of code not executed during the application execution. Only the source files that have one or more line of code executed will be included here. Any source items not executed at all will not have a X* item but will be included in the U* items (assuming the -u option was used)

Item Id The 4 fields show the record type (X) , the item id, the source file name and the line number that were not executed, but the source did have one or more lines of code executed.
<1> The actual line of source code taken from the source 'itemid' in file 'filename'

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
```

JFIND

The Windows platform does not provide a command line search program like the Unix find command. This is a limited form of the Unix find command for the Windows platform.

```
jfind C:\MYDIR -name ".*FRED.*" -print
```

JGREP

Use the jgrep command to search for strings in jBASE files or directories.

```
jgrep -Options SearchString FileName (Options
```

Option	Description
-c or (C)	make search case in-sensitive
-i or (I)	interactively ask for one or more SearchString"s
-k or (K)	search in record KEY only for string
-l or (L)	simply list the record keys they were found in
-n or (N)	do not wait for keyboard input between pages
-p or (P)	send output to printer
-s or (S)	sub-directory searches
-t or (T)	trims redundant spaces from search patterns
-r or (R)	raw display exclusive of dollar items

JRM JMV JDIR

Many commands are built-in to the cmd.exe shell and so cannot be executed like a like other executables. The following commands have been provided to use in either jsh or cmd shell.

Command	Description
jrm {-r}	remove file or directory, recursively
jmv oldfile newfile	move oldfile to newfile
jdir	list directory

JMSGBOX

Use the jmsgbox utility to display a message box on Windows platforms. Use it with a console process and the buttons are not configurable.

COMMAND SYNTAX

```
jmsgbox text
```

EXAMPLE:

```
PERFORM "jmsgbox Test Box" SETTING Result
IF Result<1,1>=0 THEN CRT "Ok was clicked"
IF Result<1,1>=1 THEN
```

JSHOW

The jshow command can be used to find jBASE files or programs.

COMMAND SYNTAX

```
jshow -Options Name {Name ...}
```

Where Options can be:

Option	Explanation
-a	display subroutine names in dll/shared object
-c	display compile time and source file
-f	file name only search
-h	display this help screen
-p	program name only search
-s	subroutine name only search
-v	verbose mode

SYNTAX ELEMENTS

Name Name of file, subroutine, program or dll/shared object

jprof

The jBASE profiling tools jprof enables developers to analyze applications to determine potential bottlenecks or trouble spots within the application code.
By default, no profiling is done in the program. Programs do not have to be compiled in any special manner to enable profiling for that program. All that is required is that the programs were not compiled with optimization, as this discards the debug information which is required for profiling.
The mechanism works by receiving a signal at every clock tick and keeping note of where the program was when the signal arrived. Thus, for the profiling to be accurate, the application must be run for a relatively long time. It will not show particularly good results if, for example, a program executes in less than a second. Several minutes or longer is preferred.
Currently profiling is available only with jBASE on Unix (including Linux) platforms.

Enabling profiling

Profiling can either be enabled using the -JP option, which only profiles the root process, or via the JBCPROFILE environment variable, which profiles the root process as well as all EXECUTEd processes.

```
MAINPROG -JP
```

This command generates a profiling file called jprof_n in the current directory where n is the port number. Only MAINPROG and any CALLed subroutines are profiled, any EXECUTEd programs will not be profiled. However, if the CPU time spent executing the actual EXECUTE statement is significant, that line will be included in the profiling statistics. Profiling terminates when the application stops or chains to another program.

```
JBCPROFILE=1
MAINPROG
```

This command generates a different profiling file for each process executed in the form jprof_pid_n, where pid is the process id and n is an incrementing number starting at 0.
The profiling file generated will only contain information about user CPU time. The time spent in system calls, file I/O, and lines which do not accumulate more than a clock tick are not included in the profiling statistics.

Profile Reporting

Use the **jprof** command to provide profile analysis of the jprof files generated by a program executed with the -JP option.

Called as :

```
jprof -kfilename {jprof{_nnn}}
jprof -a {jprof{_nnn}}
jprof -o {-v} {jprof{_nnn}}
jprof -s {jprof{_nnn}}
jprof {-n{-u}} {-i} {-fFilename}} {jprof{_nnn}}
```

-a Display all ancillary information.

-fName Name of file to extract source from.

-I Sort by increasing ticks, rather than decreasing tick.

-kKeyFile Name of file to store keyboard INPUT, used by jkeyauto.

-n Subtotaled and sorted by source name

-o Display shared object usage

-s Display list of subroutines called

-u Sorted by CPU utilization

-v Verbose mode

```
jprof{_nnn} Profile name (default "jprof")
```

EXAMPLE OF PROFILING

Imagine the source "test1.b" below has been edited into file BP, where BP is a directory. Notice the INCLUDE of another source file "test2.b".

```
OPEN "fb1" TO DSCB ELSE STOP 201,"fb1"
PRINT "Phase 1 -- start"
S1 = SYSTEM(9)
FOR Id = 1 TO 100
   Rec = ''
   FOR I = 1 TO 100
      Line = ''
      FOR J = 1 TO 20
         Line := CHAR(SEQ("A")+RND(26))
      NEXT J
      Rec = Line
   NEXT I
   WRITE Rec ON DSCB,Id
   NEXT Id
PRINT "Phase 1 -- end, CPU = ":SYSTEM(9)-S1
INCLUDE test2.b
PRINT C1:" records in file fb1"
PRINT "End"
```

The program can be created normally with the following command:

```
BASIC BP test1.b
CATALOG BP test1.b
```

By default, when the program is run, no profiling will take place.
Run the program with the -JP switch to create a file "jprof":

```
test1 -JP
```

We can now examine the profile file with the "jprof" command, using the -f option to generate optional source code listings from the file BP.

```
jprof -f BP jprof
```

PROFILE REPORT

Profile of program test1 from profile jprof Page 1

```
Source     Line  Ticks       %     Source
test2.b    8     166         32.93 READ Rec FROM DSCB,Key EL
test1.b    9     160         31.74 Line := CHAR(SEQ("A")+RND(
test1.b    11    128         25.39 Rec = Line
test2.b    7     28          5.55  WHILE READNEXT Key DO
test1.b    10    9           1.78  NEXT J
test2.b    9     5           0.99  C1++
test1.b    13    3           0.59  WRITE Rec ON DSCB,Id
test2.b    5     2           0.39  SELECT DSCB
test1.b    7     2           0.39  Line = "
test2.b    10    1           0.19  REPEAT
```

The -i option would sort the output with incrementing Ticks counts. The -n option would additionally sort it by file name, so the "test1.b" entries will be displayed separately to the "test2.b" entries.
Use the jfb command 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`

The Windows platform does not provide a command line search program like the UNIX find command. This is a limited form of the UNIX find command for the Windows platform.

`jfind C:\MYDIR -name ".*FRED.*" -print`

JSHOW

Use the jshow command to find jBASE files or programs.

COMMAND SYNTAX

```
jshow -Options Name {Name ...}
```

Where Options can be:

-a display subroutine names in dll/shared object

-c display compile time and source file

-f file name only search

-h display this help screen

-p program name only search

-s subroutine name only search

-v verbose mode

SYNTAX ELEMENTS

Name Name of file, subroutine, program or dll/shared object

JSTART

The jstart utility was originally provided to circumvent a problem with the Win95 CreateProcess API. The API fails to set the standard input handle to the new console if the standard input handle of the parent process is not a console device, i.e. a DATA statement has been used. However, jstart can be used to start up a process in the same window, in a new window or in the background. Not available on UNIX platforms.

`jstart -Options Command`

-v verbose

-w wait for process

-m run with window minimized

-b run as a background job, i.e. no console

-i inherit current window

-t'Title' specify the title

-p<class> priority class; 0 - Normal; 1 - Idle; 2 - High

EXAMPLE

```
EXECUTE "jstart jsh -s jsh -"
```

This will start up the jSHELL in a new window, with a different port number. This works for both Win9x as well as NT.

JTIC

```
jtic -Options DescriptionFile
```

Where Options can be:

-x the Description File contains extended capabilities

-v verbose mode

By default, the program jtic will take a source description file called terminfo.src and assume it contains standard terminfo names. The output will be to a file called /usr/lib/terminfo/x/xyz, where x/xyz depends upon the terminal name as contained in the description file.
You can use any alternative description file to terminfo.src by specifying the description file name on the command line. You can specify an alternative output directory to /usr/lib/terminfo by amending the TERMINFO environment variable. However, when you run a BASIC program that accesses these definitions in an alternative directory, the TERMINFO variable needs to match that when the definition was compiled; by default the jtic program assumes the description file contains standard terminfo definitions: e.g.

```
cuu1=\E[1A, cols#80,
```

If you want to create a binary with the extended capabilities, use the -x option. Remember when running jtic you will probably require root privileges to write to the /usr/lib/terminfo directory.
jtic Description File

To run jtic you need a description file to describe the capabilities. This file is very similar to that required by the UNIX tic command.
Comment lines start with # as the first character of each line. Blank lines are also counted as comment lines.

Terminal names - This tells jtic the names of the terminal definition. It is a list of names delimited by | characters. The final field is treated as a comment. Following the terminal names will be the binary definitions, integer values and string values for those terminals. When another set of terminal names are defined, the current definitions are written to file and the current definitions nulled. E.g.
vt100|vt100am|prism-ans|Simple vt100 support
Binary definition - These are just the name of the definition. E.g, on
Integer value - The name of the definition followed by # then the value. E.g. cols#132
String value - The name of the definition followed by = followed by the string value. jtic supports all the functionality of tic, such as defining characters 1 to 26 using ^A through ^Z, specials such as \E for escape, \s for space, \t for tab. It also supports the if/else/endif structure and logical/arithmetic operations supported by tic. E.g. if_prtr_letter=\E[02l,

use=name. Resets the definition to that of a previously defined name in the definition; you can use this to create a terminfo definition for one or more names, and then base subsequent definitions on that. An example of using this can be found in the source $JBCRELEASEDIR/src/prism. E.g.
use=ansi
The comments and terminal names must all start at column 1. The binaries, integers, strings and use=name must all have a leading tab character. There can be more than one binary, integer or string on a line, and each definition should be delimited by a comma. When invoked with the -x option a terminfo binary file with the suffix _jbase is generated for the extended capabilities.

Extended Definitions

Access	Long name Description
@(-53)	if_slave_only Send data to slave printer only. No VDT display
@(-54)	if_crt_type Returns VDT terminal type
@(-55)	if_crt_graphics Returns a 1 if Regis graphics available
@(-59)	if_crt_cuprot Clear all unprotected fields
@(-27)	if_crt_132 Switch VDT to 132 column mode
@(-28)	if_crt_80 Switch VDT to 80 column mode
@(-29)	if_crt_dwide Display double wide characters
@(-30)	if_crt_swide Display single wide characters
@(-32)	if_crt_sron Set-up scrolling region. (Enter ? for help)
@(-33)	if_crt_sroff Reset scrolling region to normal
@(-47)	if_crt_udhdw Top half of double height line
@(-48)	if_crt_bdhdw Bottom half of double height line
@(-60)	if_prtr_executive Change paper size to executive
@(-61)	if_prtr_a4 Change paper size to A4
@(-62)	if_prtr_monarch Envelope type "Monarch"
@(-63)	if_prtr_comm10 Envelope type "Commercial 10"
@(-64)	if_prtr_interntldl Envelope type "International DL"
@(-65)	if_prtr_reset Reset printer defaults
@(-66)	if_prtr_envfeed Envelope feeder
@(-70)	if_prtr_letter Change paper size to letter
@(-71)	if_prtr_legal Change paper size to legal
@(-72)	if_prtr_chgcpy Change number of copies to print on Laser
@(-73)	if_prtr_cpwo1 Compressed print for Service WO form 1
@(-74)	if_prtr_spcol Start printing at specified column

@(-75) if_prtr_sprow Start printing at specified row

@(-76) if_prtr_utray Use Upper Tray

@(-77) if_prtr_ltray Use Lower Tray

@(-78) if_prtr_portrt Portrait orientation

@(-79) if_prtr_land Landscape orientation

@(-80) if_prtr_simplx Simplex binding

@(-81) if_prtr_duplxl Duplex, long edge binding

@(-82) if_prtr_duplxs Duplex, short edge binding

@(-83) if_prtr_macro Call MACRO

@(-84) if_prtr_setdef Set default (Font size, HMI, VMI)

@(-85) if_prtr_lpi2 2 lines per inch

@(-86) if_prtr_lpi3 3 lines per inch

@(-87) if_prtr_lpi4 4 lines per inch

@(-88) if_prtr_lpi6 6 lines per inch

@(-89) if_prtr_lpi8 8 lines per inch

@(-90) if_prtr_lpi12 12 lines per inch

@(-91) if_prtr_dwide Double wide mode

@(-92) if_prtr_swide Single wide mode

@(-93) if_prtr_96 96 column mode

@(-94) if_prtr_pld 1/2 line down

@(-95) if_prtr_plu 1/2 line up

@(-96) if_prtr_suon Superscript mode

@(-97) if_prtr_sbon Subscript mode

@(-98) if_prtr_ssoff Superscript and subscript off

@(-99) if_prtr_40 Double wide for 80 column mode (5 pitch)

@(-100) if_prtr_48 Double wide for 96 column mode (6 pitch)

@(-101) if_prtr_ff Top of form

@(-102) if_prtr_80 80 column mode (10 pitch)

@(-103) if_prtr_132 132 column mode (16 pitch)

@(-104) if_prtr_bold Bold

@(-105) if_prtr_ul Underline

@(-106) if_prtr_norm Turn off bold and underline

@(-107) if_prtr_hmi Set horizontal motion index to U-1

@(-108) if_prtr_vmi Set vertical motion index to U-1

@(-109) if_prtr_pson Proportional spacing on

@(-110) if_prtr_psoff Proportional spacing off

@(-111) if_prtr_1key Linefeed and backspace

@(-112) if_prtr_2key Linefeed

@(-113) if_prtr_3key Linefeed and space

@(-114) if_prtr_4key Backspace

@(-115) if_prtr_6key Space

@(-116) if_prtr_7key Negative linefeed and backspace

@(-117) if_prtr_8key Negative linefeed

@(-118) if_prtr_9key Negative linefeed and space

@(-119) if_prtr_cvd Coarse vertical distance (1 line = 1 inch)

@(-120) if_prtr_mvd Medium vertical distance (1 line = 1/6 inch)

@(-121) if_prtr_type Returns slave/printer type

@(-122) if_prtr_fvd Fine vertical distance (1 line = 1/48 inch)

@(-123) if_prtr_chd Coarse horizontal distance (1 space = 1 inch)

@(-124) if_prtr_mhd Medium horizontal distance (1 space = 1/12 inch)

@(-125) if_prtr_fhd Fine horizontal distance (1 space = 1/120 inch)

@(-126) if_prtr_status Retrieve slave/printer device status

@(-220) IF_PRTR_DUL_ON

@(-221) IF_PRTR_DUL_OFF

@(-223) IF_PRTR_STRIKE_ON

@(-224) IF_PRTR_STRIKE_OFF

@(-225) IF_PRTR_HARD_HYPHEN

@(-226) IF_PRTR_HARD_BLANK

@(-230) IF_PRTR_FONT_0

@(-231) IF_PRTR_FONT_1

@(-232) IF_PRTR_FONT_2

@(-233) IF_PRTR_FONT_3

@(-234) IF_PRTR_FONT_4

@(-235) IF_PRTR_FONT_5

@(-236) IF_PRTR_FONT_6

@(-237) IF_PRTR_FONT_7

@(-238) IF_PRTR_FONT_8

@(-239) IF_PRTR_FONT_9

jBASE Extended terminfo definitions generation

To generate a terminfo extension file, using the jtic -x comamnd, setting any of the following terminfo strings references in lowercase. Ensure that the prt_video emulation flag is configured to true, for the required emulation.

e.g.

Generate jBASE extension tic file

```
ED . Mytic
TOP
.
001 # My jBASE extension file for vt220
002 vt220|VT220
003 if_crt_132=My132String,
004 if_crt_80=My80String,
.FI
```

Generate jBASE terminfo extension file.

```
jtic -x Mytic
```

This generates a jBASE extension entry for vt220 in the terminfo database. e.g.

/usr/lib/terminfo/v/vt220_ext_jbase

Test terminfo produces correct result

```
ED . TESTMYTIC.b
TOP
.
001 CRT "Output my 80 column terminfo command ":@(-28)
002 CRT "Output my 132 column terminfo command ":@(-27)
.FI
export TERM=vt220
export JBCEMULATE=ROS
jbc -Jo TESTMYTIC.b
TESTMYTIC > MyFile
```

KEYS

The keys utility displays the keyboard input. This program does not exist on the NT/Win95 platform but could be coded as follows:

```
NOEXIT = 1
LOOP
ECHO OFF
CRT "Hit a key :":
IN ch FOR 100 ELSE NOEXIT = 0
ECHO ON
WHILE NOEXIT DO
IF ch <> 127 THEN
    CRT "." : CHAR(9): "0x0" : ch "MCDX"
END ELSE
    CRT CHAR(ch) : CHAR(9): "0x0" :ch "MCDX"
END
REPEAT
```

LIBUTILS

The libutils shared library is provided to enable users to code terminal independent code to handle keyboard input.

EXAMPLE

```
* Include key definitions
INCLUDE jCmdKeys.h
* Initialize command key strings
CALL CommandInit
TimeOut = 150 ;* Set timeout value deciseconds
ECHO OFF
LOOP
* Get Next Command Value
CALL CommandNext(RetNo, RetString, TimeOut)
```

```
* RetNo should match numbers in include/header file
BEGIN CASE
CASE RetNo = cmd_cursor_up
   CRT "CURSOR UP"
CASE RetNo = cmd_cursor_down
   CRT "CURSOR DOWN"
CASE RetNo = cmd_cursor_left
   CRT "CURSOR LEFT"
CASE RetNo = cmd_cursor_right
   CRT "CURSOR RIGHT"
CASE RetNo = cmd_alpha_numeric
   CRT "ALPHANUMERIC"
CASE RetNo = cmd_timeout
   CRT "TIMEOUT"
   BREAK
END CASE
* Output the actual string returned
CRT "RetString :":OCONV(RetString,"MCP.")
REPEAT
```

@USERSTATS

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

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

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

```
EQUATE USER_PROC_PORT_NUMBER TO 1;* The port number
EQUATE USER_PROC_NUM_PROGRAMS TO 2;* Number of programs running in
this port
EQUATE USER_PROC_START_TIME TO 3;* Time user started in UTC format
EQUATE  USER_PROC_PID TO 4  ;* Process ID
EQUATE USER_PROC_ACCOUNT TO 5;* Name of the account
EQUATE  USER_PROC_USER TO 6 ;* Name of the user
EQUATE USER_PROC_TERMINAL_JBASE TO 7;* Name of terminal according to
jBASE
EQUATE USER_PROC_TERMINAL_OS TO 8;* Name of terminal as seen by OS
EQUATE USER_PROC_DATABASE TO 9;* Name of database connected to
EQUATE USER_PROC_TTY TO 10;* Name of TTY device
EQUATE USER_PROC_LANGUAGE TO 11;* Language
EQUATE USER_PROC_LISTENING_TIME TO 12;* Time in UTC the listening
thread last worked
EQUATE USER_PROC_MEM_FREE TO 13;* Amount of memory in heap space
free chain
EQUATE USER_PROC_MEM_USED TO 14;* Amount of heap space memory in use
EQUATE USER_PROC_THREAD_TYPE_INT TO 15;* Thread type expressed as an
integer
EQUATE USER_PROC_THREAD_TYPE_TXT TO 16;* Thread type expressed as a
text string
EQUATE  USER_PROC_LICENSE TO 17;* License counters
EQUATE USER_PROC_STATS_OPEN TO 18;* Number of OPEN's performed.
```

```
EQUATE USER_PROC_STATS_READ TO 19;* Number of READ's performed.
EQUATE USER_PROC_STATS_WRITE TO 20;* Number of WRITE's performed.
EQUATE USER_PROC_STATS_DELETE TO 21;* Number of DELETE's performed.
EQUATE USER_PROC_STATS_CLEARFILE TO 22;* Number of CLEARFILE's
performed.
EQUATE USER_PROC_STATS_PERFORM TO 23;* Number of PERFORM's /
EXECUTE's performed.
EQUATE USER_PROC_STATS_INPUT TO 24;* Number of INPUT's performed.
EQUATE  USER_PROC_UNUSED_1 TO 25;* Unused
EQUATE USER_PROC_OPEN_FILES_VIRTUAL TO 26           ;* Number of
files application thinks open
EQUATE USER_PROC_OPEN_FILES_REAL TO 27 ;* Number of files really
open by OS
EQUATE USER_PROC_USER_ROOT TO 28;* Application data set by
@USER.ROOT
EQUATE USER_PROC_PROCESS_TXT TO 29;* Text string to identify process
EQUATE USER_PROC_PROGRAM TO 41;* Program name and command line
arguments
EQUATE USER_PROC_LINE_NUMBER TO 42;* Line number currently being
executed.
EQUATE USER_PROC_SOURCE_NAME TO 43;* Name of source currently being
executed.
EQUATE  USER_PROC_UNUSED_2 TO 44;* Unused
EQUATE  USER_PROC_UNUSED_3 TO 45;* Unused
EQUATE  USER_PROC_STATUS_TXT TO 46;* Status of program as a readable
text
EQUATE  USER_PROC_STATUS_INT TO 47;* Status of program as an integer
EQUATE  USER_PROC_CPU_USR TO 48;* User CPU time
EQUATE  USER_PROC_CPU_SYS TO 49;* System CPU time
EQUATE  USER_PROC_CPU_USR_CHILD TO 50;* User CPU time used by child
processes
EQUATE  USER_PROC_CPU_SYS_CHILD TO 51;* System CPU time used by
child processes
EQUATE  USER_PROC_USER_THREAD TO 52;* Application data set by
@USER.THREAD
```

jBASE Independent Metrics Integration (JIMI)

There are two sides to JIMI:

1. Event Driven Metrics (EDM) which are produced in real time
2. Source Quality Metrics (SQM which are produced when programs are compiled.

Event Driven Metrics (EDM)

In jBASE release 4.1 onwards there is new functionality that allows you to gather information about application events in a non-real time manner. These statistics include information such as execution path length, database I/O operations, execution path branches (CALL and PERFORM) and so on. There are three basic ways of enabling EDM.

3. Use the API to enable this functionality on the same port or another user's port.
4. From the command line using the 'jprof' command you can enable this functionality for your own port or another user's port.
5. Using the JDIAG environment variable

The use of EDM means jBASE will generate a text file describing all sorts of information. However no attempt is made to perform any analysis of this output, it is up to the user to decide what to do with the information.

EDM calling API

Two new functions have been created to turn on and turn off the Event Driven Metrics. These functions are called JIMITraceOn and JIMITraceOff and are prototyped in the jBASE supplied file jimiB.h.

JIMITraceOn(PortNumber , FileName , Options)

This function calls turns EDM on for a particular port.

PortNumber: Enter the port number you wish to turn on or -1 to denote your own port.

FileName: The name of a file where the details will be written to. On UNIX systems it can also be the name of a tty device (e.g. "/dev/pts/0") so you can monitor on a terminal the events occurring in an application. Also "stdout" and "stderr" are valid.

Options: Define here a list of options separated by commas which describe what details to gather. By default i.e. "" , just short details are gathered. The options are…

DATABASE All OPEN's and database I/O (such as READ , WRITE) are traced

OPEN All OPEN's are traced

FILEIO All database I/O (such as READ, WRITE) but not including OPEN are traced

BRANCH All CALL, RETURN and PERFORM/EXECUTE statements are traced

CALL All CALL and RETURN statements are traced

EXECUTE All PERFORM/EXECUTE statements are traced

TRANSACTION Details of committed transactions including before and after images are traced

ALL All the above options

VERBOSE By default summaries are displayed at the end of the session. Using verbose gives a detailed output of each option as it happens.

JIMITraceOff(PortNumber)

This function call turns EDM off for a particular port.

PortNumber: Enter the port number you wish to turn off or -1 to denote your own port.

Once turned off you can read in this information in a number of ways. You can use jBASE editors like jed or ED, use UNIX editors like vi or you can read the file programmatically using OPEN and READ. Because the file might be quite substantial perhaps OPENSEQ and READSEQ would be more appropriate.

The return value from these functions are 0 for successful or non-zero for an error. You need to include jimiB.h in your application to get the relevant DEFC statements to support calling this function.

Once EDM is initiated you get a text oriented output sent to the file name specified

In a multi-thread environment, only the event details for that port are recorded even if multiple ports are running in different threads inside the same process.

EDM from the command line

You can use the jprof command to enable or disable this EDM functionality like this ..

jsh → jprof –j ON nnn filename options

Or

jsh -> jprof –j OFF nnn filename options

Where

nnn is the port number to turn on or off. You can specify % which means the same port number you are running jprof from.

filename is the name of the output file as described above in the “FileName” parameter to the JIMITraceOn function.

options is one or more of the strings as described above in the “Options” parameter to the JIMITraceOn function

EDM using the JDIAG environment variable

The JDIAG environment variable encompasses a large number of options and the EDM has been added to this. To find a short summary of all the JDIAG options use the command below from a UNIX shell.

```
machinename-~src/jbase4/4.1.3/jbase: JDIAG=HELP WHO
JDIAG=option{:option{:option ...}}

option can be one of ...
        profile={off|short|long|user|jcover|all}
        filename={stdout|stderr|tmp|pathname{,refresh_mins}
                 (%p in pathname is substituted with process id)
                 (%t in pathname is substituted with time stamp)
        memory={off|on|verify}
        branch={off|on|verbose}

jimi={database|open|fileio|branch|call|perform|transaction|all|verb
ose}
        trace=env_name{,env_name..}
        help
3 greg
machinename-~src/jbase4/4.1.3/jbase:
```

So to trace all database and transaction information into a file called ‘trace_nnn_out’ where ‘nnn’ is the process id (hence you get a new file for each process) you would do this example below, which sets JDIAG, runs a command (WHO) and then examines the resultant output file.

```
machinename-~src/jbase4/4.1.3/jbase: export
JDIAG=jimi=database,transaction,verbose:filename=trace_%p_out
machinename-~src/jbase4/4.1.3/jbase: ls -l trace*
ls: trace*: No such file or directory
Machinename-~src/jbase4/4.1.3/jbase: WHO
3 trace
Machinename-~src/jbase4/4.1.3/jbase: ls -l trace*
-rw-r--r--    1 trace     trace         1045 Mar  2 18:04
trace_7338_out
Machinename-~src/jbase4/4.1.3/jbase: cat greg_7338_out
TIMESTAMP: time 1078250684
OPEN:   FILENAME=/home/greg//4.0_rels/jbc4.1.4.3/tmp/jBASEWORK
```

```
TYPE=HASH4      STACK=1,0       SUCCESS=0SOURCE=jmainfunction.b,0
PATHLENGTH=121  TIME=2535235.970

OPEN:   FILENAME=DICT POINTER-FILE      STACK=1,0       SUCCESS=2
SOURCE=jmainfunction.b,0        PATHLENGTH=125
TIME=2535235.980

OPEN:   FILENAME=POINTER-FILE   STACK=1,0       SUCCESS=2
SOURCE=jmainfunction.b,0        PATHLENGTH=125
TIME=2535235.980

RELEASE:
FILENAME=/home/greg//4.0_rels/jbc4.1.4.3/tmp/jBASEWORK
KEY=JBASE_SET_ROOT*3    TRANSACTION=NO       STACK=1,0
SUCCESS=0       SOURCE=jmainfunction.b,0        PATHLENGTH=129
TIME=2535235.980

DELETE: FILENAME=/home/greg//4.0_rels/jbc4.1.4.3/tmp/jBASEWORK
TRANSACTION=NO  KEY=JBASE_SET_ROOT*3    STACK=1,0    SUCCESS=2
SOURCE=jmainfunction.b,0        PATHLENGTH=129  TIME=2535235.980

*

* JIMI output created by user greg on port 3 at Tue Mar  2 18:04:44
2004

*

READS:            0
WRITES:           0
OPENS:            3
DELETES:          1
CLEARFILES:       0
RELEASES:         0
RELEASEFILES:     0
SELECTS:          0
TRANSENDS:        0
TRANSABORTS:      0
PERFORMS:         0
PATHLENGTHS:      173
CALLS:            2
OPENSEQS:         0
READSEQS:         0
WRITESEQS:        0
SLEEPS:           0
INPUTS:           0
```

```
STOPS:            1
Machinename-~src/jbase4/4.1.3/jbase: unset JDIAG
Machinename-~src/jbase4/4.1.3/jbase:
```

EDM Output

The output generated by calling JIMITraceOn is simple text layout which can be sent either to a genuine operating system file or on Unix systems, to a terminal. Each line is usually a single line with each detail delimited by a tab character to make for easy parsing later. The use of the trace functions means a text file is created at the point specified by the user. Each line of text is usually a single event and consists of a number of fields delimited by a tab character.
The output may contain blank lines and also comment lines that start with * or #
The following is a list of output generated by various options on the call to JIMITraceOn.

Standard output with no additional options specified.

These will all appear as a summary when JIMI is turned OFF by various mechanisms e.g. using the JIMITraceOff API, or the user logging off, or the ‘jprof –j off’ command being executed.

```
READ:             nnn
WRITES:           nnn
OPENS:            nnn
DELETES:          nnn
CLEARFILES  :     nnn
RELEASES:         nnn
RELEASEFILES:     nnn
SELECTS:          nnn
TRANSENDS:        nnn
TRANSABORTS:      nnn
PERFORMS:         nnn
PATHLENGTHS:      nnn
CALLS:            nnn
OPENSEQS:         nnn
READSEQS:         nnn
WRITESEQS:        nnn
SLEEPS:           nnn
```

```
INPUTS:          nnn

STOPS:           nnn
```

Optional output with the DATABASE or FILEIO option.

These will appear as a summary when JIMI is turned OFF by various mechanisms e.g. using the JIMITraceOff API, or the user logging off, or the ‘jprof –j off’ command being executed.

```
TRANSTART:       NAME=XXX    COUNT=nnn

TRANEND:         COUNT=nnn

TRANSABORT:      COUNT=nnn

READ:            FILENAME=name_of_file   COUNT=nnn

READ:            FILENAME=name_of_file   COUNT=nnn

WRITE:           FILENAME=name_of_file   COUNT=nnn

DELETE:          FILENAME=name_of_file   COUNT=nnn

SELECT:          FILENAME=name_of_file   COUNT=nnn

CLEARFILE:       FILENAME=name_of_file   COUNT=nnn

RELEASE:         COUNT=nnn

RELEASE:         FILENAME=name_of_file   COUNT=nnn

READSEQ:         FILENAME=name_of_file   COUNT=nnn

WRITESEQ:        FILENAME=name_of_file   COUNT=nnn
```

Optional output with the DATABASE or OPEN option.

```
OPEN:            FILENAME=name_of_file   COUNT=nnn

OPENSEQ:         FILENAME=name_of_file   COUNT=nnn
```

Optional output with either the DATABASE or FILEIO option and VERBOSE option.

These appear before each statement is executed. Note that you should parse each field delimited by TAB, separately and not assume each field will be in any particular order.
Each line, as part of the **COMMON_INFO** as detailed later, has a SUCCESS=nn part and you should always be aware that if the value for nn is not 0, the operation failed and so some of the values might be incorrect e.g. during a READ if SUCCESS=2 then the record didn’t exist and the SIZE=nn part is invalid.

```
TRANSTART:       NAME=XXX    SYNC=ON|OFF **COMMON_INFO**

TRANEND:         **COMMON_INFO**
```

```
TRANSABORT:       **COMMON_INFO**

READ:             FILENAME=name_of_file   KEY=item_id LOCK=YES|NO
      TRANSACTION=YES|NO      SIZE=nnn    **COMMON_INFO**

READ:             FILENAME=name_of_file   KEY=item_id LOCK=YES|NO
                  TRANSACTION=YES|NO      SIZE=nnn    FIELD=nnn
                  **COMMON_INFO**

WRITE:            FILENAME=name_of_file   KEY=item_id SIZE=nnn
                  LOCK=YES|NO TRANSACTION=YES|NO **COMMON_INFO**

DELETE:           FILENAME=name_of_file   KEY=item_id
                  TRANSACTION=YES|NO      **COMMON_INFO**

SELECT:           FILENAME=name_of_file   **COMMON_INFO**

CLEARFILE:        FILENAME=name_of_file   **COMMON_INFO**

RELEASE:          TRANSACTION=YES|NO      **COMMON_INFO**

RELEASE:          TRANSACTION=YES|NO      FILENAME=name_of_file
                  **COMMON_INFO**

RELEASE:          TRANSACTION=YES|NO      FILENAME=name_of_file
                  KEY=item_id **COMMON_INFO**

READSEQ:          FILENAME=name_of_file   SIZE=data_size
                  **COMMON_INFO**

WRITESEQ:         FILENAME=name_of_file   SIZE=data_size
                  **COMMON_INFO**
```

Optional output with either the DATABASE or OPEN option and VERBOSE option.

```
OPEN:             FILENAME=name_of_file   TYPE=DB2|ORACLE|J3|J4
                  **COMMON_INFO**

OPENSEQ:          FILENAME=name_of_file   TYPE=SEQUENTIAL
                  **COMMON_INFO**
```

Optional output with the BRANCH or CALL option.

These will appear as a summary when JIMI is turned OFF by various mechanisms e.g. using the JIMITraceOff API, or the user logging off, or the ‘jprof –j off’ command being executed.

```
CALL:       SUBNAME=nnn COUNT=nnn
```

Optional output with the BRANCH or PERFORM option.

```
PERFORM:    COMMAND=Command Line Details  COUNT=nnn
```

Optional output with either the BRANCH or CALL option and VERBOSE options.

These appear after each statement is executed.

```
CALL:              SUBNAME=name_of_subroutine     **COMMON_INFO**

RETURN:            SUBNAME=name_of_subroutine
      PATHLENGTH=num_instructions    **COMMON_INFO**
```

Optional output with either BRANCH or PERFORM option and VERBOSE option.

```
PERFORM:           COMMAND=Command Line Details
      **COMMON_INFO**
```

Optional output with the TRANSACTION option.

This output appears irrespective of whether the VERBOSE option was used or not.

These appear following successful execution of a TRANSEND statement and detail in chronological order the WRITE's and DELETE's that make up the transaction.

COMMIT_BEGIN: and COMMIT_END: operations surround the entire transaction so just one set of these per transaction

WRITE_BEGIN: and WRITE_END: operations surround a WRITE operation inside a transaction so one set of these per WRITE

DELETE_BEGIN: and DELETE_END: operations surround a DELETE operation inside a transaction so one set of these per DELETE

WRITE_BEFORE_BEGIN: and WRITE_BEFORE_END: operations show any before image information associated with a WRITE

WRITE_AFTER_BEGIN: and WRITE_AFTER_END: operations show the after image information associated with a WRITE

DELETE_BEFORE_BEGIN: and DELETE_BEFORE_END: operations show any before image information associated with a DELETE.

Consider the following lines of code and assume the file is empty to start off with.

```
TRANSTART ELSE STOP
WRITE "Blood bank information line 1" ON FILEVAR,"ID1"
WRITE "Overwrite the blood bank information" ON FILEVAR,"ID1"
DELETE FILEVAR,"ID1"
TRANSEND ELSE STOP
```

This would give the following output with the characters in **bold type** showing comments added

COMMIT_BEGIN: Shows the start of the transaction information

WRITE_BEGIN: FILENAME=fb1 KEY=ID1 **the start of the first WRITE statement**

WRITE_AFTER_BEGIN: The start of the AFTER image of the first WRITE statement

000001: Blood bank information line 1 Line 1 of the AFTER image of the first WRITE statement

WRITE_AFTER_END: The end of the AFTER image of the first WRITE statement

WRITE_END: The end of the first WRITE statement

WRITE_BEGIN: FILENAME=fb1 KEY=ID1 **the start of the second WRITE statement**
WRITE_BEFORE_BEGIN: The start of the BEFORE image of the second WRITE statement
000001: Blood bank information line 1 Line 1 of the BEFORE image of the first WRITE statement
WRITE_BEFORE_END: The end of the BEFORE image of the second WRITE statement
WRITE_AFTER_BEGIN: The start of the AFTER image of the second WRITE statement
000001: Overwrite the blood bank information Line 1 of the AFTER image of the second WRITE statement
WRITE_AFTER_END: The end of the AFTER image of the second WRITE statement
WRITE_END: The end of the second WRITE statement
DELETE_BEGIN: FILENAME=fb1 KEY=ID1 the start of the first (and only) DELETE statement
DELETE_BEFORE_BEGIN: The start of the BEFORE image of the first (and only) DELETE statement
000001: Overwrite the blood bank information Line 1 of the BEFORE image of the first (and only) DELETE statement
DELETE_BEFORE_END: The end of the BEFORE image of the first (and only) DELETE statement
DELETE_END: The end of the first (and only) DELETE statement
COMMIT_END: Shows the end of the transaction information
****COMMON_INFO**** refers to a number of statements that are common to the trace messages …
STACK=3,2 this shows the current PERFORM/EXECUTE level is 3 and that the CALL stack level is 2. In other words this is the 3rd concurrent program executing for the user and in the current running program the subroutine nest is 2 i.e. there is another subroutine below it. Note the main program counts as a subroutine.
SOURCE=sourcename,line,sourcename,line,sourcename,line Shows where the trace message was called from and the stack of subroutines below it. Displayed to a maximum depth of 3 entries
PATHLENGTH=nnn Shows the current number of licensing instructions executed so far.
SUCCESS=nn where 0 is a success for the operation and any other value is the error number for the failure. If a failure, other parts of the event might be incorrect.
TIME=nnnnnn.nnn gives a decimal number in seconds since 1st Jan 1970 and so allows timings between different statements.

Source Quality Metrics (SQM)

There are options to both the BASIC command and the jcompile command such that the quality of the source code compiled can be evaluated. On the whole jBASE merely generates the statistics during a compilation. It is up to the caller to then interpret the results as he/she sees fit. Use the (J) option to generate the SQM data to the BASIC command and the –J option to the jcompile command.
In either case the output will be a simple text file whose name is of the format “sourcename.sqm” where sourcename is the original source name. The text file will be placed in the same file that the original source code was found.

The layout of the file is a multi-value file whose attributes are as follows. Use the text name given to each attribute rather than hard-coding an attribute number as the layout might change (found in jimiBh).

001 JBASE_SQM_VARIABLES

A multi-value list of all variable names local to the subroutine (or main program). In the case of dimensioned arrays the initial dimensions are given as defined e.g. FILEVAR(10,2)

002 JBASE_SQM_SUBROUTINES

A multi-value list of all subroutines called in the format SUBROUTINENAME<vm> Line Number. Note that a call such as “CALL @subname” will have “@subname” as the subroutine name allow you to differentiate between direct and indirect calls.

005 JBASE_SQM_INSERT:

A multi-value list of all the INSERT statements seen. The format of this is not accurately defined at the moment but will probably be in the format
‘NAME_OF_INSERT_FILE<sv>NUMBER_LINES_EQUATES<sv>NUMBER_LINES_CODE’

011 JBASE_SQM_SELECT:

A multi-value list of all ‘SELECT’ statements seen where each value is the line number it is seen on.

012 JBASE_SQM_RELEASE:

A multi-value list of all line numbers containing a ‘RELEASE’ statement.

013 JBASE_SQM_RELEASE_VAR:

A multi-value list of all ‘RELEASE VARiable’ statements seen where each value is the line number it is seen on.

014 JBASE_SQM_TRANSACTION

A multi-value list of all transaction statements where each value is of the format
‘TRANSTART|TRANSEND|TRANSABORT<sv>line_number’

015 JBASE_SQM_GOTO

A list of all line numbers where a GOTO is seen.

016 JBASE_SQM_RETURNTO:

A list of all line numbers where a ‘RETURN TO’ statement is seen.

019 to 049 RESERVED

050 JBASE_SQM_DATE

The date in internal format this SQM file was created

051 JBASE_SQM_TIME

The time in internal format this SQM file was created.

052 JBASE_SQM_JBASE_USER

The jBASE user name who compiled this SQM file.

053 JBASE_SQM_OS_USER

The operating system name of the user who compiled this SQM file.

Locking

CLEAR-ITEM-LOCKS

Use the CLEAR-ITEM-LOCKS command to clear a specific lock or all locks taken against a specific file. Use this command with jBASE hashed files only.

COMMAND SYNTAX

```
CLEAR-ITEM-LOCKS filename
```

```
CLEAR-ITEM-LOCKS filename, itemname
```

NOTES

Only a user with root privileges can issue this command

.

This is a UNIX/LINUX only command

.

Refer to the SHOW-ITEM-LOCKS command

.

If JEDI_NOSHMEM is set (i.e. JEDI_NOSHMEM=1) the CLEAR-ITEM-LOCKS will not function (?)

SHOW-ITEM-LOCKS

The SHOW-ITEM-LOCKS command displays details of locked items in jBASE-hashed files.

COMMAND SYNTAX

```
SHOW-ITEM-LOCKS
```

NOTES

Calling the JBCUserCustomizeDisplay subroutine can customize the output of SHOW-ITEM-LOCKS.

Use the CLEAR-ITEM-LOCKS command to explicitly clear a lock.

@USERSTATS

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

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

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

```
* Definitions for the data returned from @USERSTATS variable or from
* the record read from PROC file (using SYSTEM(1027) as file name)
*

EQUATE USER_PROC_PORT_NUMBER TO 1;* The port number
EQUATE USER_PROC_NUM_PROGRAMS TO 2;* Number of programs running in
this port
EQUATE USER_PROC_START_TIME TO 3;* Time user started in UTC format
EQUATE USER_PROC_PID TO 4  ;* Process ID
EQUATE USER_PROC_ACCOUNT TO 5;* Name of the account
EQUATE USER_PROC_USER TO 6 ;* Name of the user
EQUATE USER_PROC_TERMINAL_JBASE TO 7;* Name of terminal according to
jBASE
EQUATE USER_PROC_TERMINAL_OS TO 8;* Name of terminal as seen by OS
EQUATE USER_PROC_DATABASE TO 9;* Name of database connected to
EQUATE USER_PROC_TTY TO 10;* Name of TTY device
EQUATE USER_PROC_LANGUAGE TO 11;* Language
EQUATE USER_PROC_LISTENING_TIME TO 12;* Time in UTC the listening
thread last worked
EQUATE USER_PROC_MEM_FREE TO 13;* Amount of memory in heap space
free chain
EQUATE USER_PROC_MEM_USED TO 14;* Amount of heap space memory in use
EQUATE USER_PROC_THREAD_TYPE_INT TO 15;* Thread type expressed as an
integer
```

```
EQUATE USER_PROC_THREAD_TYPE_TXT TO 16;* Thread type expressed as a
text string
EQUATE USER_PROC_LICENSE TO 17;* License counters
EQUATE USER_PROC_STATS_OPEN TO 18;* Number of OPEN's performed.
EQUATE USER_PROC_STATS_READ TO 19;* Number of READ's performed.
EQUATE USER_PROC_STATS_WRITE TO 20;* Number of WRITE's performed.
EQUATE USER_PROC_STATS_DELETE TO 21;* Number of DELETE's performed.
EQUATE USER_PROC_STATS_CLEARFILE TO 22;* Number of CLEARFILE's
performed.
EQUATE USER_PROC_STATS_PERFORM TO 23;* Number of PERFORM's /
EXECUTE's performed.
EQUATE USER_PROC_STATS_INPUT TO 24;* Number of INPUT's performed.
EQUATE USER_PROC_UNUSED_1 TO 25;* Unused
EQUATE USER_PROC_OPEN_FILES_VIRTUAL TO 26          ;* Number of
files application thinks open
EQUATE USER_PROC_OPEN_FILES_REAL TO 27 ;* Number of files really
open by OS
EQUATE USER_PROC_USER_ROOT TO 28;* Application data set by
@USER.ROOT
EQUATE USER_PROC_PROCESS_TXT TO 29;* Text string to identify process
EQUATE USER_PROC_PROGRAM TO 41;* Program name and command line
arguments
EQUATE USER_PROC_LINE_NUMBER TO 42;* Line number currently being
executed.
EQUATE USER_PROC_SOURCE_NAME TO 43;* Name of source currently being
executed.
EQUATE USER_PROC_UNUSED_2 TO 44;* Unused
EQUATE USER_PROC_UNUSED_3 TO 45;* Unused
EQUATE USER_PROC_STATUS_TXT TO 46;* Status of program as a readable
text
EQUATE USER_PROC_STATUS_INT TO 47;* Status of program as an integer
EQUATE USER_PROC_CPU_USR TO 48;* User CPU time
EQUATE USER_PROC_CPU_SYS TO 49;* System CPU time
EQUATE USER_PROC_CPU_USR_CHILD TO 50;* User CPU time used by child
processes
EQUATE USER_PROC_CPU_SYS_CHILD TO 51;* System CPU time used by child
processes
EQUATE USER_PROC_USER_THREAD TO 52;* Application data set by
@USER.THREAD
```

Windows Telnet (jtelnetd)

jBASE is supplied with a windows telnet server to enable users to telnet into a windows server. After connecting to the telnetd service and giving your password you will be logged into the NT system in a simple passthru mode and be sitting at a normal DOS command shell. You may however change the startup program from cmd.exe to any jBASE program including the jsh.exe (jSHELL).

Account Naming Issues

Windows NT account names can exist in several name spaces. For example a Windows NT station in an Advanced Server domain has a local "Administrator" account and also has a corresponding "Administrator" account in its default domain. jBASE TCP Remote Logon Services use the following rules to disambiguate account names:

- If the account name is qualified (contains a backslash), the name preceding the backslash is first treated as a domain name, if there is no corresponding domain, then it is treated as a machine name. (Example: "MainDomain\Administrator").

- If the account name is not qualified (does not contain a backslash), the name is first looked up on the local machine. If the account name is not found, it is then looked up in the default domain of the machine.

User Environment

When users logon, their environment will contain all system-wide environment variables that are set on the local system. They will not receive their normal user environment settings at this moment in time (the Win32 API does not provide this ability), however, this will change in a future release. To circumvent this omission in the Win32 API, the jBASE TCP Remote Logon Services automatically set the following environment variables:

USERDOMAIN	The domain name in which the user account is defined.
USERNAME	The account name of the user.
HOMEPATH	The path name of the home directory of the user. If the user's home directory is a remote path, then this will contain the Universal Naming Convention (UNC) name of the user's home directory.
HOMESHARE	Always set to NULL. See the comments on remote directories below.
HOMEDRIVE	If the user's home directory is local, this contains the drive letter followed by a colon. If the home directory is remote, this is set to NULL.
JBC_TELNET_FLAG	This is set so that a user process can determine if it is being run via telnet session or via the console.

Because the remote user shares the drive map with all other users, it is not possible to automatically mount a remote user's remotely named directory on its normal drive letter. However many sites may wish to establish conventions whereby remote users are allowed to use certain drive letters remotely. Further, other environment variables may need to be set at logon. Thus the jBASE TCP Remote Logon Services execute the file "remote.cmd" if present in the user's home directory. If a remote user's home directory is specified as a remote directory, the user's initial directory will be "C:\". If desired, this can be overridden in "remote.cmd".

By default if the user has a "remote.cmd" file, this will be read by the telnet process, and parsed, the first command it finds will be executed. This file is intended mainly to allow the user to set up the environment, and so this parsing is primitive, for example only the first command it finds is executed, so a remote.cmd created by the jBASE "iju" for instance this will be the jsh. To disable this feature and have the NT command proceessor read the file, the registry setting ParseProfile should be set to 0. (see below). You should therefore avoid using command such ad "cd" and "call"

If supported by the telnet client, then the TERM variable will also be set.

Customization

WARNING : You can edit the registry by using Registry Editor (Regedit.exe or Regedt32.exe). If you use Registry Editor incorrectly, you can cause serious problems that may require you to reinstall your operating system. jBASE does not guarantee that problems that you cause by using Registry Editor incorrectly can be resolved. Use Registry Editor at your own risk.

NOTE: If you change or create any of the registry entries below, you will need to stop and then restart the services with the new or changed values before those entries will take effect.

Presenting a banner to the remote user after logon.

Create a registry entry of type REG_SZ with the name Banner under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

In the "Value data:" string, the sequence "\n" generates an end of line output, to use the '\' characters in the banner string use two '\' characters in a row.

EXAMPLE:

Name Type Data
Banner REG_SZ Welcome to jBASE\nThe Multi-value database of the future\n

Presenting a banner to the remote user prior to logon.

Create a registry entry of type REG_SZ with the name PreBanner under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

In the "Value data:" string, the sequence "\n" generates an end of line output, to use the '\' characters in the banner string use two '\' characters in a row.

EXAMPLE:

Name Type Data
PreBanner REG_SZ jBASE telnet process\nPlease log into system\n

Changing the logon prompt.

Create a registry entry of type REG_SZ with the name LogonPrompt under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE:

Name Type Data
LogonPrompt REG_SZ Windows User Name:

If this registry value is not present, the default logon prompt is "Account Name: ".

Changing the password prompt.

Create a registry entry of type REG_SZ with the name PasswordPrompt under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE

Name Type Data
PasswordPrompt REG_SZ Windows Password:

If this registry value is not present, the default password prompt is "Password: ".

Changing the Failed logon message.

Create a registry entry of type REG_SZ with the name FailedLogonMessage under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

In the "Value data:" string, the sequence "\n" generates an end of line output, to use the '\' characters in the banner string use two '\' characters in a row.

EXAMPLE:

Name Type Data
FailedLogonMessage REG_SZ \nFailed logon...Valid "WINDOWS" USER/PASSWORD?\n

If this registry value is not present, the default message is:

"Logon failed: unknown user name, password or privilege incorrect."

Logging user logon/logoff messages in the registry.

Create a registry entry of type REG_DWORD with the name LogEventLogon under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

If this registry entry is present and set to 0, then normal logon and logoff events will not be logged into the registry. If this entry is either not present, or it is set to 1, then an entry will be written into the registry whenever a user logs on or off the system.

Changing the default command processor.

Create a registry entry of type REG_SZ with the name CommandProcessor under registry key::

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE:

Name Type Data
CommandProcessor REG_SZ C:\JBASE30\BIN\JSH.EXE -

This will cause the jSHELL to start up and execute the login PROC in your MD if you have one.

By default, all services invoke CMD.EXE as the command processor. By adding the registry value you can override the command processor used by all users. NOTE: if you override the command processor, then the automatic environment setup using the REMOTE.CMD script will no longer be available.

Bump up priority while logging on.

Create a registry entry of type REG_DWORD with the name IncreaseLogonPriority under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

On loaded systems, the logon process can be slow. You can set this registry value to 1 if you want to increase the priority of the logon, AT THE EXPENSE OF OTHER PROCESSES ON THE SYSTEM.

If this registry entry is present and set to 1 then the priority of the logon process will be increased.

Changing default exit detection timeout.

Create a registry entry of type REG_DWORD with the name ExitDetectionTimeout under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE:

Name Type Data
ExitDetectionTimeout REG_DWORD 0x000001f4 (500)

Due to a technical issue related to pipes, the telnetd server has to “guess” when a command prompt is about to exit. This will be corrected in a later release. At those times when it this guess occurs, the telnetd server waits before doing a read of input. If this read timeout is too short, the exit detection doesn’t work, and so users need to type an extra input before the telnetd exit. (In other words, when the user types “exit” to CMD.EXE, the telnet client will not “hang up” until the user types an extra character.) Through internal experimentation we have found that a value of 400 milliseconds works well under most circumstances. However some customers have found this inadequate, so we provide this value. The value should be set to as small a value as provides the desired behavior.

If this registry value is not present, the default value is 400 milliseconds "0x00000190 (400)".

Changing the number of logon attempts

Create a registry entry of type REG_DWORD with the name LogonAttempts under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE:

Name Type Data
LogonAttempts REG_DWORD 0x00000005 (5)

If this registry value is not present, the default value is 3 attempts "0x00000003 (3)".

Parsing the “remote.cmd” file

Create a registry entry of type REG_DWORD with the name ParseProfile under registry key:

```
HKEY_LOCAL_MACHINE\Software\JAC\jBASE Telnetd Server\CurrentVersion
```

EXAMPLE:

Name Type Data
ParseProfile REG_DWORD 0x00000000 (0)

By default the telnet service will read any remote.cmd file that it finds in a users home directory, and execute the first command it finds. This is efficient and avoids having a command processor for each port, however, the commands that are supported are restricted to setting environment variables (set) and echoing text (echo). If you want the NT command processor to parse the file then set the value to 0.

Change the telnet port (jBASE 4.1 only)

The telnet port used by jtelnetd is stored in the %windir%\system32\etc\services item. jBASE will use the port signified by jtelnet (if present) or telnet if no jtelnet is present. To change the port for jBASE telnet only you can add this entry, which allows another telnet client to run on the port defined by "telnet".

```
jtelnet 2323/tcp  (This would start jBASE telnet process on port
2323)
```

or change telnet itself if jBASE telnet is the only telnet process to be run

```
telnet 2323/tcp  (This would start jBASE telnet process on port
2323)
```

The format of the services file will be similar to:

```
# Copyright (c) 1993-1999 Microsoft Corp.
#
# This file contains port numbers for well-known services defined
by IANA
#
# Format:
#
# <service name> <port number>/<protocol> [aliases...] [#<comment>]
#

echo 7/tcp
echo 7/udp
...
telnet 23/tcp
```

Troubleshooting / Technical Support

Event Log

The jBASE TCP Remote Logon Services report error messages to the Application Event Log. This log can be viewed using the Event Viewer application which can usually be launched by double clicking its icon in the Program Manager group: Administrative Tools. Make sure the Application event log is selected. (Its entry in the Log menu should have a check mark beside it... if not, select it.)

All jBASE TCP Remote Logon Services entries begin with the tag “jBASE”. Most of the error messages are self explanatory. Any error codes mentioned are standard Windows NT error codes as returned by GetLastError(). Your VAR should be able to help you with any errors here.

On rare occasions you may have a service failure. These are logged by the Service Control Manager in the System Log.

List of known problems

Eventlog says: jBASE Command Starter: CreateProcess: 5

This means that an account with Administrator privileges tried to logon, but that account does not have sufficient rights to execute transcmd.exe. The error cannot be caught sooner as Administrator accounts have a large number of privileges that allow all other operations up to this point to succeed.

Service Manager can't find .exe or Start failed: 2

Usually this error means that you've moved the software after you installed it. Services cannot be moved after installation as an absolute path name is stored. To correct the problem, remove, then reinstall the software.

Access denied when accessing a drive mounted with NET USE.

Because Windows NT wasn't initially designed with the idea of more than one interactive user logged on at the same time, oddities often occur when accessing remote drives via a drive letter in the shared drive map. Most users seem to have better results when accessing remote drives via the SUBST command. Accessing remote drives via UNC names also works and is recommended.

Login failure messages in the Event Log every minute.

This is due to a bug in Compaq's Insight Manager program. Compaq has provided a work-around: Upgrade to version 2.60b or newer of the Insight Manager product, then use the Control Panel for the Insight Manager product to disable telnet detection. Compaq is working on a better solution.

Program exits immediately with no output.

The most likely problem is that you have a needed DLL missing from your PATH. Because telnetd users need to see the error messages in a non-GUI form, we set things so that Windows NT will not let an error pop-up box occur for this error, but rather cause the error code to be returned to the calling program. In most instances this setting has the desired effect of letting a remote error see the error message, however, for reasons unknown to us, Microsoft has made the CMD.EXE suppress the error message for a missing DLL. This is particularly odd because it would be necessary for Microsoft to have written CMD.EXE to explicitly ignore that error.