

# D3 to jBase Differences

Pete Jewell

8th March 2002

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Environment</b>	<b>5</b>
2.1	VME . . . . .	5
2.2	MD . . . . .	5
2.2.1	Programs . . . . .	5
2.2.2	Files . . . . .	6
2.2.3	DXing Files . . . . .	7
2.3	SYSTEM . . . . .	7
2.4	Compiling and Cataloging . . . . .	7
2.4.1	BASIC and CATALOG . . . . .	7
2.4.2	Speed . . . . .	8
2.4.3	Listing BP files . . . . .	8
2.5	Select Lists . . . . .	9
2.6	Ports . . . . .	9
2.7	Debugger . . . . .	11
2.8	Devices . . . . .	12
2.8.1	Floppy Disks . . . . .	12
2.8.2	Tape Drives . . . . .	13
2.8.3	D3 Pseudo Floppies . . . . .	13
2.8.4	jBase Backups and Restores . . . . .	14
2.8.5	File Statistics . . . . .	15

<b>3</b>	<b>TCL Commands</b>	<b>16</b>
3.1	WHO . . . . .	16
3.2	LISTU . . . . .	16
3.3	WHERE . . . . .	16
3.4	FIND . . . . .	17
3.5	POVF . . . . .	18
3.6	CREATE-FILE . . . . .	19
3.7	jshelltype . . . . .	20
<b>4</b>	<b>BASIC</b>	<b>22</b>
4.1	Case Sensitivity . . . . .	22
4.2	Executing OS commands . . . . .	22
4.3	Permissions . . . . .	23
4.4	Sequential Files . . . . .	24
4.5	File Operations . . . . .	25
4.6	Secondary Indices . . . . .	26
4.6.1	Opening an Index . . . . .	26
4.6.2	Setting the Start Position . . . . .	26
4.6.3	Moving through the Index . . . . .	27
4.6.4	Alternative to SELECT . . . . .	27
4.7	Function Keys . . . . .	28

# 1 Introduction

This document intends to cover the differences between D3 and jBase I have encountered while porting our POSitive application.

There are doubtless more differences between the two systems than are covered in this document. It is worth bearing in mind that one of the benefits of jBase is that you don't have to do things in the same way that you're used to (although generally you can do that too), and the new way may be better than the old.

I have purposely not covered installation of jBase, as I will document that separately.

## 2 Environment

### 2.1 VME

Perhaps the biggest difference between D3 and jBase is the absence in jBase of a VME. In some ways jBase is more like uniVerse than D3 in that respect. Database files are simply located in the directory you place them into. Like uniVerse, there are at least two files for each FILE. For example, here's the directory listing for the PSTK.DAT file -

```
-rw-rw-rw- 1 pos pos 2252800 Feb 25 14:23 PSTK.DAT
-rw-rw-rw- 1 pos pos 12288 Feb 27 17:26 PSTK.DAT]D
-rw-rw-rw- 1 pos pos 1052672 Feb 25 14:45 PSTK.DAT]I
```

The first file contains the actual data, the second is the DICTIONARY, while the third holds the index data. If there's no secondary index defined for a file, the ]I file won't exist.

### 2.2 MD

You don't have to have a Master Dictionary. However, you can if you want to. If it exists in the directory you have invoked jBase in (more about this later), it is used to supplement the list of places that jBase will look for programs and data.

If you require PROCS, then these rely on there being an MD.

After you have created an MD, using the **CREATE-FILE** verb, run the program **UpdateMD** to populate it with the standard jBase entries.

#### 2.2.1 Programs

When you **BASIC** a program or subroutine, and then subsequently **CATALOG** it (more on this later), the MD doesn't get updated with an entry. If you need to find out if a particular program or subroutine is available, you can use the **jshow** utility (the next section will explain why I'm using the term *available* instead of *cataloged*) -

```

Usage:  jshow {-cfhpsv} Name {Name ...}
Where:
-c      Create time and file display.
-f      File name only search.
-h      Display this help screen.
-p      Program name only search.
-s      Subroutine name only search.
-v      Verbose mode.
Name    Name of file, subroutine or program.

```

I've yet to find a way of identifying all the programs/subroutines that are available (i.e., an equivalent to 'LIST MD WITH A1 EQ "VR" A5), however if you supply the `jshow` command the `-v` option, it will list all the places that it checks. These can then be listed to see if your program is there. For instance -

```
ls $HOME/bin
```

Would list the programs available to the current user from their HOME directory. Using shell meta characters like `$` works best in either the `msh` or `sh` shells (see section 3.7).

## 2.2.2 Files

Even with an MD in the current directory, if you create a file, it won't get an MD entry. This isn't a major problem, although it does complicate matters slightly if you want to generate a select list of all the files accessible from the current directory, including those defined via Q and F Pointers in the MD.

If all you want to do is produce a listing, you can use `LISTFILES`, however select lists have to be handled slightly differently. To generate a select list you do do this -

```
SELECT .
```

Which equates to `SELECT` all the files in the current directory. If you want to just select just the DATA files whose name begins with PSTK, use this -

```
SELECT . = "PSTK]" AND # "[D]"
```

If you don't include the `'AND # "[D]'` bit on the end, your select list will also include the DICTionary files.

The only way I've found, so far, of generating a select list of all the DATA files accessible from the current directory is to generate 2 select lists (one using `'.'`, the other `'MD'`), then combine them with a program (or by hand using `ED` or `JED`). It's a shame there's

not uniVerse's SET commands for combining lists (although we could always write them ourselves if we wanted to).

See also section 3.6 for a solution to the above problem, by using a PQN wrapper around the CREATE-FILE command.

See also section 2.5 for more information about select lists.

### 2.2.3 DXing Files

Typically, if you don't want a file to be backed up you adjust it's MD entry to have DX on attribute one. jBase has a different mechanism to handle this, called the jchmod command. To DX a file you would run the command like this -

```
jchmod -B filename
```

While to unDX it -

```
jchmod +B filename
```

For more details see the man entry (`man jchmod`) or the relevant section in the jBase System Administrators Manual.

## 2.3 SYSTEM

The default SYSTEM file is located in `/usr/jbc/src` as a J4 hashed file (i.e., a standard jBase data file). However it can be located anywhere by setting up the environment variable `JEDIFILENAME_SYSTEM` to point to it.

It usually contains a Q-Pointer to itself, (i.e., item name SYSTEM, with a Q on the first attribute), and is used to define where accounts are located, along with some behaviour regarding them.

I've not had much luck setting these up as yet, so will fill this bit in with more detail after I've had some time to play around with them a bit more. They're not vitally important to getting the POSitive system working.

## 2.4 Compiling and Cataloging

### 2.4.1 BASIC and CATALOG

jBase compiles BASIC code straight to C. This is similar to how D3 does things, but D3 still uses a pcode engine, and only uses C code in a few instances (so I've recently discovered from Bill).

When you **BASIC** a program, jBase compiles it into object code, and places it into the file containing the source code (BP file), prefixing the item name with an exclamation mark (!). However, you can't use that object code until it's been **CATALOGed**, which links it to the relevant shared jBase libraries, and by default places the resulting executable in `$HOME/bin` of the user you are logged in as.

When you **BASIC** a subroutine, again jBase compiles it into object code. The **CATALOG** stage however places the subroutine into a shared library stored in `$HOME/lib`. It *appears* to decide for itself which actual file the subroutine will be placed into, although it's best to have routines which call each other in the same shared library file.

The advantage of storing the subroutines in a shared library is that when multiple users call the same subroutine, there will only be one copy of it loaded into memory.

The default locations for programs and subroutines can be changed by setting a couple of environment variables. `JBCDEV_BIN` for the compiled programs, and `JBCDEV_LIB` for the shared libraries. If they're not set then the defaults described above are used.

## 2.4.2 Speed

The other thing you'll notice straight away (especially if you try to compile an entire BP file in one go) is that compiling is quite slow. This is perfectly normal, but can be sped up by making use of a unix programming facility called Makefiles (type '`man make`' at the shell prompt for further details). These define the programs and their relationships, so that you can purely compile program code which has changed, instead of recompiling everything. I'm told they will even handle changes to the `COMMON.INCLUDE` block (although this would obviously force a recompile of all programs and subroutines containing the `INCLUDE`).

I will be investigating Makefiles in more detail at a later stage.

## 2.4.3 Listing BP files

If you try and list a BP file you've compiled code in, you notice that there are lots of entries whose item names start with ! and \$ characters. The \$ prefixed items are compiled object code (See 2.4.1), while the ! prefixed items hold debugging information.

This can get annoying when you just want to see a list of programs. The easiest way of doing this is to perhaps create a PROC in the MD, maybe calling it `LISTS` (for `LIST SOURCE`), that rejects the unwanted items -

```
LISTS
001 PQ
002 HLIST
003 A2
004 H WITH *AO NE "$]" AND WITH *AO NE "!"
005 P
```

## 2.5 Select Lists

Select lists behave as you would expect under jBase. The only thing to watch out for is where they are stored. If you use a stock jBase system (i.e., you've not changed the default configuration after installation), then select lists end up in a shared file, usually `/usr/jbc/tmp/jBASEWORK`. This file is used for quite a lot of temporary information while jBase is running, as well as select lists.

A better approach is to either create a POINTER-FILE accessible from the account you are in, or set the environment variable `JBCLISTFILE` to point to the file you wish to use. When I say file, this can either be a jBase database file, or a directory. The advantage of using a directory is that the select lists are stored as simple text files (one item per line), and you don't have to worry about specifying a modulo and separation.

Even if you're using a directory to hold the select lists, you can still define either a Q-Pointer or F-Pointer to the location of the directory, which may be useful if you want to store them in one place system wide.

If you also set the environment variable `JBCLISTID` (entering `'export JBCLISTID='` before invoking jBase is enough to set it), the select list name will follow the convention **SEL\*account\_name\*list\_name**. Without `JBCLISTID` set (the default), select list names are as you define them with the `SAVE-LIST` verb.

According to Pat Pogson at jBase, there are a couple of caveats when using a directory to hold the select lists -

- If the 'item', in a directory, contains 'binary' data, specifically characters `x'0A'` and `x'0D'` (interpreted as carriage returns and line feeds) then these will be converted to Attribute marks, when the 'item' is 'read in' via jBASE, and character `x'FE'` will be converted to `x'0A'` when the item is written, unless special measures are taken when accessing such items.
- The item id, in a 'directory' should not contain the directory delimiter character `'/'` ( slash).

## 2.6 Ports

Port numbers in jBase can be allocated by various mechanisms. Without going into all the gory details, this is what you need to know.

- There is a configuration file (`/usr/jbc/config/Config_jPML`) which contains settings for default port ranges, both for foreground and background processes. The file also contains detailed instructions on it's usage.
- You can define what the port number will be by setting the environment variable `JBCPORTNO`. This must be done before a connection to jBase is made.

- You can link specific devices to port numbers by configuring them in the `/usr/jbc/config/Config_jPML` file. For instance, if you wanted to tie the port range 10 to 17 to the ports on a Cyclades multiport card, you would use -

```
port_def = /dev/ttyC0 10
port_def = /dev/ttyC1 11
port_def = /dev/ttyC2 12
port_def = /dev/ttyC3 13
port_def = /dev/ttyC4 14
port_def = /dev/ttyC5 15
port_def = /dev/ttyC6 16
port_def = /dev/ttyC7 17
```

- If none of the above approaches are taken, the port number is determined by way of the Linux file `/var/run/utmp`. I cannot work out exactly how this works, although the port number allocation seems to be based on the order of the network connections. I have asked jBase if there is already a way of tying a network address to a port number, and if not, can they add the feature. They replied with a simple solution which ensures each user always gets the same port number, and doesn't allow two connections from the same network address (so we wouldn't want to use it in house, but at customer sites it's perfect).

Their solution involves the creation of a directory to hold details of which ports are tied to which network names and address, and then some shell scripting.

1. Create a directory `/home/system/jbcPorts`, and make sure it's readable and writable by the users who will be connecting via the network.
2. Create a file in the newly created directory called `LAST_JBC_PORT`, and put the number 0 into it.
3. Insert the following section of shell script into the script used to start jBase, before the line which defines `JBCCONNECT` -

```
001 unset JBCPORTNO
002 jbcPortDir=/home/system/jbcPorts
003 if [ -f "$JBCPORTDIR/$REMOTEHOST" ] ; then
004   JBCPORTNO='cat $JBCPORTDIR/$REMOTEHOST'
005   export JBCPORTNO
006   echo "Loading from file, PORT no $JBCPORTNO from IP $REMOTEHOST"
007 else
008   currentPort='cat $JBCPORTDIR/LAST_JBC_PORT'
009   (( currentPort++ ))
010   echo $currentPort > $JBCPORTDIR/LAST_JBC_PORT
011   JBCPORTNO=$currentPort
012   export JBCPORTNO
013   echo $JBCPORTNO > $JBCPORTDIR/$REMOTEHOST
014   echo "New IP logged for JBC PORT $JBCPORTNO on IP $REMOTEHOST"
015 fi
```

The above will need to be adjusted so it doesn't try and assign a port when there is no \$REMOTEHOST (because they're on the console, or a serial port), but we have a solution!

## 2.7 Debugger

This is very different under jBase, although it's very flexible. There's no distinction between system and program debugger, as you might expect.

Whether or not a program drops out to the debugger prompt on encountering an error is governed by the environment variable `JBASE_WARNLEVEL`. To configure it you use a bit-mapped numeric value, where the behaviours available are -

- 1 The error message is written to the `jbase_error_trace` file
- 2 The error message is not displayed to the user's screen
- 4 The application does not enter the debugger

By default it's not set to anything, so any runtime error, in any program (including one supplied by jBase!) will drop you into the debugger.

In a default installation there are five errors which are affected by this setting -

- Floating point exception
- Divide by zero
- Math error
- Non-numeric value – ZERO USED
- Invalid or uninitiated variable

Further errors can be configured to obey the `JBASE_WARNLEVEL` behaviour by editing their definition in the file `/usr/jbc/jbc.init.err`, then regenerating the `/usr/jbc/jbcmessages` file with the `jmakerr` command. The documentation recommends editing a copy of the `jbc.init.err` file, and it's worth bearing in mind that with each release there will be a new `jbc.init.err` file which may or may not contain new errors (this isn't a massive problem, as we can find the new ones using the Linux `diff` command).

Instead of going into a massive amount of detail about how the debugger works, I recommend you consult the documentation on it in the jBase Knowledgebase (of which we have a copy locally), but here are the oft used commands -

- c** continue execution
- s** continue execution of the next line, then return to the debugger
- S** continue execution of the next subroutine, then return to the debugger
- v** equivalent to `/VARIABLE.NAME` without having the ability to modify the contents. If you want to modify the contents you would use `'v -m VARIABLE.NAME'` instead.
- w** display a window of source code. By default this displays 4 lines before and after the current one (i.e., 9 in total), but can be changed by specifying a number after the `w`.
- q** quit
- o** logoff

One feature worth knowing about is -

```
PROGRAM.NAME -Jd
```

Which will take you straight into the debugger at the start of your program.

## 2.8 Devices

jBase is very flexible in allowing you to define what devices you wish to be able to use the T- and associated commands with. By default they are all defined in the directory `/usr/jbc/dev`. There is a text file within that directory called `DEVICE_NOTES` which goes into detail about how you can configure the device files located in that directory.

The following sections talk about specific devices in relation to that directory

### 2.8.1 Floppy Disks

There is no `SET-FLOPPY` command as there is in R83 and D3. Instead you use one of the following -

```
T-ATT FLOPPY0 to attach to the A: drive
```

```
T-ATT FLOPPY1 to attach to the B: drive
```

All the normal T- commands are there, although there is no (A option to the T-CHK command, so you will need to repeatedly run it for the number of files you're expecting on the disk(s).

## 2.8.2 Tape Drives

There are, by default, preconfigured device files for the following -

DAT0 for DAT tape drives

HALF0 for half inch tape drives

SCT0 for quarter inch tape drives

Each one requires adjusting so that it points to the correct device file under Linux (see DEVICE\_NOTES for more details), although you can specify the device on the fly like this -

```
T-ATT SCT0 DEVICE=/dev/st0
```

## 2.8.3 D3 Pseudo Floppies

It is possible to read file and account saves, along with T-DUMPs, saved in D3 as Pseudo Floppies. I've not tried reading an uncompressed one (but I imagine it wouldn't work), so you'll need to uncompress the file first -

```
uncompress name_of_pseudo_floppy
```

And then use the D30 device I have setup specifically for this -

```
T-ATT D30 DEVICE=/path/to/pseudo_floppy
```

Here's the D30 file contents from the /usr/jbc/dev directory, it's an amended version of the FILE0 one, and only the first line is important -

```
JBC__EDIjdevTAPE -I D30 -M FILE -H12 -L R83,500 -B 500
#
# $Id: DevTemp.FILE0,v 3.1 1998/02/19 12:44:51 jbasedev Exp $
#
#-----
#
# This is the base definition for a device driver for tape
# support of type files. You can write to a normal Unix file
# by doing the following :
#
# T-ATT FILE0 DEVICE=FileName
#
# Where FileName is a normal Unix File for which you have
```

```

# permissions.
#
# The file /usr/jbc/dev/DEVICE_NOTES gives further details
# of all the command line options that may be passed to
# program jdevTAPE.

```

If you wanted to be allow multiple users to be able to read from more than one attached D3 Pseudo Floppy at a time, you could copy the above to a D31, D32, D33, etc, file, amending the part on the first line '-I D30' so that it matched the filename. This is used to ensure each user gets exclusive access to the device.

## 2.8.4 jBase Backups and Restores

Instead of performing account saves, it's recommended that we make use of the `jbackup` and `jrestore` commands instead. Both utilities will handle any file or directory you throw at it, and will respect any record locks that exist on jBase files.

I'd actually recommend using `jbackup` and `jrestore`, instead of simply copying files using Linux utilities. The reason being that the `jrestore` command will allow us to selectively restore data in a way similar to, but much more flexible than, a `SEL-RESTORE`.

Here's a couple of commands which will backup an 'Account' to a Linux file, compressing it in the process (so it's a bit like a compressed pseudo-floppy in that respect), and then check that the backup is readable. These commands work best in either the `sh` or `msh` shells -

```

jbackup -Apos -m1024 | gzip > /aptmp/pseudodir/FILESAVE
zcat /aptmp/pseudodir/FILESAVE 2>/dev/null | jrestore -P -q 2>&1 | grep
'^ERROR'

```

The first command merely backs up the data, compressing it through `gzip`. The second will display nothing if the backup is OK, otherwise it will display the `jrestore` errors, here's example output when there are errors -

```

ERROR! Missing segment mark, Buffer - 0288p9p/home/pos/POINTER-FILE/POS.OPEN.A
ERROR! Premature filemark detected restore aborted

```

I caused this by changing a couple of characters in the last segment of the gzipped `FILESAVE` file, however the `jrestore` errors seem to be consistent in that they always start `ERROR!`, so should be fairly easy to detect.

## 2.8.5 File Statistics

You can also use the `jbackup` command to produce file statistics. One way of doing this is -

1. Create a file to hold the file statistics information. If this is going to be in the current account you will probably want to configure it not to be backed up (See section 2.2.3).
2. If one doesn't already exist, create an F-Pointer in the MD for the file, with attribute 3 set to point to the file `/usr/jbc/jbackup]D`, so that you get the dictionaries for the file stats (this can also be achieved by using the `USING` clause when you run the `LIST` or `SORT` command).
3. Run `jbackup` in the following way -

```
jbackup -Aaccount_name -Sstats_filename > /dev/null
```

4. Marvel at the data flying up the screen! This is because you're running the command using the `jsh` shell, which deals with characters like `*` and `>` in a different way to a normal Linux shell (so you can do things like `LIST MD WITH *AO = "PCON.]"` for instance). Hit `Control-C` to interrupt the backup.

To fix this you can either hit `F3` to go into the mixed shell (`msh`), or, if you're using a terminal on which the function keys have been redefined, type in the command `'jshelltype msh'` (For more details see section 3.7). Now try the `jbackup` command again.

5. Once the `jbackup` command completes you should get some statistics about the number of files backed up. You can now use the information in the file you specified with the `-S` option to generate a file statistics report. Have a look at the `DICTIONARY` for a list of possible fields you can list. Here's an example -

```
LIST FILE.STATS NAME HASH.BUCKETQTY HASH.BUCKETSIZESHASH.RECORD.BYTES  
HASH.RECORD.COUNT SIZE ID-SUPP
```

6. If you now want to go back to using the `jsh` shell you can either hit `F1`, or type `'jshelltype jsh'`. `F2`, or `'jshelltype sh'` would get you a default `bash` shell (`sh`).

## 3 TCL Commands

### 3.1 WHO

You cannot specify 'WHO \*' to get a list of all the ports in use. In fact, currently this will dump you into the jBase Debugger (they're aware of it, it's a bug!). Issuing WHO will give you the port number and account name as you would expect -

```
jsh pos ~ -->WHO
21 pos
```

If you require a listing of all ports in use, use LISTU.

### 3.2 LISTU

Options available are P, which redirects to the printer, and N, which prevents the listing from paging.

	Port	PID	Account	Lang.	Location	Date	Time
	17	32349	pos	en_GB		01 MAR 02	15:29:57
	18	13546	pos	en_GB		04 MAR 02	12:28:32
	*21	13519	pos	en_GB		04 MAR 02	12:09:22
	22	13586	pos	en_GB		04 MAR 02	12:28:40
	10000	32437	pos	en_GB		01 MAR 02	16:27:52
	10001	32534	pos	en_GB		01 MAR 02	16:40:36

The processes on port numbers from 10000 onwards are background, or phantom, processes (but see section 2.6). In this particular instance they're handling print queues (see section 3.3, for the WHERE command).

### 3.3 WHERE

Here's the run down of the usage and options available with the WHERE command -

Usage: WHERE Ports (Options)

Where Ports can be a range of Ports of the form Port-Port.

Where Options can be:

A - All ports displayed.  
N - Nopage.  
P - Redirect output to printer.  
R - Raw output for scripts, no header, 1 line per logged on user.  
S - Display processes NOT waiting at jshell prompt.  
U - Suppress own process from display.  
V - Verbose output.

Here's an example of the output you get -

	Port	Device	Account	PID	Command
	17	0	pos	32349	/usr/jbc/bin/jsh -
				13637	POS.START
	18	1	pos	13546	/usr/jbc/bin/jsh -
	*21	4	pos	13519	/usr/jbc/bin/jsh -
				13640	WHERE
	22	5	pos	13586	/usr/jbc/bin/jsh -
	10000	tty	pos	32437	jspprint -Jb -Q COSMOS 1
	10001	tty	pos	32534	jspprint -Jb -Q STANDARD 0

As you can see, the last two processes are running jspprint, which identifies them as handling printer queues. The Device column is slightly misleading, as lines which show just a number are omitting the bit of the device name which would tell you whether they were a serial port, or network connection (i.e., ttyS0 or pts/0).

The Verbose option is very useful, telling you exactly which subroutine the process is in, including the line number! Although it's not as useful if your program is in the INPUT subroutine. You also get statistics about CPU and memory usage, along with other information about the number of DELETES, EXECUTES, and so forth, that the process has run.

## 3.4 FIND

There is no FIND command in jBase. However, there is a replacement called 'jgrep'. It works by default on jBase hashed files. If you want to search a directory (containing source code, perhaps), you have to supply it with the S option, otherwise it doesn't return anything. Here's the synopsis -

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

Where Options can be:

- 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

### 3.5 POVf

Although there is a POVf command available in jBase, it only lists the available space on the mounted filesystems. Here's an example -

```
| jsh pos ~ -->POVF
|      1- 1233280:   1233280       1233281- 1418800:   185520
| Total number of free blocks (512 bytes) : 1418800
```

Compared to -

```
| jsh pos ~ -->df --block-size=512
| Filesystem      512-blocks      Used Available Use% Mounted on
| /dev/hda6       3682736      2262376  1233280  65% /
| /dev/hda1       202970        6970    185520   4% /boot
```

If we go on the assumption that we're always going to store our data accounts in a sub-directory of /home, then the following basic code (taken from POS.CFU.FREE.SPACE) will return the number of available frames -

```
001 OS.EXE = CHAR(255):"s"
002 FRAME.SIZE = 4096 ;* J4 type file
003 EXECUTE OS.EXE:"df --block-size=":FRAME.SIZE CAPTURING DF.OUTPUT
004 DF.OUTPUT = TRIM(DF.OUTPUT)
005 CONVERT " " TO VM IN DF.OUTPUT
006 DF.LINES = DCOUNT(DF.OUTPUT,AM)
007 FRAMES.LEFT = 0
008 FOR X = DF.LINES TO 2 STEP -1 UNTIL FRAMES.LEFT
009   MOUNT.POINT = DF.OUTPUT<X,6>
010   BEGIN CASE
011     CASE MOUNT.POINT EQ "/home"
```

```

012     FRAMES.LEFT = DF.OUTPUT<X,4>
013     CASE MOUNT.POINT EQ "/"
014     FRAMES.LEFT = DF.OUTPUT<X,4>
015     END CASE
016 NEXT X

```

## 3.6 CREATE-FILE

When you create a file in the current account/directory, the MD does not get updated to reflect it's existence (see section 2.2.2). However, this is possible to achieve using a PQN PROC wrapper around the jBase CREATE-FILE command.

Pat Pogson at jBase has provided us with such a wrapper. It needs to be placed into the MD of the account you wish to run it from (I'm guessing it could also be linked to via the MD too, if we wanted to keep the PROCS in one file), and you also need to ensure that the environment variable JBCNOINTERNAL is set to 1. This way the MD is searched for jBase commands before running the jBase version (as long as you're running the command from the jsh prompt, or via a EXECUTE or PERFORM statement in a BASIC program).

Here's the PROC -

```

CREATE-FILE
001 PQN
002 C
003 C Uncomment the 'DEBUG ON' statement below to 'debug' the 'Proc'.
004 C 'DEBUG ON' is a jBASE enhancement to 'Procs'
005 C
006 CDEBUG ON
007 C
008 C Use the environment variable 'JBCRELEASEDIR' directory path
009 C to call the 'original' jBASE 'CREATE-FILE' executable from this
010 C 'Proc' wrapper
011 C
012 H$JBCRELEASEDIR/bin/CREATE-FILE
013 C
014 S2
015 99 A
016 C Loop to add all parameters from the original command line.
017 IF A G 99
018 C
019 C Issue the 'proper' ;-) CREATE-FILE command, using the 'PU' below.
020 C 'PU' is a jBASE enhancement to 'Procs'.
021 C
022 C 'PU' will issue the command via the 'kshell' in order that the
023 C '$JBCRELEASEDIR' part is expanded as required.
024 C

```

```

025 PU
026 C
027 C Now create the required entry in the 'MD'
028 C
029 C Get the current 'JEDIFILENAME_MD' setting. Using the jBASE
030 C 'Proc' enhancement 'IE' command. 'IE' will input the contents
031 C of the specified environment variable into the current
032 C 'Proc' input buffer parameter.
033 C
034 C The converse is the jBASE enhancement 'EE', to set an environment
035 C variable from a 'Proc'.
036 C
037 C The 'JEDIFILENAME_MD' environment variable defines the file used
038 C to hold 'Q' and 'F' pointers. This could be a file explicitly
039 C called 'MD', but doesn't have to be.
040 C
041 S99
042 IEJEDIFILENAME_MD
043 C
044 C Nothing new here if you have used 'PQN' 'Proc's before ;- )
045 C
046 F-OPEN 1 %99
047 X Failed to open JEDIFILENAME_MD file ('%99')
048 C
049 C Get the filename, as specified on the command line, into
050 C parameter %101.
051 C If parameter is 'DICT Filename' get 'Filename' from command line.
052 C
053 MV %101 %2
054 IF %101 = "DICT" MV %101 %3
055 MV &1.0 %101, 'F', "./*%101, "./*%101*" ]D"
056 IF %2 = "DICT" MV &1.2 ""
057 C
058 C And finally write the 'F' pointer to the file specified by the
059 C '$JEDIFILENAME_MD' environment variable.
060 C
061 F-W 1

```

### 3.7 jshelltype

This command doesn't exist in D3, nor does it need to. It's a new command added by jBase to make it easier to work at the TCL prompt (or jshell).

**jsh** Under normal usage you're running the jsh, which behaves mostly like a standard TCL prompt would, i.e., *doesn't* think that \*A1 means everything ending in A1, and *doesn't* think that > means redirect output to a file.

**sh** Next there is the normal Linux shell, which might be bash or ksh, depending on what the environment variable SHELL is set to. This is great for running commands like 'find' or 'grep', or the jbackup and jrestore commands.

**msh** Finally the mixed shell. This tries to intelligently handle the shell meta characters, so that they're correctly interpreted depending on context. I've not had much success running grep in this, although jbackup and jrestore work OK.

Switching between them is very easy if you've not reprogrammed the function keys (see section 4.7) in your terminal (emulator). You just hit F1 for jsh, F2 for sh, and F3 for msh.

Otherwise you can use the command -

```
jshelltype shell
```

Where *shell* is one of **jsh**, **sh** or **msh**.

## 4 BASIC

### 4.1 Case Sensitivity

The jBase compiler (BASIC/jBC) is *case sensitive*, so the two lines -

```
MYVAR = "Uppercase"  
myvar = "lowercase"
```

Would define two variables.

I repeat, the jBase compiler is *CaSe SeNsItIvE!*

### 4.2 Executing OS commands

In D3, when you EXECUTE a Linux command you have to precede it with a bang (!) character, i.e. -

```
EXECUTE "!ls -l /tmp" CAPTURING OUTPUT
```

With jBase this is no longer necessary, although I'd still recommend forcing the sh shell to be used. By default the command will be executed using the shell you ran your original program from (despite the manual saying it uses the sh shell by default).

However, you can force jBase to use a specific shell by preceding the Linux command with one of the following -

**CHAR(255):"k"** Korn shell (which is what the jSHELL or jsh is based on).

**CHAR(255):"c"** C shell (not recommended - see <http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/> for why).

**CHAR(255):"s"** Bourne shell (which under Linux is the bash shell in sh compatibility mode).

## 4.3 Permissions

When you connect to jBase, you keep the same user permissions as the user you're logged in as, unlike D3, where we had the d3 executable set `suid root` (meaning that you became the root user when it was executed).

This isn't a major problem, and is in fact a safety mechanism. However, there are times when it will be inconvenient, such as allowing users to change the date and time, a function that only the superuser (root) is permitted to do.

The solution, when these cases arise, is to wrap the program (that we need to execute as root) within a Perl script that has been set `suid root`. Here is the script I have used to solve the date and time problem -

```
001 #!/usr/bin/perl -w
002
003 #
004 # suid script to allow a user to set the date and time
005 #
006 use strict;
007 use English;
008
009 # make the following environment variables safe
010 delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
011 $ENV{PATH} = "/bin:/usr/bin";
012
013 my $date_params = $ARGV[0]; # pickup the first commandline parameter
014 my $safe_date;
015
016 # untaint the $date_params otherwise perl will complain
017 if ($date_params =~ /\^(\\d+)$/ ) {
018     $safe_date = $1;
019 } else {
020     die "Invalid date parameter supplied: $date_params";
021 }
022
023 # set our user and group ID's to 0, which means we're root
024 $UID = 0;
025 $GID = 0;
026
027 # finally, run the date and hwclock commands
028 exec("date $safe_date; /sbin/hwclock --systohc --utc")
029 or die "Unable to exec: date $safe_date; /sbin/hwclock --systohc
--utc";
```

The permissions on the file are -

```
-rwsr-sr-x    1 root    root    615 Mar  6 12:13 /usr/local/bin/user_date
```

The two s's indicate that the effective user and group id that will be used when the script is executed is the same as the owner and group of the file (in this case root). You set these permissions like this -

```
chmod 755 user_date    # this allows everyone to run the script, but only
root can alter it
chmod u+s user_date    # this sets the suid permission
chmod g+s user_date    # this sets the guid permission
```

You'll also notice that in the perl script itself I set the actual user and group id's to zero. This is because the `date` and `hwclock` commands check the actual user and group, not the effective user and group (which is what the suid and guid settings on the file actually changes).

Confused?

You may wonder why I used Perl, and not something like a simple bash or ksh script, to do this. The Linux operating system will not obey the suid and guid permissions on bash and ksh scripts, but it will with Perl.

## 4.4 Sequential Files

D3 has a range of C functions that allow you to read and write to sequential and binary files. These are -

```
%OPEN(), %READ(), %WRITE() and %CLOSE()
```

jBase has the `OPENSEQ`, `READSEQ`, `WRITESEQ`, `CLOSESEQ`, and `GET/GETX` commands instead. Where you're reading in chunks of a file from the Linux filesystem (as `POS.READ.LINUX.FILE` does), you can open the file with `OPENSEQ`, use `GET` to get the 32k chunks, and `CLOSESEQ` to close the file.

If you need to read a file on a line by line basis, the `READSEQ` command is limited to lines of 1024 bytes by default. However, using the `IOCTL` function it is possible to both redefine the End Of Line character (even to null, if you didn't want one), and specify a new maximum line length. The following example is adapted from a posting to the jBase tech mailing list by Daniel Klien of jBase (DanielK@jBASE.com) -

```
001 $INCLUDE JBC.h ;* this is required when using IOCTL
002 recordSize = 32767 ;* some ridiculous size
003 seqFileName = "/path/to/linux/file/filename.dat"
004 OPENSEQ seqFileName T0 seqFileVar ELSE STOP 201,seqFileName
005 * the following command isn't really required under linux
006 * (EOL = CHAR(10) anyway), but illustrates the command
```

```

007 IF IOCTL(seqFileVar, JIOCTL_COMMAND_CHANGE_DELIMITER, CHAR(10)) ELSE
STOP
008 * you need to ensure the recordSize variable is a number,
009 * not a string containing a number!
010 IF IOCTL(seFileVar, JIOCTL_COMMAND_SEQ_CHANGE_RECORDSIZE, recordSize)
ELSE STOP
011 EOF=0
012 LOOP
013  READSEQ seqRecord FROM seqFileVar ELSE EOF=1
014 UNTIL EOF DO
015  actuallyRead = LEN(seqRecord)
016  CALL PROCESS.RECORD(seqRecord, actuallyRead)
017 REPEAT
018 *
019 CLOSESEQ seqFileVar

```

## 4.5 File Operations

All the file operation commands (including READNEXT) allow you to set a variable in the event that something goes wrong, as well as THEN/ELSE. This variable contains a number from which you can deduce the problem.

For example -

```

001 errorVar = 0
002 READ record FROM fileVar SETTING errorVar THEN
003  * process record
004 END ELSE
005  IF errorVar THEN
006    BEGIN CASE
007      CASE errorVar EQ 128
008        CRT "No such file or directory"
009      CASE errorVar EQ 4096
010        CRT "Network error"
011      CASE errorVar EQ 24576
012        CRT "Permission denied"
013      CASE errorVar EQ 32768
014        CRT "Physical I/O error or unknown error"
015    END CASE
016  END
017 END

```

There is also the ONERROR clause, which all the above conditions except 128 will trigger if defined in the statement.

See the manual (the web version is best) for more details.

## 4.6 Secondary Indices

Just like D3, jBase supports secondary indices on hashed files. The syntax to create an index in the first place is slightly different though -

```
D3:
CREATE-INDEX PSTK.DAT A1 (0
jBASE:
CREATE-INDEX -cnov PSTK.DAT indexname BY 1
```

The main difference is that jBase indexes are named. See the jBase Knowledge Base pages for more information about the TCL commands available.

### 4.6.1 Opening an Index

Just like D3, when you want to make use of an index within a basic program you have to open it to an index file variable. Here's the syntax -

```
OPENINDEX filename, indexname TO index.var {SETTING error.var} THEN |
ELSE
```

Apart from the addition of the optional SETTING part, it's fairly similar to the D3 equivalent.

### 4.6.2 Setting the Start Position

Once you have the index open, you need to set where in the index you want to start. Here's the syntax of the extended SELECT command -

```
SELECT index.var {TO select.var} {ATKEY index-key{, record-key{, vmcount}}}
```

If you wanted to start at the beginning, you could just use -

```
SELECT index.var
```

At the first occurrence in the index of "JONES" -

```
SELECT index.var ATKEY "JONES"
```

There might be many "JONES" in the index, but you know that the item id of the one you probably want starts with a G, so you could use -

```
SELECT index.var ATKEY "JONES", "G"
```

To try and position yourself a bit nearer to the record you're after.

### 4.6.3 Moving through the Index

As you've probably already guessed, if we're using `SELECT` to position ourselves in the index, we're probably going to be using `READNEXT` to move through it - you're right!

You use both `READNEXT` and `READPREV` to move forwards and backwards along the index. Here's the syntax for `READNEXT`, it's the same for `READPREV` -

```
READNEXT KEY index-key{, record-key{, vmcount}} {FROM select.var} THEN  
| ELSE
```

The *index-key*, *record-key*, and *vmcount* get populated by the `READNEXT`. The *vmcount* is there because you can index right down to the multivalued on an attribute.

`READPREV` works in exactly the same way, but there are a few things to watch out for when you change from `READNEXT` to `READPREV`, and vice versa (taken from the jBase Knowledgebase article on `READPREV`) -

- When a `SELECT` statement is first executed a forward direction is assumed. Therefore if a `SELECT` is immediately followed by a `READPREV`, then a change of direction is assumed.
- During the `READNEXT` or `READPREV` sequence a next-key pointer is kept up to date. This is the record key, or index key to use should a `READNEXT` be executed.
- During a change of direction from forward (`READNEXT`) to backward (`READPREV`) then the next record key or index key read in by the `READPREV` will be the one preceding the next-key pointer.
- When the select list is exhausted it will either point one before the start of the select list (if `READPREV`'s have been executed) or one past the end of the select list (if `READNEXT`'s have been executed). Thus in the event of a change of direction the very first or very last index key or record key will be used.

### 4.6.4 Alternative to SELECT

There is one more index command available in Basic, which is the `SELECTINDEX` command. Although not as efficient as a combination of `SELECT` and `READNEXT`, it will populate a dynamic array with the results of your query on the index. Here's the syntax -

```
SELECTINDEX indexname{, index-key} FROM file.var {TO select.var}
```

It selects the open file pointed to by *file.var*, and returns the item id's sorted in the order defined in *indexname*. If you specify an *index-key*, the results will be limited to just those that match.

## 4.7 Function Keys

jBase provides us with an alternative to programming the function keys on a dumb terminal or Axel workstation to simplify the POS.INPUT subroutine. Instead of using IN we can use the Command Key subroutines. Their details are in the jBase Knowledgebase, under Keyboard Independence, but here's a quick run down.

You have to include the following lines of code in any program that uses this facility -

```
$INCLUDE JBC.h
$INCLUDE jCmdKeys.h
CALL CommandInit ;* Initialise keyboard mappings
```

And then everytime you just require to know what a keypress was, you call -

```
CALL CommandNext(KeyDescriptionCode, KeyData, Timeout)
```

Where:

**KeyDescription** is returned with a value which relates to the actual key pressed. The jCmdKeys.h file contains equates so that your program remains readable (see example below).

**KeyData** is returned with the actual code sequence produced when the key was pressed, or the ASCII code if it was an Alpha/Numeric key which was pressed.

**Timeout** is the number of seconds (expressed in tenths) you want the routine to sit there waiting for a key to be pressed. Setting it to zero disables the timeout.

The key pressed is still echoed to screen, so they recommend using ECHO OFF/ON around the subroutine call. Here's an example (taken from the jBase Knowledgebase), just to illustrate how easy it would be to use -

```
001 $INCLUDE JBC.h
002 $INCLUDE jCmdKeys.h
003 EQU Timeout TO 300; * 30 Seconds CALL CommandInit
004
005 LOOP
006   ECHO OFF
007   CALL CommandNext(RtnNo, KeyData, Timeout)
008   ECHO ON
009 UNTIL RtnNo = cmd_escape DO
010   BEGIN CASE
```

```

011     CASE RtnNo = cmd_alpha_numeric ;* Simple alpha-numeric
012         Line := KeyData
013         GOSUB CursorRight
014     CASE RtnNo = cmd_clear_end_line ;* Clear to end of line
015         CRT @(-4):
016     CASE RtnNo = cmd_insert_value ;* Insert a char(253)
017         Line := @VM
018         GOSUB CursorRight
019     CASE RtnNo = cmd_cursor_up ;* Move cursor up a line
020         GOSUB CursorUp
021     CASE RtnNo = cmd_cursor_down ;* Move cursor down a line
022         GOSUB CursorDown
023     CASE RtnNo = cmd_cursor_right ;* Move cursor to the right
024         GOSUB CursorRight
025     CASE RtnNo = cmd_cursor_left ;* Move cursor to the left
026         GOSUB CursorLeft
027     CASE RtnNo = cmd_error ;* Unknown command string
028         GOSUB InputError
029     CASE RtnNo = cmd_timeout ;* The input timed out
030         GOSUB CheckStatus
031     CASE RtnNo = cmd_winsize ;* Size of the X window changed.
032         GOSUB RefreshScreen
033     CASE 1
034         GOSUB InputError
035     END CASE
036 REPEAT

```