Using FoxPro and DDE to Store J Words

Antwerp Edition

John D. Baker
CompuServe: 71242,2702
Internet: 71242.2702@CompuServe.COM

August 14, 1994

Abstract

This article describes `JDict` a simple client server J source code dictionary built using the Windows DDE protocol, FoxPro 2.6 and J 7.0. The dictionary provides convenient and organized storage of J source code. FoxPro's high performance table indexing permits quick access to *many* thousands of J words. Furthermore this system can be easily adapted to any Windows programming environment that supports DDE and stores source code as ASCII text.

Contents

1 What is J?

J is a modern array oriented functional programming language that is completely described by an elegant little document—*The Dictionary of J* [?, Pages 61–97]. Programming in J[1] has a charming and distinctive flavor. Tasks decompose into scores of tiny programs that are collectively known as *words*. J terminology borrows from English grammar and J's words are roughly classified into nouns, verbs, adverbs and conjunctions.

The following, taken from Iverson's *The Dictionary of J*, gives a taste of the language and its precise grammar:

```
   fahrenheit=. 50
   (fahrenheit-32)*5%9
10
   prices=. 3 1 4 2
   orders=. 2 0 2 1
   orders * prices
6 0 8 2
   +/order*prices
16
   +/\1 2 3 4 5
1 3 6 10 15
   bump=. +&1
   bump prices
4 2 5 3
```

PARTS of SPEECH

`50 fahrenheit`	Nouns/Pronouns
`+ - * % bump`	Verbs/Proverbs
`/ \`	Adverbs
`&`	Conjunction
`( )`	Punctuation
`=.`	Copula

> Verbs act upon nouns to produce noun results; the nouns to which a particular verb applies are called its *arguments*. A verb may have two distinct (but usually related) meanings according to whether it is applied to one argument (to its right), or two arguments (left and right). For example `2%5` yields `0.4` and `%5` yields `0.2`.
>
> An adverb acts on a single noun or verb to its *left*. For example `+/` is a *derived* verb (which might be called *plus over*) that sums an argument list to which it is applied, and `*/` yields the product of a list. A conjunction applies to two arguments, either nouns or verbs.
>
> Punctuation is provided by parentheses that specify the sequence of execution as in elementary algebra.
>
> The word `=.` behaves like the copulas "is" and "are" and is read as such, as in "area is 3 times 4" for `area=.3*4`. The name `area` thus assigned is a pronoun and, as in English, it plays the role of a noun. Similar remarks apply to names assigned to verbs, adverbs and conjunctions.

[1] Why "J"? It is easy to type. Roger Hui [?]

[?, Page 62]

With a world awash in programming languages, why bother with J? The answer is simple. J notation is a spear in a world of bent spoons.

Consider the problem of writing a general purpose program for multiplying two or more polynomials.

The following *program* is a complete J solution to this problem?

```
   pp =. +//.@(*/)      NB. polynomial product verb

   c =. 1 3 3 1         NB. coefficients
   d =. 5 4 3 2 1

   c pp d
5 19 30 28 20 12 5 1

   e =. 2j5 3j7 0 1       NB. complex number coefficients
   f =. 1j2 0 3j7 0 0 2   NB. AjB is J's notation for A + Bi

   e pp f                 NB. _1 is negative 1 in J
_8j9 _11j13 _29j29 _39j44 0 7j17 6j14 0 2
```

It is important to understand that the word `pp` is not sneaking in a call to some built-in routine that just happens to multiply polynomials. The key to `pp` is the oblique `/.` conjunction which applies the derived verb `+/` (*plus over*) in an unexpected manner. J notation is so rich and versatile that you can program in it for years and still lose your breath over words as short as `pp`. The systematic modification of verbs and nouns by adverbs and conjunctions is the heart of functional programming and the source of J's special powers.

2 Why use a code Dictionary?

As `pp` demonstrated, J encourages brevity; words accumulate rapidly. Unless care is taken it is easy to submerge in a sea of words. There is a simple way to manage words—a *dictionary*. To organize my J programming I created `JDict`, a simple single-user client server J/DDE/FoxPro database system. J words are stored in FoxPro database tables. They are managed with a small set of J client words that interact, via DDE, with a FoxPro J server.

The advantages of a dictionary database

Organizing J words in such a database, referred to hereafter as a dictionary, has significant benefits.

- All defined J words are immediately ready for use. Putting words in script files often leads to time wasting word searches. Having your entire lexicon at hand improves productivity.
- There is only *one definition* for a given word. When copies of a word are scattered throughout many script files it is difficult to avoid multiple word definitions. Finding the current "version" is not always easy.
- There are no significant limits on vocabulary size. Script files can hold thousands of words but it is not practical to work with such large files.
- The *complete definition* of a word, (all code, examples and test scripts), can be quickly examined. Good English dictionaries contain far more than definitions. There are etymologies, synonyms, usage comments and illustrations. Similarly, *literate* [**?**] software documentation contains far more than source code. You should find descriptions of basic algorithms, remarks about coding techniques, references to published material, program test suites, detailed error logs and germane diagrams. Storing all this information in source code comments would unnecessarily clutter programs. A dictionary is the best place to put such material.
- *Relationships between words* can be stored. For example, when a word is inserted in the dictionary it can be analyzed for references to other words. These references are stored in a related uses or concordance table. An accurate concordance makes it easy to use words that depend on others in new contexts.
- Finally, *it's often easier to program in two or more languages than one.* FoxPro's array handling is as primitive as J's database capabilities. I wouldn't use J to build a database application nor would I use FoxPro to compute complex convolutions or permutation products. Splitting programming jobs into tasks that can be easily solved by cooperating systems will become more prevalent as DDE, OLE and other interprocess mechanisms mature.

I started this project to learn about DDE and ended up changing the way I program. A code dictionary is so useful that I've adapted `JDict` to other programming languages.

3 Installing the Dictionary

To install the dictionary Windows 3.1 must be installed on a 386, 486 or Pentium computer with at least 8 MB of RAM and 10 MB of free disk space.

The JDict system is distributed on three 1.44 MB MS-DOS 3 $\frac{1}{2}$ disks. The disks contain the JDict system, the FoxPro 2.6 runtime environment and a stripped down version of J 7.0. J 7.0 is shareware and Iverson Software Incorporated has granted the right to copy J 7.0 provided copies are not made for direct commercial advantage. I am placing JDict in the public domain and granting unlimited rights to copy and modify the system. I only ask that you choose new names for JDict variants.

JDict uses a standard Windows setup procedure. To install JDict do the following.

Installation Procedure

1. Start Windows and select the Program Manager.

2. Select the Run item from the File menu.

3. Insert disk #1 in drive a: and type: a:\setup

4. In a few moments setup will prompt you for a destination root directory and the name of the Program Manager group you want to place the JDict icon in.

   - The default destination root directory is: c:\jdict\ [2]
   - Put the JDict icon in the group of your choice.

   After you select the destination directory and Program Manager group setup will decompress and copy JDict files to appropriate directories: see Appendix 2. During this operation setup prompts for disks #2 and #3.

   When file transfer is complete setup starts the main jdict.exe dictionary program. jdict.exe initializes the private and public distribution dictionaries and sets profile.js in the c:\jdict\j7win directory. This process takes a few minutes on slow machines.

   When dictionary initialization is complete, JDict is loaded and ready to respond to client DDE commands.

5. To complete the installation put the J program icon in a Program Manager group. To do this:

   (a) Start the Windows File Manager.

   (b) Find jwin.exe in c:\jdict\j7win

   (c) Select jwin.exe and drag it into your chosen Program Manager group.

   Refer to your Windows user manual if you are not familiar with this operation.

[2] Example paths assume a default destination directory; if you choose another root make the appropriate substitutions.

6. Before starting J insure that a `c:\temp` directory exists. J and `JDict` put temporary swap files in this directory. See the J noun `Swap` (41) .

7. Now Double-click on J. J will load and execute the dictionary `profile.js` (54). J is set to echo script input. You will see `profile.js` run.

   When script execution ceases type the J sentence `did 0` to check your dictionary server. `JDict` should respond with a display similar to:

   ```
      did 0        NB. identify the current dictionary
   +-+-------------+
   |1|My Dictionary|
   +-+-------------+
   ```

   The dictionary is now ready to use.

8. For *experienced* LaTeX and TeX users only:

   `JDict` can generate a variety of structured J listings in LaTeX format. LaTeX is a widely used TeX macro package. To process the LaTeX `*.jxs` and `*.jxl`[3] files created by the dictionary it is necessary to install a few LaTeX preamble files and style files in your TeX input directories. All TeX and LaTeX related files, including the source for this document, are in the self-extracting `PkZip 2.09` archive file `c:\jdict\docs\jdictdoc.exe`. Unpack these files and read `READ.DOC` for further details.

[3] **J** te**X** **S**hort and **J** te**X** **L**ong

4 Using the Dictionary

To use JDict it is necessary to switch Windows tasks. There are many ways to switch Windows tasks. I prefer ALT-TAB’ing. If you have not mastered this skill refer to your Windows manuals.

To use the dictionary make sure that JDict and J are loaded. When JDict and J are ready switch to the J task.

Listing Dictionaries

If you successfully installed JDict you can access two dictionaries: an empty My Dictionary and a Public Dictionary. My Dictionary is your default dictionary. JDict opens the default dictionary when it is started. Any dictionary can be made the default.

To list known dictionaries use the pickd verb.

```
   pickd ’’      NB. list dictionaries
+-+-------------+-----------------+
|1|My Dictionary|Public Dictionary|
+-+-------------+-----------------+
```

Pick My Dictionary

```
   pickd ’M’     NB. distinct prefix is sufficient
+-+---------------------------------+
|1|My Dictionary is active; owner -> |
+-+---------------------------------+
```

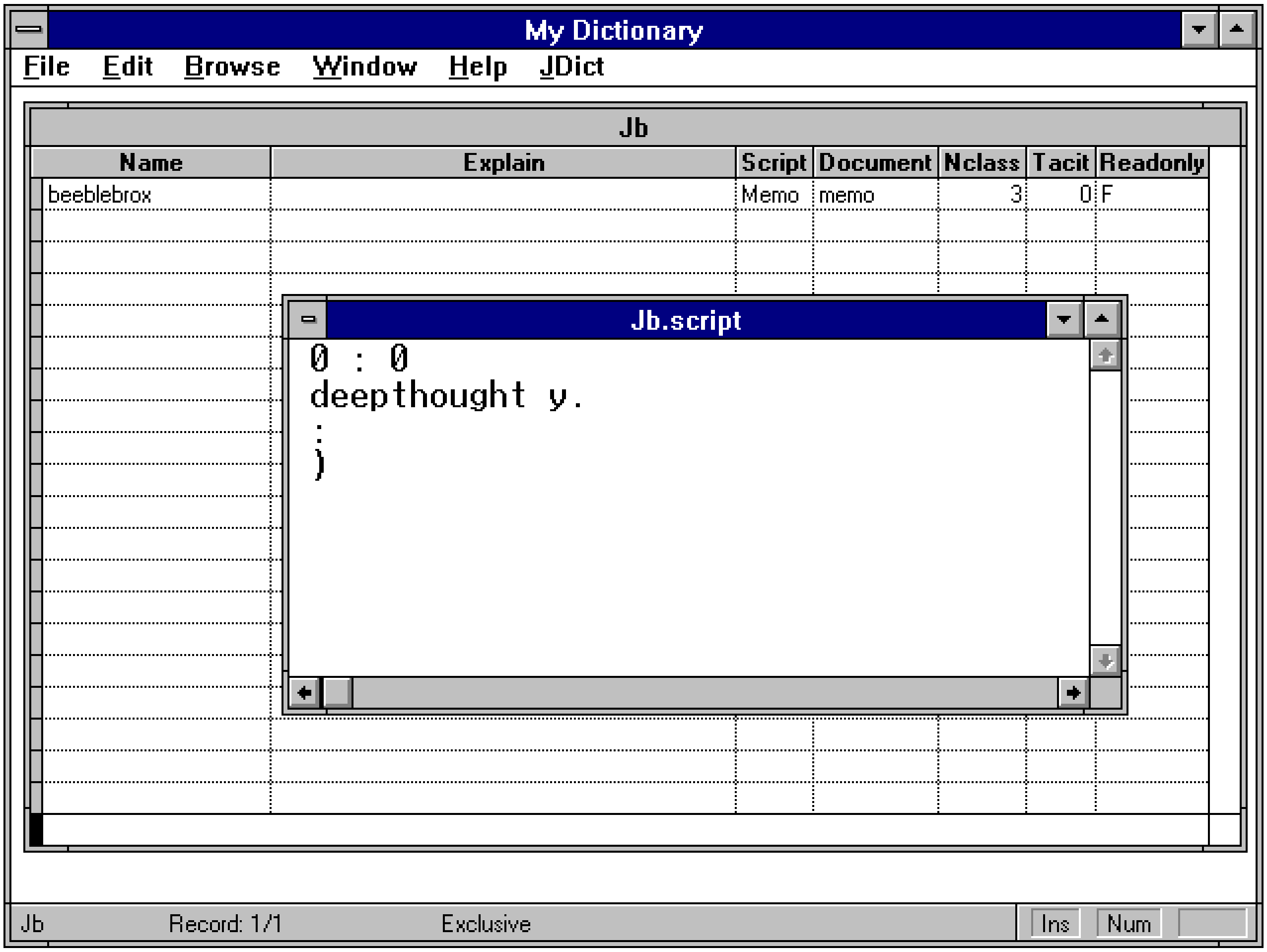
Putting words into a dictionary

First create a word in J.

```
   beeblebrox =. ’’ : ’’ NB. empty explict verb
```

Store beeblebrox in the dictionary with a put.

```
   put ’beeblebrox’      NB. store in dictionary
+-+-------------+
|1|My Dictionary|
+-+-------------+
```



Name	Explain	Script	Document	Nclass	Tacit	Readonly
beeblebrox		Memo	memo	3	0	F

```
0 : 0
deepthought y.
:
)
```

Jb Record: 1/1 Exclusive

Figure 1: Editing the J word `beeblebrox`.

The boxed result of `put` is typical of user words (see table 1 on page 14). The first item, a boxed 1 or 0, indicates success or failure. The second item is a message. In `put`'s case the second item is the dictionary's name.

`beeblebrox` is an empty shell. It can be edited in the `JDict` server.

Set the server to browse words.

```
      br 0          NB. set FoxPro to browse words
   +-+---------------+
   |1|Browsing words|
   +-+---------------+
```

Now switch to the `JDict` task.

To edit beeblebrox double-click on the SCRIPT memo field.

Enter the following definition of beeblebrox in the Jb.script window. See figure 1 on page 8.

```
0 : 0
deepthought y.
:
)
```

Double-click on the close box of Jb.script to save beeblebrox.

Getting words out of a dictionary

Switch back to J and fetch beelbebrox with get.

```
   get ’beeblebrox’
+-+-------------+-+
|1|beeblebrox__|3|
+-+-------------+-+
```

get’s result is also a boxed list. The second item is the word’s locale name and the third item is its *name class*. Name class is computed after defining the retrieved word: see def2 (42) .

Analyzing global references in words

Before executing beeblebrox analyze its global references with globs.

```
   2 globs ’beeblebrox’  NB. update global concordance
+-+------------------------+-----------+
|0|!jdict server: missing->|deepthought|
+-+------------------------+-----------+
```

2 globs detected a missing word deepthought. JDict will not store references to missing words.

Define and store deepthought and deeperthought.

```
   deepthought   =.  deeperthought   NB. empty tacit
   deeperthought =.  42"_            NB. constant verb

   put&> 'deepthought';'deeperthought'
+-+-------------+
|1|My Dictionary|
+-+-------------+
|1|My Dictionary|
+-+-------------+
```

Now update global references.

```
   2 globs&> 'beeblebrox';'deepthought'
+-+---------------------------------+
|1|My Dictionary concordance updated|
+-+---------------------------------+
|1|My Dictionary concordance updated|
+-+---------------------------------+
```

Listing global references in words

The immediate calls of a words can be listed with the `uses` verb.

```
   uses 'beeblebrox'
+-+-----------+
|1|deepthought|
+-+-----------+
```

The dyad of `uses` yields a complete list of globals.

```
   0 uses 'beeblebrox'
+-+-------------+-----------+
|1|deeperthought|deepthought|
+-+-------------+-----------+
```

Base Locale words not in the current dictionary

Get the words required to execute `beeblebrox`.

```
   get&>  'beeblebrox' ; }. 0 uses 'beeblebrox'
+-+---------------+-+
|1|beeblebrox__   |3|
+-+---------------+-+
|1|deeperthought__|3|
+-+---------------+-+
|1|deepthought__  |3|
+-+---------------+-+
```

Execute beeblebrox and store TheAnswer.

```
   ] TheAnswer =. beeblebrox 'Life the Universe and Everything'
42
```

Switch to the Public Dictionary.

```
   pickd 'Public'
+-+-------------------------------------+
|1|Public Dictionary is active; owner -> |
+-+-------------------------------------+
```

Find base locale words missing from the Public Dictionary.

```
   notput 4!:1 [ 2 3 4 5
+-+---------+----------+-------------+-----------+
|1|TheAnswer|beeblebrox|deeperthought|deepthought|
+-+---------+----------+-------------+-----------+

   pickd 'My'  NB. return to My Dictionary
+-+---------------------------------+
|1|My Dictionary is active; owner -> |
+-+---------------------------------+
```

Grouping dictionary words

Related words can be grouped together. A word can belong to any number of groups.

Create and populate an EARTH group.

```
   ’’ grp ’earth’
+-+---------------------+
|1|Group <EARTH> created|
+-+---------------------+
   ’earth’ grp~ ’beeblebrox’;’deeperthought’;’deepthought’
+-+---------------------+
|1|Group <EARTH> updated|
+-+---------------------+
```

Deleting groups and words

To make room for a new hyperspace bypass let’s delete `EARTH`.

```
1 del ’earth’
+-+---------------------+
|1|Group <EARTH> deleted|
+-+---------------------+
```

The `EARTH`’s words still exist.

```
   seek ’beeblebrox’   NB. search for "beeblebrox"
+-+----------+
|1|beeblebrox|
+-+----------+
```

Delete `beeblebrox` and the deep thinkers.

```
   del ’beeblebrox’
+-+---------------------------------------+
|1|<beeblebrox> deleted from My Dictionary|
+-+---------------------------------------+
   del&> }. 3 seek ’deep’  NB. words beginning with "deep"
+-+----------------------------------------+
|0|!jdict server: word is in use (words)   |
+-+----------------------------------------+
|1|<deepthought> deleted from My Dictionary|
+-+----------------------------------------+
```

Words cannot be deleted if they are in use. In the previous example `deeperthought` is referenced by `deepthought`. After `deepthought` has been deleted `deeperthought` can be removed: see `del` (17).

```
del&> }. 3 seek ’deep’
 +-+------------------------------------------+
 |1|<deeperthought> deleted from My Dictionary|
 +-+------------------------------------------+
```

Previous examples showed how to use some client words. The next section describes each client word in detail.

5 J Client Words

The client words listed in Appendix 5 define the J portion of the dictionary system. Client words can be divided into two classes: user words and utilities. User words are meant to be directly invoked. They test their arguments and return results in a consistent boxed list format. User words are summarized in table 1 on page 14. Utilities exist to serve user words. They seldom test their arguments and are not intended to be called directly. When client words are loaded they are placed in a *locale* [**?**]. Locales safely encapsulate words and help to organize J systems.

The J scripts `profile.js` and `jdict.js` load client words. `profile.js` is an example profile script. It creates a dictionary locale `jd`, loads all `jdict.js` words into it, and then defines a base locale user word interface. The resulting J locale structure is shown in figure 2 on page 15.

To use the dictionary it is sufficient to learn 18 J words and the FoxPro `JDict` menu. The rest of this section describes J user words.

Each word is documented as follows.

word *name class* short description of word. (code page number)

Monad `Hungarian notation`

```
Monad examples if any. Only input is shown.
```

Dyad `Hungarian notation`

```
Dyad examples if any.
```

Long description of word.

Usage	Example	Explanation
`br` *options*	`br 0`	Set FoxPro to browse dictionary word relationships.
`calls` *word*	`calls 'me'`	List all words that use a word.
options `calls` *word*	`1 calls 'me'`	List groups containing a word.
`del` *word*	`del 'me'`	Delete a word.
options `del` *name*	`1 del 'tools'`	Delete a dictionary group.
`did` *dummy*	`did ''`	Identify dictionary.
dummy `did` *dummy*	`'' did ''`	Basic dictionary statistics.
`disp` *word*	`disp 'me'`	Display word definition.
`fld` *empty*	`fld ''`	List all word fields.
`fld` *word*	`fld 'help'`	Retrieve the `DOCUMENT` field of a word.
field `fld` *word*	`'nclass' fld 'mean'`	Return the value of any `JB` table field.
`get` *word*	`get 'mean'`	Get a word.
locale `get` *word*	`'we' get 'rich'`	Get a word and put it in a locale.
`globs` *word*	`globs 'locale'`	Globals in a base locale word.
options `globs` *word*	`1 globs 'dictionary'`	Globals in a dictionary word.
`grp` *empty*	`grp ''`	List all dictionary groups.
`grp` *group*	`grp 'jdict'`	List the words in a group.
empty `grp` *group*	`'' grp 'idioms'`	Create a new group.
words `grp` *group*	`('a';'group') grp 'demos'`	Set the members of a group.
path `newd` *name*	`'c:\make\' newd 'A dictionary'`	Create a new dictionary.
`notput` *words*	`notput 'check';'us'`	Test words for dictionary membership.
`pickd` *dictionary*	`pickd 'J7 Tools'`	Pick a dictionary database.
`put` *word*	`put 'away'`	Store a word.
dummy `put` *word*	`0 put 'deep'`	Store a word and it's global name references.
word `rnto` *word*	`'old' rnto 'new'`	Rename a word.
`seek` *prefix*	`seek 'truth'`	List first matching word.
nclass `seek` *prefix*	`3 seek 'beauty'`	List all matching verbs.
nclass `seek` *empty*	`4 seek ''`	List all adverbs.
test `testw` *word*	`'test' testw 'me'`	Initialize a test case.
`uses` *word*	`uses 'what'`	Globals directly used by a word.
dummy `uses` *word*	`0 uses 'all'`	Recurse globals used by a word and return a complete call list.
word `wcopy` *copy*	`'make' wcopy 'acopy'`	Copy the full definition of a word.

Table 1: J Client User Words

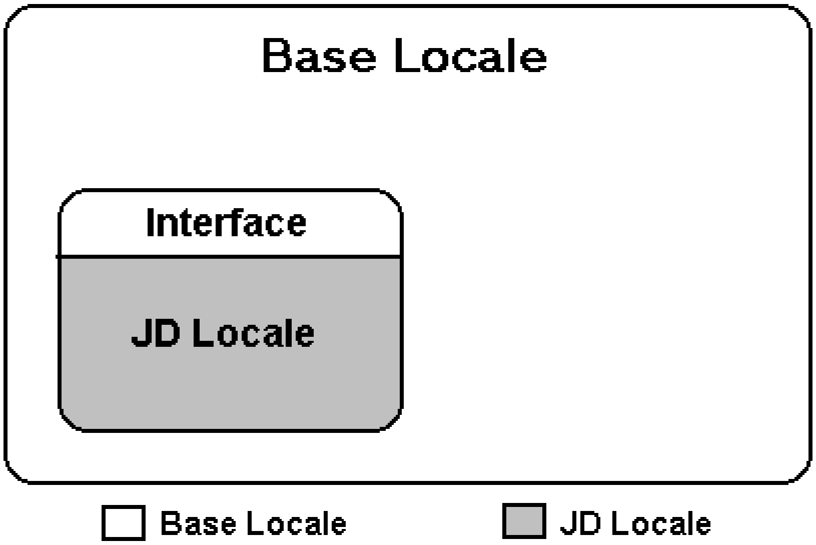


Figure 2: J Client Locale Structure

Hungarian notation is a naming convention that uses the first few letters of a name to describe its datatype. This convention has been adapted to J and extended to handle argument rank as well. Commonly used argument descriptions are shown in table 2 on page 17.

Some of the following examples refer to the verb (`names =. 4!:1`).

br *verb* sets FoxPro browse windows. (41)

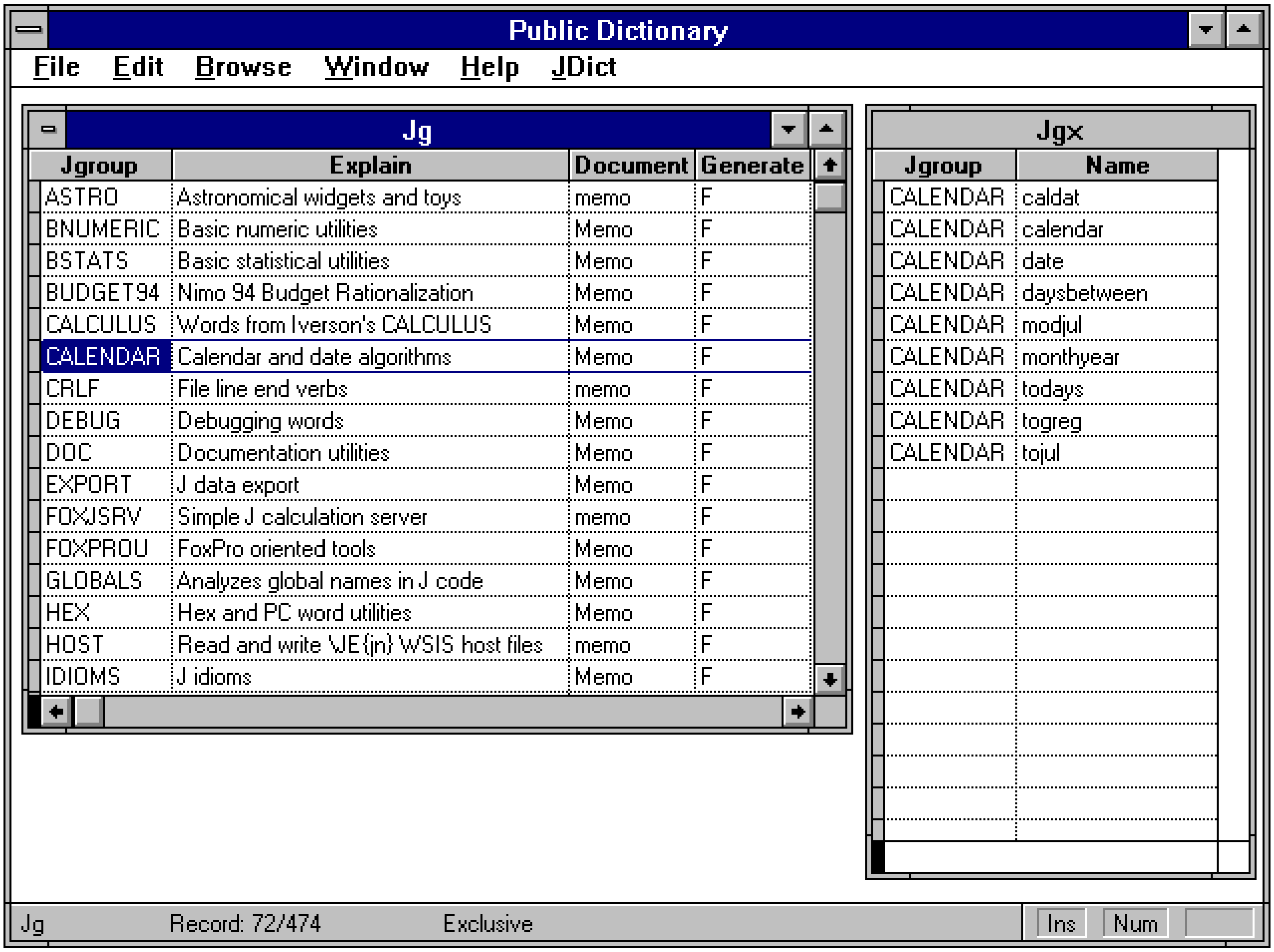
Monad `blDmsg =. br iaOption`

```
br 0        NB. browse words

br 1        NB. browse groups

br 2        NB. browse test cases

br 'wrong'   NB. bad argument
```

Dyad `undefined`

`br` signals the `JDict` server to browse word relationships. It's possible to browse words, groups, uses (concordance) and test cases. After executing `br` switch to FoxPro to observe the result. Figure 3 on page 16 shows the effect of `br 1`.

If `br` is given an invalid argument it returns a boxed list describing valid options.



Jgroup	Explain	Document	Generate
ASTRO	Astronomical widgets and toys	memo	F
BNUMERIC	Basic numeric utilities	Memo	F
BSTATS	Basic statistical utilities	Memo	F
BUDGET94	Nimo 94 Budget Rationalization	Memo	F
CALCULUS	Words from Iverson's CALCULUS	Memo	F
CALENDAR	Calendar and date algorithms	Memo	F
CRLF	File line end verbs	memo	F
DEBUG	Debugging words	Memo	F
DOC	Documentation utilities	Memo	F
EXPORT	J data export	Memo	F
FOXJSRV	Simple J calculation server	memo	F
FOXPROU	FoxPro oriented tools	Memo	F
GLOBALS	Analyzes global names in J code	Memo	F
HEX	Hex and PC word utilities	Memo	F
HOST	Read and write \JE{jn} WSIS host files	memo	F
IDIOMS	J idioms	Memo	F

Jgroup	Name
CALENDAR	caldat
CALENDAR	calendar
CALENDAR	date
CALENDAR	daysbetween
CALENDAR	modjul
CALENDAR	monthyear
CALENDAR	todays
CALENDAR	togreg
CALENDAR	tojul

Jg Record: 72/474 Exclusive Ins Num

Figure 3: FoxPro Windows set to browse groups by `br 1`

Hungarian	Description
`blclFields`	boxed list of character lists `Fields`.
`blDmsg`	boxed list `Dmsg`.
`blWords`	boxed list `Words`.
`blz`	empty boxed list — `z` for zilch.
`clWord`	character list `Word`.
`clz`	empty character list.
`ftMatrix`	floating table `Matrix`.
`iaOption`	integer atom `Option`.
`jtRoots`	complex table `Roots`.
`laItsybit`	logical (boolean) atom `Itsybit`.
`llBits`	logical list `Bits`.
`nlParms`	any numeric list `Parms`.
`ulValue`	universal list (any datatype) `Value`.
`uuDummy`	any array `Dummy`.

Table 2: Hungarian Notation Examples

calls *verb* lists words, groups and tests that use a word. (41)

Monad `blWords =. calls clWord`

```
calls 'reb'  NB. words that directly call "reb"

calls&> }. grp 'bstats'  NB. calls table for group "bstats"
```

Dyad `blCalls =. ia calls clWord`

```
0 calls 'reb'  NB. same as monad

1 calls 'reb'  NB. groups containing word "reb"

2 calls 'LF'   NB. test cases attached to "LF"

0 1 2 calls"0 1 'reb'  NB. full "reb" calls table
```

`calls` searches the dictionary's cross-reference tables and lists words, groups and test cases that call or use a given word. `calls` uses exact name matching.

The result of `calls` critically depends on the data in the cross-reference tables. If cross-references are not kept up to date by dyadic `put`'s and `globs`'s the resulting `calls` list will be incorrect.

del *verb* deletes word, groups and test cases. (43)

Monad `blDmsg =. del clWord`

```
del 'it'  NB. delete word "it"

del&> }. 0 seek 'boo'  NB. delete words beginning with "boo"
```

Dyad   `blDmsg =. ia del clWord`

```
0 del 'it'        NB. deletes "it" same as monad

1 del 'BOOHOO'    NB. deletes group "BOOHOO"

2 del 'testme'    NB. deletes test case "testme"

4 del 'EMPTYMAN'  NB. empties group "EMPTYMAN"

NB. delete all call references to "nowordknowsme"

4242 del 'nowordknowsme'
```

`del` deletes words, groups, test cases and *references* from the dictionary. Object deletion is controlled by an integer left `x.` argument code.

`0 del` deletes words. All references to a word are removed. *A word cannot be deleted if it is in use.* A word is in use if it:

- is called by another word.
- belongs to a group.
- is *attached* to a test case.

`1 del` deletes groups. A group is *any* collection of J words. The dictionary provides tools for manipulating groups from J and FoxPro. Words belonging to the deleted group remain in the dictionary. *Words can only be removed with* `0 del`*.*

`2 del` deletes test cases.

`4 del` empties groups. An empty group remains in the group table.

`4242 del` removes all references to a word. This “unuse” option should be applied with the utmost care as it:

- deletes *all* test cases attached to a word.
- insures concordance cross-references are out of date.
- removes the word from all groups.

Warning: do not directly delete objects in FoxPro. Dictionary deletion is a many table operation. If all tables are not correctly modified it's easy to violate database integrity.

did *verb* identifies the dictionary. (43)

Monad `blDmsg =. did uuDummy`

```
did ''     NB. name of current dictionary

did 0      NB. any argument returns name
```

Dyad `blDmsg =. uuDummy did uuDummy`

```
'' did ''  NB. basic statistics

0 did 0    NB. any arguments return statistics

did~0      NB. useful statistics idiom
```

Monadic `did` returns the name of the current dictionary and dyadic `did` returns basic dictionary statistics.

disp *verb* displays a word's dictionary definition. (44)

Monad `clScript =. disp clWord`

```
disp 'reb'  NB. script of word "reb"

NB. scripts of all verbs beginning with "b"

bees =. disp&.> }. 3 seek 'b'
```

Dyad `undefined`

`disp` gets the script text of a J word without defining it in the base locale. `disp` is a special case of the more general `fld` verb.

fld *verb* returns the value of any word *field*. (44)

Monad
```
ulValue    =. fld clWord  NB. result may be atomic
blclFields =. fld clzEmpty
```

```
fld ''       NB. boxed list of word field names

fld 'reb'    NB. get documentation for word "reb"

fld 'mean'   NB. documentation for "mean"

; fld&.> }. grp 'bstats'  NB. "bstats" group documentation
```

Dyad `u =. clField fld clWord`

```
'explain' fld 'reb'    NB. get short explanation of word

'creation' fld 'reb'   NB. get words's dictionary creation time

'nclass' fld 'reb'     NB. name class of "reb"

(}. fld '') fld&.> <'reb' NB. complete boxed "reb" record
```

`fld` and `disp` are the only dictionary user verbs that do not always return boxed lists. The result of `fld` is J's best approximation of the dictionary's FoxPro field value. Only numeric, character, date, logical and memo fields are returned by `fld`. Logicals are mapped to 0's and 1's. Memo and character fields are represented as character lists and must be less than 31,000 characters long. Dates are converted to `YYYYMMDD` integers.

get *verb* retrieves a word from the dictionary. (44)

Monad `blDmsg =. get clWord`

```
get 'me'                NB. get word "me"

get&> }. grp 'bstats'   NB. get group "bstats"

get&> }. calls 'me'     NB. get all words that call "me"
```

Dyad `blDmsg =. clLocale get clWord`

```
'come' get 'me'         NB. put "me" in locale "come"

NB. put group "bstats" into locale "stats"

(<'stats') get&> }. grp 'bstats'
```

`get` retrieves a word from the dictionary and defines it in a locale. The monad defines words in the base locale and the dyad defines words in named locales. The result of `get` is a three item boxed list. The last item is the *name class* of the word.

globs *verb* analyzes globals in dictionary words. (45)

Monad `blGlobals =. globs clWord`

```
globs 'me'    NB. globals in base locale word "me"
```

Dyad `blDmsg =. iaOption globs clWord`

```
0 globs 'me'  NB. globals in base locale "me" (monad)

1 globs 'me'  NB. globals in dictionary "me"

2 globs 'me'  NB. synchronize concordance
```

`globs` parses J word code and extracts global name references. This is a static name analysis; it will not detect dynamic global references via execute, indirect assignments and other *opaque* call methods.

`globs` uses *comment declarations* to mark opaque references. All names following `(*)=:` and `(*)=.` are *declared* global and local respectively.

```
NB. (*)=: We Are Globals
('Globals';'We';'Are') =: 1 2 3

NB. case matters in J (*)=. we are locals
('are';'we';'locals') =. i.3
```

`globs` can analyze words in the base locale and the dictionary. It can also bring the concordance table up-to-date when there is a discrepancy between a word and the table.

grp *verb* manipulates groups. (46)

Monad `blMembers  =. grp clGroup`
`blGrlist =. grp clzEmpty`

```
grp ''        NB. return a list of group names

grp 'bstats'  NB. list the members of group "bstats"

<:@#@grp&> }. grp ''  NB. number of words in all groups
```

Dyad `blDmsg =. clzEmpty grp clGroup`

```
'' grp 'new'  NB. create group "new"

(<'') grp&> 'this';'and';'that'  NB. create three groups
```

`grp` creates and modifies dictionary groups. A group is a collection of related J words. How the words are related is arbitrary. A group may, or may not, form a system. Groups are stored in the dictionary as a name cross reference table. A word can belong to any number of groups and operations on groups do not change words.

newd *verb* creates a new dictionary. (47)

Monad `undefined`

Dyad `blDmsg =. clPath newd clDictionary`

```
NB. paths must be fully qualified and end with \

'c:\yajd\' newd 'Yet Another J Dictionary'

NB. network drives are cool

'w:\lion\' newd 'Look I am a dictionary On Novell'
```

A dictionary is a set of database files stored in a directory. `newd` creates a standard directory structure and generates the corresponding database files. `newd` also appends a record to `JDICTCFG.DBF` to make new dictionaries visible to `pickd` (22).

Warning: `newd` is prone to DDE time-outs on slow machines. When a time-out occurs `newd` returns `!err: no server data`. With few exceptions the new dictionary has been created. You can verify dictionary creation by issuing a `pickd` command.

notput *verb* lists J base locale words that are not in the dictionary. (48)

Monad `blMissing =. notput blWords`

```
notput names 2 3 4 5  NB. base locale words not in dictionary

notput 'b' names 3    NB. verbs beginning with "b" not in dictionary
```

Dyad `undefined`

pickd *verb* picks a dictionary. (49)

Monad
```
blMsg    =. pickd clPrefix  NB. may be atomic
blDnames =. pickd clzEmpty
```

```
pickd ''     NB. list registered dictionaries

pickd ';'    NB. ";" open's default dictionary
pickd 'j7'   NB. pick first dictionary matching "j7"

pickd 'jn'   NB. first dictionary matching "jn"

pickd '\'    NB. unmatching strings close current dictionary
```

Dyad `undefined`

`pickd` picks a dictionary from a list of known dictionaries stored in `JDICTCFG.DBF`. `pickd` uses *prefix* matching. One and only one dictionary can be active at a time.

To select a dictionary it must be registered in `JDICTCFG.DBF` and exist in the specified directories. When `pickd` picks a dictionary it closes any open dictionary and attempts to open the selection. If `pickd` fails to open the selected dictionary it does not re-open the current dictionary. This provides a means of intentionally closing a dictionary.

Warning: picking a dictionary is a complex initialization process. Do not directly open data files in FoxPro.

put *verb* stores a word in the dictionary. (49)

Monad `blDmsg =. put clWord`

```
put 'away'    NB. store word "away" in dictionary

put&> 'n' names 3  NB. store base locale verbs beginning with "n"
```

Dyad `blDmsg =. uuDummy put clWord`

```
0 put 'deep'  NB. store word "deep" with its global references
```

`put` can store any noun, verb, adverb or conjunction with a definition smaller than 31,000 characters. For tacitly defined words, which includes all nouns, the definition cannot contain any `Xdictdels` (41) delimiters.[4]

Dyadic `put` stores a word and it's global name references. The name references are stored in the uses or concordance table. This table is crucial to the correct operation of `calls` and `uses`. When you have completed the definition of a word it's a good idea to store it with a dyadic `put`.

[4] This restriction applies only to J. From FoxPro you can add *new* words containing `Xdictdels`.

rnto *verb* renames a word. (50)

Monad `undefined`

Dyad `blDmsg =. clOld rnto clNew`

```
'change' rnto 'me'  NB. "change" becomes "me"

NB. rename "boo" "foo" "goo" to "fun" "bun" "son"

(_3 ]\'boofoogoo') rnto"(1) _3 ]\'funbunson'
```

`rnto` renames dictionary words. All references to the word are changed.[5] A word cannot be renamed if it is in use. To “unuse” a word see `del` (17).

Warning: do not directly rename words in FoxPro. Renaming is a many table operation. If all tables are not correctly modified it's easy to violate database integrity.

seek *verb* searches for words. (50)

Monad `blDmsg =. seek clPrefix`

```
seek ''      NB. first word (index order) in dictionary

seek 'boo'   NB. first word beginning with "boo"
```

Dyad `iaNclass seek clPrefix`

```
6 seek ''    NB. list all dictionary words

2 seek ''    NB. list all dictionary nouns

3 seek 'nm'  NB. all dictionary verbs beginning with "nm"

3 4 seek"0 'pq'  NB. table of "p" verbs and "q" adverbs
```

`seek` matches word name prefixes and positions the `JB` code table pointer. Switching to FoxPro after a `seek` lands you right on the matching word.

`seek` lists are limited to 31,000 characters. For dictionaries containing more than 3,000 words `6 seek ''` may fail. Of course restricted seeks like `seek 'me'` will work on very large dictionaries.[6]

[5] Names embedded in J code are not changed by `rnto`, only names stored in dictionary name fields are modified.

[6] Dictionaries have been assigned code `6` because they hold words like locales.

testw *verb* initializes a test case. (52)

Monad `undefined.`

Dyad `blDmsg =. clTest testw clWord`

```
'boo0' testw 'boo'  NB. create test case "boo0" for "boo"
```

Test cases are arbitrary J scripts that are stored in the dictionary. The dictionary provides basic facilities for combining the test cases associated with particular words and groups into single J scripts. The file `JDICTJAM.JS` is a script generated by combining all the test cases associated with the dictionary group.

uses *verb* returns a list of words used by a given word. (52)

Monad `blList =. uses clWord`

```
uses 'put'    NB. words directly called by "put"

fld&> }. uses 'put'  NB. document all words used by "put"
```

Dyad `blList =. uuDummy uses clWord`

```
0 uses 'put'  NB. all words required to execute "put"

get&> }. '' uses 'put'  NB. get words needed to run "put"
```

Monadic `uses` searches the concordance table and selects words that are *directly* referenced by a given word.

Dyadic `uses` gathers *all* the words used by a given word. The call list returned by dyadic `uses` is very useful for defining new groups.

Warning: dyadic `uses` is prone to DDE time-outs on slow machines. I have experienced no problems on 66MHZ 486's and Pentium's but have seen timeouts on 20MHZ 386's.[7]

wcopy *verb* copies a dictionary word. (52)

Monad `undefined`

Dyad `blDmsg =. clWord wcopy clCopy`

```
'reb' wcopy 'rebcopy'
```

`wcopy` copies an entire word. `READONLY` status is not copied.

[7] The FoxPro function `useall` that implements dyadic `uses` is an interesting iterative SQL `SELECT` based call tree search algorithm.

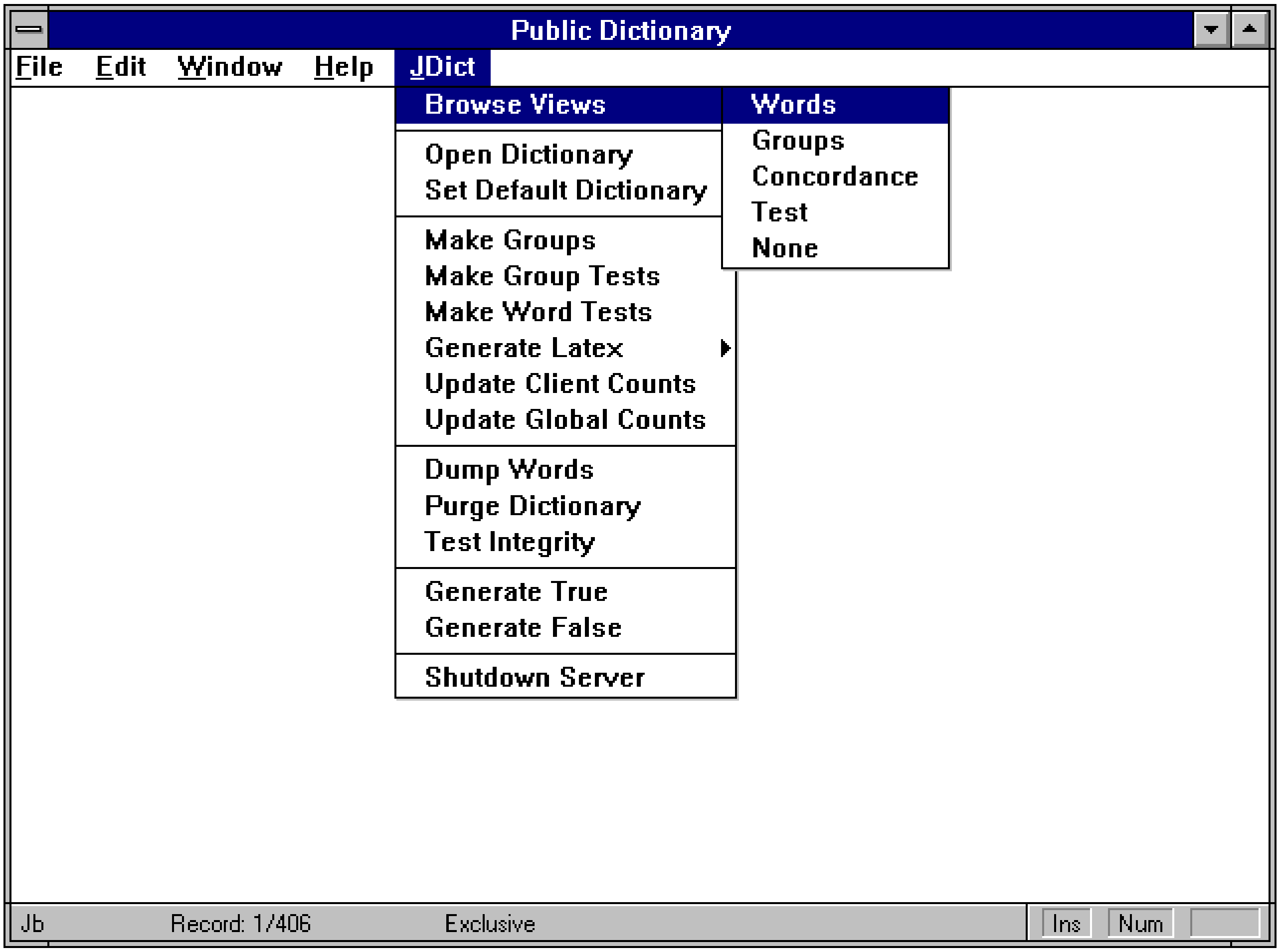


Figure 4: Main JDict server menu

6 Server Menu Programs

All server utilities can be accessed from the main JDict menu: see figure 4 on page 26.

This section describes all main JDict menu items. The FoxPro function or procedure that implements a menu item is noted on the right margin.[8]

Browse Views `dictbrow`

Selects a dictionary browse option. The possible options are presented on a submenu. Browse Views provides the same views as the J client word `br` (15).

Open Dictionary `dicscreen`

[8] FoxPro source code is distributed in three files `JDICTGO.PRG`, `JDICTMNU.PRG` and `JDICT.PRG`. See Appendix 2.

Prompts the user with a list of known dictionaries and then opens the selected dictionary. One, and only one, dictionary can be open at a time. Known dictionaries are stored in `JDICTCFG.DBF`.

Set Default Dictionary `dicscreen`

Prompts the user with a list of known dictionaries and then marks the selected dictionary as the default. The default dictionary is automatically opened when the `JDict` server is loaded. Setting a default dictionary does not change the current dictionary.

Make Groups `makegroups`

Generates J group `*.js` scripts and writes them to the dictionary's group directory. The group directory is a `JDICTCFG.DBF` setting.

Group script generation is controlled by the `GENERATE` field of the main groups (alias `JG`), table. When `GENERATE` is `True` a script is generated.

Warning: generated scripts are `LF` delimited. Conversion to `CRLF` delimiters may be necessary for some programs.

Make Group Tests `testgroups`

Builds J group test scripts and writes them to the dictionary's test case directory. Group test scripts have a `*.jgt` file extension. The test case directory is a `JDICTCFG.DBF` setting.

Test case inclusion is controlled by two logical fields. The `TESTON` field of the main groups table (alias `JG`) must be `True` to generate a group test script. For a particular case to be included in the generated script `CASESON`, of the main test case table (alias `JT`), must be `True`.

Make Word Tests `testwords`

Builds J word test scripts and writes them to the dictionary's test case directory. Word test scripts have a `*.jwt` file extension.

Case inclusion is controlled by the `CASEON` field of the test case table (alias `JT`).

Generate Latex `makelatex`

Displays a submenu that provides various LaTeX [?] listing options. Listings are written to the dictionary's group directory with `*.jxs` and `*.jxl` file extensions.

LaTeX file generation is controlled by the `READABLE` field of the main groups table (alias `JG`). The `PREAMBLE` field of the main groups table (alias `JG`) is written to the top of group listings.

Warning: the lines of `*.jxs` and `*.jxl` are `LF` delimited. Many PC TeX systems expect `CRLF` delimited lines.

Update Client Counts `cntclients`

The *clients* of a word are all the words that directly or indirectly reference it. A high client count implies a word is heavily used and should be changed with care.

Computing a word's client count requires a backward traversal of its call tree. This process can take a few minutes for dictionaries with hundreds of words.

Update Global Counts `globalcnts`

Stores the number unique of global references a word makes. This statistic is quickly computed from the concordance table.

Dump Words `dumpwords`

Dumps the current dictionary as three `ASCII` text files. The standard dump files are: `DUMPWORD.JS`, `DUMPWORD.JGX` and `DUMPWORD.JUX` These files are written to the dictionary's dump directory.

`DUMPWORD.JS` is a J script that rebuilds the dumped dictionary when run. See Appendix 4 page 39.

Note: not all dictionary fields are dumped. Calculated fields, Julian time stamps and `put` counts are omitted.

Make frequent dumps of your production dictionaries.

Purge Dictionary `jpurge`

Reclaims unused FoxPro file space.

Test Integrity `integrity`

Checks the referential integrity of a dictionary database and writes a report `INTEGREP.TXT` in the dictionary's dump directory.

Test Integrity checks primary and foreign keys and insures that necessary multi-table key relations hold. It does not detect *interfield* errors. For example, it's possible, when editing J words in server memo fields, to change the definition of a word from tacit to explicit. Unless the corresponding `TACIT` field is updated an interfield inconsistency results.[9]

A central part of integrity checking is rebuilding database index files. Database keys are extracted from `JDICTKEY.DBF`. If you add new keys to dictionary tables make sure they're entered in `JDICTKEY.DBF` or else the next integrity check will blow them away. Dictionary keys are so important that they are backed up in `JDICTKBK.DBF`.

[9] `get` ignores tacit to explicit errors but often fails on explicit to tacit errors.

Generate True `toggelbits`

`JDict` uses logical fields to turn script, LaTeX and test case text generation on or off. Generate True sets all *generation* logicals for the currently active browse window to `True`. Some logical fields like `READONLY` and `LOCALE` are not changed by Generate True or Generate False.

Generate False `toggelbits`

Similar to Generate True except logicals are set `False`.

Shutdown Server `closedict`

Closes all database files, releases DDE services and exits to Windows. *This is the only way to exit the server program.*

7 Summary

`JDict` evolved from my efforts to extend my knowledge of DDE, J and FoxPro. The result is a useful J programming tool and a collection of handy FoxPro programs. Parts of the dictionary have found use in "serious" bread-and-butter applications.

I am placing `JDict` in the public domain; feel free to use and modify it. I only ask that you give new names to any `JDict` variants. Like all authors I am interested in hearing from readers. I can be reached at the `Internet` address: `71242.2702@CompuServe.COM`

John D. Baker
Glenburnie, Ontario

References

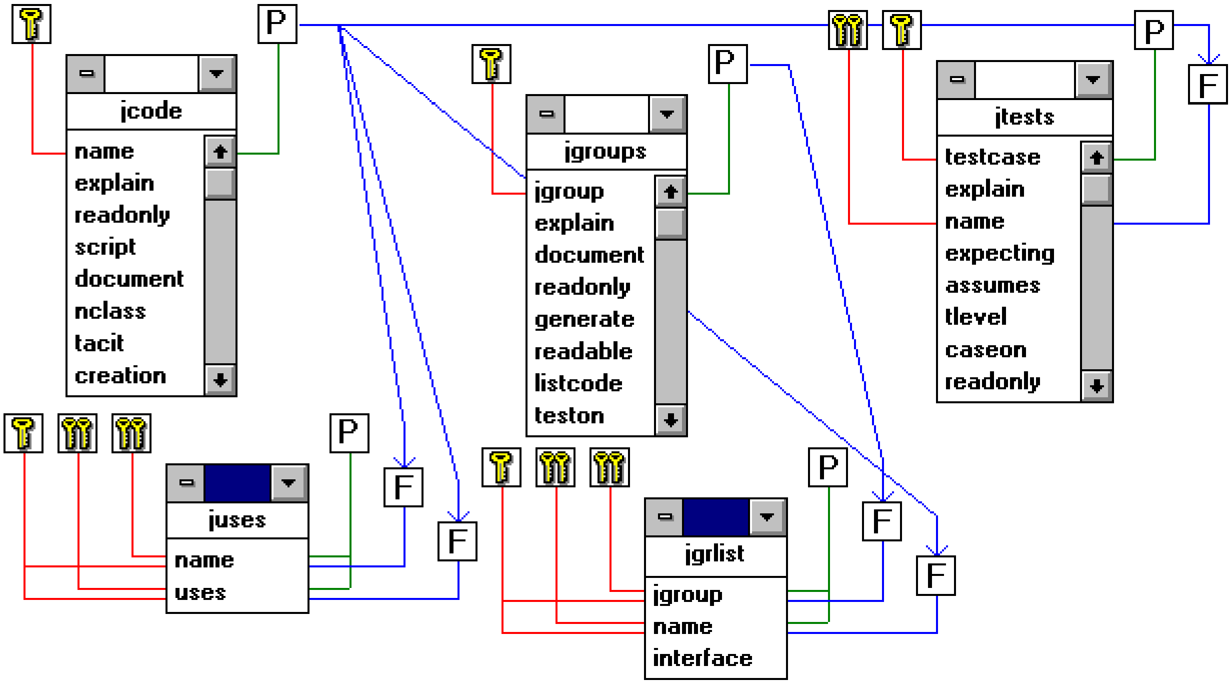


Figure 5: Dictionary Table Map

Appendix 1 - Dictionary Table Map

A dictionary is a relational database. The design of a standard dictionary is shown in figure 5.[10]

The icons of figure 5 are:

1. **P** labels primary key fields.

2. **F** labels foreign key fields.

3. A single *key* icon denotes a unique index.

4. A double *key* icon denotes a nonunique index.

5. Tables with white headers store dictionary objects. Tables with black headers maintain cross-references.

6. Relationships between files are shown with directed lines.

[10] File names may vary depending on `JDICTCFG.DBF` settings.

Appendix 2 - Dictionary Directory Structure

The *default* `JDict` directory structure is listed below. Many file names can be changed by editing the configuration table `JDICTCFG.DBF`.

`c:\jdict\config.fpw` FoxPro configuration file.

`c:\jdict\chalice.ico` J8 `JDict` icon file.

`c:\jdict\foxtools.fll` FoxPro dynamic link tools library.

`c:\jdict\foxw2600.esl` FoxPro runtime support library.

`c:\jdict\jdict.exe` Main `JDict` executable.

`c:\jdict\jdict.js` J client word script.

`c:\jdict\jdict.prg` FoxPro server procedure file.

`c:\jdict\jdictcfg.dbf` Main `JDict` configuration table.

`c:\jdict\jdictgo.prg` Startup up procedure.

`c:\jdict\jdictjam.js` J client test script.

`c:\jdict\jdictkbk.dbf` Server key definition table backup.

`c:\jdict\jdictkey.dbf` Server key definition table.

`c:\jdict\jdictmnu.prg` Server menu definition procedure.

`c:\jdict\jdictubk.dbf` Resource file backup.

`c:\jdict\jdictubk.fpt` Resource file backup memos.

`c:\jdict\jdictusr.dbf` Resource file.

`c:\jdict\jdictusr.fpt` Resource file memos.

`c:\jdict\jdtemp.dbf` Temporary table.

`c:\jdict\wheel.ico` J7 `JDict` icon.

`c:\jdict\docs\jdictdoc.exe` `PkZip 2.09` self-extracting archive of `JDict` documentation.

`c:\jdict\j7win\cpyright.txt` J copyright notice.

`c:\jdict\j7win\jserv387.exe` J 387 executable.

`c:\jdict\j7win\jservemu.exe` J emulation executable.

`c:\jdict\j7win\jwin.exe` J 7.0 Windows session manager.

`c:\jdict\j7win\jwin.hlp` J 7.0 Windows help file.

`c:\jdict\j7win\profile.js` `JDict` specific profile.

`c:\jdict\j7win\profile.txt` original profile text.

`c:\jdict\j7win\status.txt` J status.

`c:\jdict\j7win\xenos.txt` Description of J's `!:` facilities

`c:\jdict\j8sup\j8dict.js` J8 client word script.

`c:\jdict\j8sup\j8dict.exe` J8 dictionary server executable.

`c:\jdict\ours\jcode.cdx` Public Dictionary code index file.

`c:\jdict\ours\jcode.dbf` Public Dictionary code table.

`c:\jdict\ours\jcode.fpt` Public Dictionary code memo file.

`c:\jdict\ours\jgrlist.cdx` Public Dictionary group cross-reference index file.

`c:\jdict\ours\jgrlist.dbf` Public Dictionary group cross-reference table.

`c:\jdict\ours\jgroups.cdx` Public Dictionary group index file.

`c:\jdict\ours\jgroups.dbf` Public Dictionary group table.

`c:\jdict\ours\jgroups.fpt` Public Dictionary group memo file.

`c:\jdict\ours\jtests.cdx` Public Dictionary test index file.

`c:\jdict\ours\jtests.dbf` Public Dictionary test case table.

`c:\jdict\ours\jtests.fpt` Public Dictionary test memo file.

`c:\jdict\ours\juses.cdx` Public Dictionary uses or concordance cross-reference index file.

`c:\jdict\ours\juses.dbf` Public Dictionary uses or concordance table.

`c:\jdict\ours\dmp\` Public Dictionary dump file directory.

`c:\jdict\ours\ex\anova2c.js` Public Dictionary external J script directory.

`c:\jdict\ours\ex\calltree.js` — example external script.

`c:\jdict\ours\ex\numword.js` — example external script.

`c:\jdict\ours\gr\` Public Dictionary group script directory.

`c:\jdict\ours\jt\` Public Dictionary test script directory.

`c:\jdict\ours\lo\` Public Dictionary locale script directory.

`c:\jdict\mine\dmp` My Dictionary dump directory.

`c:\jdict\mine\ex` My Dictionary external script directory.

`c:\jdict\mine\gr` My Dictionary group script directory.

`c:\jdict\mine\jt` My Dictionary test script directory.

`c:\jdict\mine\lo` My Dictionary locale script directory.

`c:\jdict\mine\jcode.cdx` My Dictionary code index file.

`c:\jdict\mine\jcode.dbf` My Dictionary code table.

`c:\jdict\mine\jcode.fpt` My Dictionary code memo file.

`c:\jdict\mine\jgrlist.cdx` My Dictionary group cross-reference index file.

`c:\jdict\mine\jgrlist.dbf` My Dictionary group cross-reference table.

`c:\jdict\mine\jgroups.cdx` My Dictionary group index file.

`c:\jdict\mine\jgroups.dbf` My Dictionary group table.

`c:\jdict\mine\jgroups.fpt` My Dictionary group memo file.

`c:\jdict\mine\jtests.cdx` My Dictionary test index file.

`c:\jdict\mine\jtests.dbf` My Dictionary test case table.

`c:\jdict\mine\jtests.fpt` My Dictionary test memo file.

`c:\jdict\mine\juses.cdx` My Dictionary uses or concordance cross-reference file.

`c:\jdict\mine\juses.dbf` My Dictionary uses or concordance table.

Appendix 3 - Dictionary Table Descriptions

JCODE alias JB is the main J code table. It stores all word specific information. The primary key is `NAME`.

1. `NAME` *Character 20* is the word's name.
2. `EXPLAIN` *Character 50* is a brief explanation of the word.
3. `READONLY` *Logical 1* is a flag that when set to `True` in FoxPro prevents J clients from altering or deleting a word.
4. `SCRIPT` *Memo 10* holds the J script that defines a word.
5. `DOCUMENT` *Memo 10* is the documentation text associated with a word. The dictionary can process documentation text in a special LaTeX derived format.
6. `NCLASS` *Numeric 1* is the J name class of a word.
7. `TACIT` *Numeric 1* is a code that tells whether the word defined in the corresponding `SCRIPT` field is either 0 (explicit), 1 (tacit) or 2 (an arbitrary script).
8. `CREATION` *Numeric 11.4* is the time a word was first `put` (23) into the dictionary. Time is stored as a decimal Julian day number — resolution is subminute.
9. `CHANGED` *Numeric 11.4* is the time a word was last `put`, it has the same Julian day number format.
10. `JVERSION` *Character 6* records the version of J `put`'ing words.
11. `CLIENTCNT` *Numeric 6* is a count of the number of clients a word has. *A client is a word that depends on the definition of a given word.* A client will either directly or indirectly reference a word. Words with high client counts are heavily used and should be changed with care.
12. `GLOBALCNT` *Numeric 6* is a count of the number of global references made in a word.
13. `CHANGECNT` *Numeric 6* counts how often a word has been `put`.
14. `LASTSCRIPT` *Memo 10* is a one-level backup of `SCRIPT`.
15. `OWNER` *Character 3* is an "owner" code. This field can be left blank.

JGROUPS alias JG is the J group table. It contains group specific information. The primary key is `JGROUP`.

1. `JGROUP` *Character 8* is a group name. Group names are valid J names limited to eight characters. The group name is used by the dictionary to generate group script files. Case is not significant for group names.

2. EXPLAIN *Character 50* is a brief explanation of the group.
3. DOCUMENT *Memo 10* holds detailed group documentation.
4. READONLY *Logical 1* is a flag that when set to True prevents J clients from altering a group.[11]
5. GENERATE *Logical 1* controls the generation of J group scripts. GENERATE must be True to generate a group script.
6. READABLE *Logical 1* controls the generation of LaTeX *.jxs and *.jxl files. READABLE must be True to generate LaTeX for a particular group.
7. LISTCODE *Logical 1* controls the inclusion of J code in *.jxs and *.jxl files.
8. TESTON *Logical 1* controls the generation of group test case *.jgt files.
9. TESTENV *Memo 10* is an arbitrary J script that sets up a group's test environment. When group test case *.jgt files are generated TESTENV text is written to the top of the file.
10. DO *Memo 10* is an arbitrary J script that is appended to the end of generated group scripts.
11. PREAMBLE *Memo 10* holds LaTeX code that is written to the top of *.jxs and *.jxl files. PREAMBLE permits extensive customization of group listings.
12. LOCALE *Logical 1* controls the setting of the SCRIPTNAMES noun in generated group scripts. When LOCALE is True only *interface* words are added to SCRIPTNAMES. Otherwise all group members are added to SCRIPTNAMES. Excluding names permits utilities like expose in PROFILE.JS to define interfaces to groups loaded into locales.
13. CREATEGR *Numeric 11.4* is a fractional Julian group creation date.
14. OWNER *Character 3* is a group owner code.

JGRLIST alias JGX is the group list table. The group list table is a cross-reference table relating words to groups. The primary key of JGRLIST is JGROUP + NAME. Where + denotes string concatenation.

1. JGROUP *Character 8* is a group name.
2. NAME *Character 20* is a word name.
3. INTERFACE *Logical 1* if INTERFACE is True the word with NAME belongs to the locale interface of JGROUP. INTERFACE settings are used only if the LOCALE flag in table JGROUPS is True. The same word may belong to the interface of one group and be "hidden" in the locale of another.

[11] There are no such restrictions on the FoxPro server.

JUSES alias JUX is the uses or concordance table. It is a cross-reference table that relates words to "used" or "called" words. The primary key is `NAME + USES`.

1. `NAME` *Character 20* is a word name.
2. `USES` *Character 20* is a word name.

JTESTS alias JT is the main J test case table. The primary key is `TESTCASE`.

1. `TESTCASE` *Character 20* is the test case name. Test case names are valid J names limited to 20 characters.
2. `EXPLAIN` *Character 50* is a brief explanation of the test case.
3. `NAME` *Character 20* is a word name. The word name relates the test case to a particular word.
4. `EXPECTING` *Character 50* is a brief description of the test's expected outcome.
5. `ASSUMES` *Character 50* describes any assumptions made.
6. `TLEVEL` *Numeric 3* is a numeric code that orders test cases in test case script files. Cases with low `TLEVEL`'s are written first. `TLEVEL` can be used to assemble a test script that starts with easy tests and progresses to more difficult ones.
7. `CASEON` *Logical 1* controls whether a particular test case is included in a test script.
8. `READONLY` *Logical 1* is a flag that when set to `True` prevents J clients from altering a test case.
9. `CREATECASE` *Numeric 11.4* is a fractional Julian test case creation date.
10. `COMMENTS` *Memo 10* are arbitrary test case comments.
11. `CASECODE` *Memo 10* is an arbitrary J script that defines a test case.
12. `OWNER` *Character 3* is a test case owner code.

JDICTCFG is the dictionary configuration table. When `JDict` is first run this table is created in the dictionary system directory. New dictionaries are created by the `newd` (22) verb.

Each record in `JDICTCFG` corresponds to a single dictionary.

1. `JD_DEFDIC` *Character 1* is the default dictionary mark. The first record marked with a `*` character is opened as the default dictionary.
2. `JD_CODE` *Character 8* is the file name assigned to a particular dictionary's code table. The default name is `JCODE`. `JDict` cannot create nonstandard file names. To use nonstandard names rename dictionary files and enter the new names into the corresponding `JDICTCFG` record.

3. JD_GROUPS *Character 8* is the file name assigned to a dictionary's group table — default JGROUPS.
4. JD_GRLIST *Character 8* is the file name assigned to the group cross-reference table — default JGRLIST.
5. JD_USES *Character 8* is the file name assigned to the concordance or uses cross-reference table — default JUSES.
6. JD_TESTS *Character 8* is the file name assigned to the test case table — default JTESTS.
7. JD_DATDIR *Character 128* is a dictionary's data directory path. All the tables comprising a dictionary are stored in the data directory. *Warning:* Dictionary paths must end with the backslash character \.
8. JD_SYSDIR *Character 128* is the dictionary's system directory. The system directory holds all FoxPro and J source code.
9. JD_GRDIR *Character 128* is the group script directory. When the FoxPro server generates J group scripts they are written to this directory.
10. JD_JTDIR *Character 128* is the test case script directory.
11. JD_DICTID *Character 35* is the dictionary's name. This name is visible to the pickd verb.
12. JD_JVER *Character 6* is the current J version.
13. JD_DUMPJS *Character 8* is the file name assigned to the dictionary dump script.
14. JD_DUMPDIR *Character 128* is the dump directory.
15. JD_LOCDIR *Character 128* is the locale loading script directory.
16. JD_OWNER *Character 3* is an owner code. When a word, group or test case is added to the dictionary this code is attached.

JDICTKEY is the master database key table. It stores key expressions used to build FoxPro *.cdx index files. This table is so important that a backup JDICTKBK is also distributed.

Warning: only keys registered in JDICTKEY will survive a database integrity check! An integrity check deletes and rebuilds a dictionary's index files. The keys governing the rebuild are extracted from JDICTKEY.

1. TTABLE *Character 8* is an internal table name. Table names can vary depending on JDICTCFG settings. The internal name is used for critical operations like building indexes.
2. TAG *Character 12* is a FoxPro *.cdx tag name.

3. `KEYCLASS` *Numeric 1* is a code describing the key. 1 is a primary key, 2 is foreign key and 3 is an auxiliary key.
4. `INDEXPR` *Character 64* is a FoxPro index expression.
5. `TALIAS` *Character 3* is a table alias.
6. `TORD` *Numeric 1* is an internal table order used in some loops.

Appendix 4 - Restoring Dump Scripts

It's asking a lot of a programmer to surrender precious source code to a database system. What happens to my code *when* the system crashes? What if the database vendor abandons current binary file formats? How can I detect and repair corrupt database files? The nightmares are endless. I refuse to load code into a black box and I don't expect anyone else to either.

`JDict` can dump the contents of a dictionary database as three `ASCII` files:

1. `DUMPWORD.JS` is a J script that reconstructs a dictionary database when run from a J client.

2. `DUMPWORD.JGX` is a comma-delimited dump of the group list cross-reference table.

3. `DUMPWORD.JUX` is a comma-delimited dump of the concordance cross-reference table.

Because of `DUMPWORD.JS`'s size there are several precautions you should take before restoring a dictionary.

1. **Insure there is sufficient disk space to hold the fully restored database.** Plan on at least 2,000 bytes per word. J words are usually much smaller than 2,000 bytes but when you factor in documentation, cross-references, associated test cases and index files the space consumed by a word increases.

2. **Always restore words to an empty dictionary.** `newd` (22) creates an empty dictionary. It is possible to restore words to nonempty dictionaries. This is one way to merge dictionaries. Be careful not to overwrite words.

3. **Shut down all unnecessary Windows tasks.** Windows will operate more reliably and your database restore will be processed faster.

4. **Check the `Dumpfile` path noun near the top of** `DUMPWORD.JS`. If this file is moved or copied to another directory `Dumpfile` must be edited.

5. **Make sure the dictionary server is active before running a restore.**

6. **Log the output of `DUMPWORD.JS` to a file.** As the restore script runs J client messages are written to the screen. Capturing these messages is often useful.

7. **To abort a restore stop the J client.** Hitting the `ESCAPE` key in the server task will safely shutdown the server however the J script will continue to run yielding nothing but `put` and `stuff` errors.

8. **When the restore completes test database integrity.**

It's a good practice to frequently dump and backup production databases. Remember the **the DATA is more important than the BASE!**

Appendix 5 - JDICT.JS display

J client code is presented here in a readable format.[12] Words are listed alphabetically in **boldface**. Name class appears in *italics*. Tacitly defined words are shown as they are entered in J. For explicitly defined words the monad (**x.**) is above the ⇕ symbol and the dyad (**y.**) is below. If there is no dyad or monad the ∅ symbol is shown. This display format differs only in appearance from the script file JDICT.JS which is in a form suitable for directly loading into J 7.0.[13]

A note on names: Verbs in this system use cryptic local names like m99 and u99. These names were chosen for a purpose. Many client verbs test the validity of global names. If a local name clashes with a global name these tests fail. Using twisted "99" names greatly reduces clashes.

JDICT group listed on: 1994/07/27 10:43:42
J words extracted from: 7 Dictionary

CR *noun* Carriage return

```
CR =: 13 { a.
```

Codedel *noun* reserved delimiter

```
Codedel =: 2 { a.
```

Dc *noun* dictionary help locale global

```
Dc =: 0 : 4
/br     sets FoxPro browse windows
/calls  lists words, groups and tests calling a word
/del    deletes words, groups and tests
/did    dictionary identification
/disp   displays dictionary word definitions
/fld    gets one word field value from dictionary
/get    gets a J word from the dictionary
/globs  analyzes a word's global references
/grp    lists members of a dictionary group
/newd   creates a new dictionary
/notput lists J workspace words not in dictionary
/pickd  picks a dictionary
/put    puts a J word into the dictionary
/rnto   renames a dictionary word
/seek   seeks J words in dictionary
/testw  creates a dictionary test case
/uses   lists words used by a given word
/wcopy  copies a dictionary word
)
Dc =. Dc -. 10{a. [ Dc =. Dc -. 13{a.
```

[12] The server utility makelatex generated the LaTeX code used to print client words.

[13] Earliar versions of J use a different script format and cannot load JDICT.JS.

```
Dc =. |: ,: ’JDICT Dictionary Commands’ ; > <;._1 Dc
```

Diclen *noun* maximum length of dictionary names

```
Diclen =: 35
```

Jddelim *noun* maximum number of DDE characters

```
Jddelim =: 31000
```

LF *noun* new line character

```
LF =: 10 { a.
```

Namelen *noun* maximum length of server names

```
Namelen =: 20
```

Prdefs *noun* protected dictionary verbs

```
Prdefs =: <;._1 ’ def2__ get__ def2_jd_’
```

Quoterep *noun* ” character replacement

```
Quoterep =: 3 { a.
```

Semirep *noun* ; replacement

```
Semirep =: 4 { a.
```

Swap *noun* temporary swap file

```
Swap =: ’c:\temp\0$$0.jn’
```

Xdictdels *noun* special dictionary delimiters

```
Xdictdels =: Codedel , Quoterep , Semirep , LF , CR
```

apLF *verb* appends a newline to non-empty char lists

```
apLF =: ,&(10{a.)‘]@.(0&=@#)
```

br *verb* sets FoxPro browse windows

```
ser =. ’wrd’;’grp’;’test’;’uses’
msg =. 0;’!arg: opts -> 0.words 1.groups 2.tests 3.uses’
msg [ $.=.>(0 = #$ y.){’’;$.
msg [ $.=.>(y. e. i. 4){’’;$.
dmsg jddedata wd (’brw’ ddereq >y.{ser),’;’
⇕
 ∅
```

calls *verb* lists words, groups and tests calling a word

```
0 calls y.
```

```
⇕
 m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
 m99 =. 0;’!arg: options -> 0.words 1.groups 2.tests’
 m99 [ $.=.>(0 = #$ x.){’’;$.
 m99 [ $.=.>(x. e. i. 3){’’;$.
 jddecut2 jddedata wd (’who’ ddereq y.),’,’,(": x.),’;’
```

class *verb* name class of J words

```
class =: 4!:0
```

ctl *verb* character table to newline delimited list

```
ctl =: }.@(,@(1&(,"1)@(-.@(*./\."1@(=&’ ’@])))) # ,@((10{a.)&(,"1)@]))
```

cutstxb *verb* basic parse of `*.stx` character tables

```
 b =. 1 (0 10 11 14)} 17#0  NB. structure extended field starts
 stx =. b (<;.1)"1 y.       NB. cut and convert last two cols to numeric
 stx =. (".&.> _2 {."1 stx) (_2{."1 i.$stx)} stx
 ((-.&’ ’)&.> 0{"1 stx) (0{"1 i.$ stx)} stx  NB. remove blanks from fields
⇕
  ∅
```

dbox *verb* converts dictionary disp code to char table

```
dbox =: >@<;._1@((10{a.)&,@(]@-.&(13{a.)))
```

ddecode2 *verb* formats a J word for DDE transfer

```
ddecode2 =: ’0’&,@exscript‘(’1’&,@(5!:5)@<)@.ttac
```

ddepoke *verb* dictionary DDE poke command prefix

```
ddepoke =: ’ddepoke jdict do ’&,@([ , ’,’&,@(namemask@(-.&’ ’@])))
```

ddereq *verb* dictionary DDE request command prefix

```
ddereq =: ’ddereq jdict do ’&,@([ , ’,’&,@(namemask@(-.&’ ’@])))
```

def2 *verb* defines a dictionary J word in a given locale

```
 ’’ def2 y. NB. base locale default
⇕
 0;y. [ $.=.(’’;$.)>@{~ ’!’~:{.y.
 0;’!arg: not character’ [ $.=.(’’;$.)>@{~ 2=type x.
 0;’!arg: not list’ [ $.=.(’’;$.)>@{~ 1>:#$ x.
 $. =. $. }.~ 0=# x.
   0;’!arg: invalid locale name’ [ $.=.(’’;$.)>@{~ _1<class <x.
 l99 =. x. -. ’ ’     NB. target locale
 u99 =. y. i. ’=’     NB. get name
 n99 =.(’_’,l99,’_’) ,~ }. u99 {. y.
 NB. cannot define (def2) or (get) in current locale while on stack
 0;’!err: self definition’ [ $.=.>(Prdefs e.~ <n99){$.;’’
```

```
$. =. $. {~ ’012’ i. {. y.
  (0<t99);n99;t99=.class <n99 [ 0!:5 n99 , u99 }. y.      NB. explicit
  (0<t99);n99;t99=.class <n99 [ ’’". n99 , u99 }. y.      NB. tacit
  (0<t99);n99;t99=.class <n99 [ 0!:5 y. }.~ >: y. i. ’:’  NB. script
  0;’!err: invalid tacit code’
```

del *verb* deletes words, groups and test cases

```
0 del y.
⇕
m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
m99 =. 0;’!arg: opts -> 0.words 1.groups 2.tests 4.grp members 4242.calls’
m99 [ $.=.>(0 = #$ x.){’’;$.
y. =. y. -. ’ ’
$. =.  $. {~ 0 1 2 4 4242 i. x.
  dmsg jddedata wd (’del’ ddereq y.),’;’
  ’edl’ delgrp y.
  dmsg jddedata wd ’;’ ,~ ’kt’ ddereq y.
  ’egp’ delgrp y.
  dmsg jddedata wd ’;’ ,~ ’ku’ ddereq y.
  m99
```

delgrp *verb* deletes a dictionary group

```
 ∅
⇕
y. =. y. -. ’ ’
0;’!arg: invalid group name’ [ $.=.>(8 < #y.){$.;’’
dmsg jddedata wd (x. ddereq y.),’;’
```

dicreset *verb* clears temporary dictionary locale nouns

```
4!:55 ’Jdict’;’t’;’Globtxt’;’Dstruc’;’Dumpfile’
⇕
 ∅
```

did *verb* dictionary identification

```
dmsg jddedata wd ’;’ ,~ ’did’ ddereq ’’
⇕
dmsg jddedata wd ’;’ ,~ ’dsm’ ddereq ’’
```

disp *verb* displays dictionary word definitions

```
0;’!arg: empty’ [ $.=.>(0 e. #y.){$.;’’
’script’ fld y.
⇕
 ∅
```

dmsg *verb* formats a dictionary client message

```
dmsg =: ([: ’!’&~: {.) ; ]
```

downd *verb* shuts down the dictionary server

```
  NB. downs server regardless of server state.
  NB. hard codes server names "jdict" and "jbusy"
  wd ’ddereq jdict do end,,0;’
  6!:3 [ 2   NB. wait for possible server termination
  wd ’ddereq jbusy do end,,0;’
  1;’Server stopped’
⇕
 ∅
```

dy *verb* dyad code as character table

```
  dy =: >@{:@(5!:2)@<
```

eqset *verb* tests for item set equality

```
  eqset =: -. -: -.˜
```

exscript *verb* explicit word code in line text format

```
  (shead y.),LF,(apLF ctl mo y.),’:’,LF,(apLF ctl dy y.),’)’
⇕
 ∅
```

fld *verb* gets one word field value from dictionary

```
  ’document’ fld y.  NB. displays word documentation
⇕
  m99 [ $.=.>(>{.m99=.0 jnt y.){’’;$.
  0;’!arg: not character’ [ $.=.(’’;$.)>@{˜ 2=type x.
  0;’!arg: not list’ [ $.=.(’’;$.)>@{˜ 1>:#$ x.
  m99 =. jddedata wd (’fld’ ddereq y.),’,’,(x.-.’ ’),’;’
  $. =. $. {˜ 0 i.#y.  NB. if null cut reply else convert to J datatype
    jddecut2 m99
    jddeftr m99
```

get *verb* gets a J word from the dictionary

```
  def2 getcode2 y.
⇕
  x. def2 getcode2 y.
```

getcode2 *verb* gets J word code

```
  >1{m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
  ’!err: embedded locale’ [ $.=.>(’_’={: y.){$.;’’
  jddedata wd ’;’ ,˜ ’g2’ ddereq y.
⇕
 ∅
```

globnames *verb* extracts global names from J code

```
  0;’!arg: not character’ [ $.=.(’’;$.)>@{˜ 2=type y.
  0;’!arg: not table or list’ [ $.=.(’’;$.)>@{˜ 2>:#$ y.
  1;'' [ $.=.>(0 e. $parsed=.tabit y.){$.;''
  mask =. masknb parsed
  locs =. '' [ gbls =. ''
  $.=.>(1 e. '(*)=' E. , parsed){fi;$.  NB. if possible opaques search
    ('locs';'gbls') =. mask opaqnames parsed
    olap =. locs -. locs -. gbls        NB. intersection
    0;'!err: confused declarations ->';olap [ $.=.>(0<# olap){$.;''
  fi)
  mask =. 0 [ parsed =. parsed jnb~ -. mask NB. blank comments & clear mask
  parsed =. parsed #~ parsed +./ . ~: ' '   NB. remove blank rows
  parsed =. (;: :: 0:)&.> <"1 parsed  NB. parse code
  0;'!err: word syntax' [ $.=.>(parsed e.~ <0){$.;''
  labels =. 0 1{"1 (2>.}.$>parsed){."1 >parsed  NB. insure 2 columns
  labels =. (')'e.&> 1{"1 labels)#0{"1 labels
  parsed =. labels -.~ ; parsed  NB. remove labels
  parsed =. parsed -. parsed #~ 1|.parsed = <'=.'  NB. remove =. assignments
  parsed =. (parsed #~ _1 ~: class parsed)-.locs,'$.';'x.';'y.';'$:'
  parsed =. ~. gbls , parsed
  1;(/:>parsed){parsed  NB. return unique sorted globals
⇕
  ∅
```

globs *verb* analyzes a word's global references

```
  0 globs y.
⇕
  0;'!arg: not character' [ $.=.('';$.)>@{~ 2=type y.
  0;'!arg: not a list' [ $.=.('';$.)>@{~ 1>:#$ y.
  0;'!arg: null name' [ $.=.('';$.)>@{~ 0~:# y. =. y. -. ' '
  m99 =. 0;'!arg: opts -> 0.locale 1.dictionary 2.synch'
  m99 [ $.=.>(0 = #$ x.){'';$.
  m99 [ $.=.>(x. e. i. 3){'';$.
  $.=.>(x. e. 1 2){(wks -. dic);dic
    wks) c99 =. class <y. =. (]@,&'__')`]@.('_'&=@{:) y.
      0;'!arg: noun name' [ $.=.('';$.)>@{~ 2~:c99
      0;'!arg: not in locale or bad name' [ $.=.('';$.)>@{~ 3 4 5 e.~ c99
      jddecode y.  NB. sets Globtxt from base local word
      globnames dbox Globtxt
    dic) Globtxt =: disp y.  NB. read dictionary text without defining
      Globtxt [ $.=.('';$.)>@{~ 2=type Globtxt  NB. text or boxed error
  m99 =. globnames dbox Globtxt [ $. =. >(x. = 2){'';$.
  0;'!err: name analysis' [ $.=.>(>{.m99){'';$.
  NB. compare with dictionary names and update if necessary
  c99 [ $.=.>(>0{c99 =. uses y.){'';$.
  1;'Concordance matches' [ $.=.>(m99 eqset c99){$.;''
  y. globswap m99
```

globswap *verb* updates global names via swap file

```
  ∅
⇕
  y. =. qtnames  }. y.  NB. list of globals
  m99 =. y. (1!:2) :: 0: <Swap
  0;’!err: unable to write swap’ [ $.=.>(0 -: m99){$.;’’
  m99 =. jddedata wd (’con’ ddereq x.),’,’,Swap,’;’  NB. >0{x. is name
  $.=. $. }.~ ’!’ = {. m99
    1;m99 [ $.=.’’
    0;(m99 {.~ >: u99) ; <;._1 }. m99 }.~ u99=.m99 i. ’>’  NB. missing
```

grp *verb* lists members of a dictionary group

```
  m99 [ $.=.>(>{.m99=.0 jnt y.){’’;$.
  jddecut2 jddedata wd ’;’ ,~ ’grp’ ddereq y.
⇳
  m99=.0;’!arg: invalid group name’ [ $.=.>({.jnt y.){’’;$.
  m99 [ $.=.>(8 < #y.){$.;’’ [ y. =. y. -. ’ ’
  $.=.>(0 = #x.){(group -. crgrp);crgrp
    group)0;’!arg: not list’ [ $.=.(’’;$.)>@{~ 1>:#$ x.
      0;’!arg: invalid names’ [ $.=.($.;’’)>@{~ _1 e. class x.
      0;’!arg: names to long for server’ [ $.=.($.;’’)>@{~ Namelen< >./ #&> x.
      m99 =. (qtnames x.) (1!:2) :: 0: <Swap
      0;’!err: unable to write swap’ [ $.=.>(0 -: m99){$.;’’
      m99 =. jddedata wd (’gmk’ ddereq y.),’,’,Swap,’;’
      $.=. $. }.~ ’!’ = {. m99
        1;m99 [ $.=.’’
        0;m99 [ $.=.>(’>’ e. m99){’’;$.
        0;(m99 {.~ >: u99) ; <;._1 }. m99 }.~ u99=.m99 i. ’>’
    crgrp) dmsg jddedata wd ’;’ ,~ ’cgp’ ddereq y.
```

ht *verb* head and tail

```
  ht =: {. , {:
```

isupper *verb* bit mask of upper case characters

```
  isupper =: (65&<:*.<:&90)@(a.&i.) NB. ASCII collating sequence
```

jddecode *verb* dictionary DDE code

```
  $. =. $. }.~ ’1’ = {. s99 =. ddecode2 y.
    s99 [ Globtxt =: }. s99 -. LF
    s99 [ Globtxt =: }. s99
⇳
   ∅
```

jddecut2 *verb* cuts DDE server data

```
  jddecut2 =: ’!’&~:@{. ; <;._1‘]@.(’!’&=@(0&{) ::0:)
```

jddedata *verb* extracts *data from DDE

```
  jddedata =: >@(’*data’&wget ::(’!err: no server data’&[))
```

jddeftr *verb* translates DDE text to J datatypes

```
  jddeftr =: 0&;@<‘}.‘(".@}.)‘(".@}.)‘(".@}.)‘}.‘]@.(’!CNLDM’&i.@{.)
```

jdstruc *verb* load dictionary table structures

```
 y. =. y. -. ’ ’
 str =. i. 0 2 [ err =. 0;’!err: structure load’
 cnt =. 0 [ tbls =. _2 <\ ’JBJGJT’    NB. stuff’able table aliases
 $. =. od ,˜ , (#tbls) $ ,: $. -. od  NB. iterate over tables
   stx =. jddedata wd (’stx’ ddereq >cnt{tbls),’,’,y.,’;’
   err [ $.=.>(’!’ = {. stx){$.;’’
   str =. str , (cnt{tbls) , <swapstx stx
   cnt =. >: cnt
 od) (|: str) , ’NAME’;’JGROUP’;’TESTCASE’  NB. attach primary key names
⇕
  ∅
```

jnb *verb* blanks out J code leaving only comments

```
 y. jnb˜ masknb y.
⇕
 (x. * >: i. $ x.){’ ’,,y.
```

jnt *verb* J name test

```
 1 jnt y.
⇕
 0;’!arg: not character’ [ $.=.(’’;$.)>@{˜ 2=type y.
 0;’!arg: not a list’ [ $.=.(’’;$.)>@{˜ 1>:#$ y.
 1;’!ok: null name’ [ $.=.(’’;$.)>@{˜ -.(0 = #y.) *. x. -: 0
 0;’!arg: invalid name’[ $.=.(’’;$.)>@{˜ _1 ˜: class <y.
 0;’!arg: name to long for server’ [ $.=.(’’;$.)>@{˜ Namelen >: #y.
 1;’!ok: valid name’
```

masknb *verb* bit mask of unquoted comment starts

```
 ’NB.’ masknb y.
⇕
 c =. ($y.)$x. E. ,y.
 +./\"1 c > ˜:/\"1 y. e. ’’’’
```

mo *verb* monad code as character table

```
 mo =: >@{.@(5!:2)@<
```

namemask *verb* name with case mask

```
 namemask =: ] , ’,’&,@(1&":@(isupper@]))
```

nctac *verb* formats name class and tacit bit

```
  ∅
⇕
 ’,’,(":x.),’,’,{. y.
```

newd *verb* creates a new dictionary

```
  ∅
 ⇕
  m96 =. 0;'!arg: not character' [ $.=.('';$.)>@{~ 2=type x.
  m97 =. 0;'!arg: not list' [ $.=.('';$.)>@{~ 1>:#$ x.
  m98 =. 0;'!err: delimiters' [ $.=.($.;'')>@{~ +./x. e.~ Xdictdels
  m99 =. 0;'!arg: null' [ $.=.('';$.)>@{~ 0~:# x. =. x. -. ' '
  0;'!arg: bad path' [ $.=.>((':' e. x.) *. '\'={:x.){'';$.
  m96 [ $.=.('';$.)>@{~ 2=type y.
  m97 [ $.=.('';$.)>@{~ 1>:#$ y.
  m98 [ $.=.($.;'')>@{~ +./y. e.~ Xdictdels
  m99 [ $.=.('';$.)>@{~ 0~:# y. =. ,y.
  0;'!arg: dictionary name to long' [ $.=.('';$.)>@{~ Diclen >: #y.
  y. =. Codedel ((' '=y.)#i.$y.)} y.  NB. hide blanks
  dmsg jddedata wd ';' ,~ x. ,~ ',' ,~  'new' ddereq y.
```

notput *verb* lists J workspace words not in dictionary

```
  0;'!arg: not list' [ $.=.('';$.)>@{~ 1>:#$ y.
  u99 =. 0;'!arg: invalid names' [ $.=.($.;'')>@{~ _1 e. class y.
  u99 [ $.=.($.;'')>@{~ 0=# y.
  0;'!arg: names to long for server' [ $.=.($.;'')>@{~ Namelen < >./ #&> y.
  u99 =. (qtnames y.) (1!:2) :: 0: <Swap
  0;'!err: unable to write swap' [ $.=.>(0 -: u99){$.;''
  jddecut2 jddedata wd ('np' ddereq ''),',',Swap,';'
 ⇕
  ∅
```

odq *verb* query dictionary owner public variable

```
  jddedata wd ';' ,~ 'oq' ddereq ''
 ⇕
  ∅
```

ods *verb* sets dictionary owner public variable

```
  0;'!arg: not a char list' [ $.=.>((1>:#$ y.) *. 2=type y.){'';$.
  msg =. wd ';' ,~ 'os' ddereq ,y.
  ('!'~:{.msg) ; msg =. jddedata msg
 ⇕
  ∅
```

opaqnames *verb* extracts declared opaque names

```
  y. opaqnames~ masknb y. NB. compute mask
 ⇕
  b  =. +./"1 x.          NB. use supplied mask
  x. =. b # x. [ y. =. b # y.
  y. =. x. jnb y.         NB. search only comment text
  '';'' [ $.=.>(+./ '(*)=' E. , y.){'';$.  NB. result if no declarations
  locals =. (,y.) #~ , '(*)=.' masknb y.
  locals =.  ~. <;._1 ' ',locals #~ -. '  ' E. locals
  locals =. < locals #~ _1 < class locals
```

```
    globals =. (,y.) #˜ , ’(*)=:’ masknb y.
    globals =. ˜. <;._1 ’ ’,globals #˜ -. ’  ’ E. globals
    globals =. < globals #˜ _1 < class globals
    locals,globals
```

pickd *verb* picks a dictionary

```
    0;’!arg: not character’ [ $.=.(’’;$.)>@{˜ 2=type y.
    0;’!arg: not a list’ [ $.=.(’’;$.)>@{˜ 1>:#$ y.
    NB. replace blanks in dictionary names for DDE
    4!:55 <’Jdict’  NB. force reset on put’s
    u =. Codedel ((’ ’=y.)#i.$y.)} y.
    $.=. $. }.˜ 0 = # y.
      dmsg jddedata wd ’;’ ,˜ ’pck’ ddereq u [ $.=.’’
      u =. jddedata wd ’;’ ,˜ ’pdq’ ddereq ’’
      (’!’ ˜: {.u);<;._1 (Codedel={.u) }. Codedel,u
  ⇕
    ∅
```

put *verb* puts a J word into the dictionary

```
    m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
    cl99 =. class <n99,’__’ [ n99 =. y.-.’ ’
    0;’!arg: cannot put given name’ [ $.=.($.;’’)>@{˜ ’:’ e. n99
    0;’!arg: not in base locale’ [ $.=.(’’;$.)>@{˜ 2 3 4 5 e.˜ cl99
    0;’!err: to large for DDE’ [ $.=.(’’;$.)>@{˜ Jddelim>#  c99=.jddecode n99,’__’
    $.=. $. }.˜ ’1’ ˜: {. c99
      0;’!err: delimiters’ [ $.=.($.;’’)>@{˜ +./c99 e.˜ Xdictdels
    c99 =. (’p2’ ddepoke y.) , (cl99 nctac {.c99) , putfmt }. c99
    $.=.(2=class <’Jdict’) }. $.  NB. set active dictionary if necessary
      Jdict =: >1{did ’’  NB. Jdict locale global
    m99 =. (0{"1 wd c99) -.@e.˜ <’*error’
    m99 ; m99 { ’!err: not put’;Jdict
  ⇕
    u99 [ $.=.>(>{.u99=.put y.){’’;$.
    m99 =. dbox Globtxt [ y. =. y. -. ’ ’  NB. Globtxt set by put
    0;’!err: name analysis’ [ $.=.>(>{.m99 =. globnames m99){’’;$.
    u99 [ $.=.>(0 = #>1{m99){$.;’’  NB. no globals
    y. globswap m99
```

putfmt *verb* replaces chars for DDE poke

```
    c =. Semirep ((’;’=y.)#i.$y.)} y.
    qq Quoterep ((’"’=c)#i.$c)} c
  ⇕
    ∅
```

qq *verb* quotes with ” and ; for DDE

```
    qq =: ’ "’&,@(,&’";’@[)
```

qstruc *verb* query dictionary table structures

```
 'JB' qstruc y.
⇕
 x. =. toupper x. -. ' '
 y. =. toupper y. -. ' '
 ti =. (0{Dstruc) i. <x.
 0;'!err: no table' [ $.=.>(ti = {:$ Dstruc){$.;''
 tbl =. >ti{1{ Dstruc
 fdi =. (0{"1 tbl) i. <y.
 0;'!err: no field' [ $.=.>(fdi = {.$ tbl){$.;''
 x. ; fdi { tbl
```

qtnames *verb* boxed name list to comma delimited char list

```
qtnames =: ;@('"'&,@(,&(34 10{a.))&.>@])
```

restxref *verb* issues a DDE command to restore xref files

```
 '!err: missing dump noun' [ $. = >(0 = 4!:0 <'Dumpfile'){$.;''
 dmsg jddedata wd ';' ,~ ('rx' ddereq ''),',',Dumpfile -. ' '
⇕
  ∅
```

rnto *verb* renames a dictionary word

```
  ∅
⇕
 m99 [ $.=.>(>{.m99=.jnt y.){'';$.
 m99 [ $.=.>(>{.m99=.jnt x.){'';$.
 m99 =. wd ('ren' ddereq x.),',',(namemask y.-.' '),';'
 ('!'~:{.m99) ; m99 =. jddedata m99
```

seek *verb* seeks J words in dictionary

```
 _1 seek y.
⇕
 m99 [ $.=.>(>{.m99=.0 jnt y.){'';$.
 m99 =. 0;'!arg: class -> _1.first 2.noun 3.verb 4.adv 5.conj 6.all'
 m99 [ $.=.>(0 = #$ x.){'';$.
 m99 [ $.=.>(x. e. _1 2 3 4 5 6){'';$.
 x. =. 0 >. x. [ m99 =. > (',all';',one') {~ _1 = x.
 jddecut2 jddedata wd ('seek' ddereq  y.),m99,',',(' ' -.~ ": x.),';'
```

service *verb* sets DDE poke and request commands

```
 u99 =. 'ddepoke';'ddereq'
 m99 =. 5!:5 u99
 p99 =. (0 , {:$ m99) +"0 1 (i. #y.) +"1 0  m99 i."1 {. y.
 m99 =. (>u99) ,"1 ' =: ' ,"1 y. p99} m99
 m99 =. (m99 , '1 ') : ''
 m99 0
⇕
  ∅
```

setservice *verb* sets the client DDE service

```
 ser =. y. { 'jdict' ,: 'jbusy'
 NB. next command will fail as it breaks the
 NB. DDE channel before a request can be sent
 wd ';' ,~ 'd9' ddereq toupper ser
 service ser  NB. re-define client service commands
 NB. request dictionary name using new service
 did 0
⇕
 ∅
```

shead *verb* explict definition `0 : n` header

```
shead =: '0 : '&,@(":@(_3&+@(class@<)))
```

stuff *verb* inserts data into dictionary table fields

```
 ∅
⇕
 ('key';'tbl';'fie') =. y.     NB. opaque locals (*)=. key tbl fie
 $.=.(2=class <'Jdict') }. $.  NB. set dictionary name
   Jdict =: >1{did ''
 $.=.(2=class <'Dstruc') }. $. NB. load table structures
   Dstruc =: jdstruc Swap
 mu [ $.=.>(0 -: >{.mu =.tbl qstruc fie){$.;''  NB. field structure
 kn =. (2{Dstruc) {~ (0 { Dstruc) i. <toupper tbl   NB. key field name
 0;'!err: to large for DDE'[$.=.>(Jddelim<#mu=.key stuffpoke mu,x.;kn){$.;''
 msg =. (0{"1 wd mu) -.@e.~ <'*error'
 msg ; msg { '!err: not stuffed';Jdict
```

stuffmt *verb* formats non-memo data for stuff

```
stuffmt =: ('st'"_ ddepoke [) , (_1"_ |. [: ; [: ,&','&.> 0 1 2 6"_ { ])
```

stuffpoke *verb* stuff poke for non-memo data

```
stuffpoke =: ([ stuffmt ]) , ([: putfmt [: > 5: { ])
```

swapstx *verb* reads and parses `*.stx` swap file

```
 stx =. (1!:1) :: 0: <y.           NB. read swap file
 0;'!err: unable to read swap' [ $.=.>(0 -: stx){$.;''
 stx =. stx }.~ - ({:stx) = 26{a.  NB. remove DOS eof if present
 stx =. stx -. CR
 stx =. ><;._1 LF,stx
 cutstxb stx #~ -. stx *./ . = ' ' NB. remove blank lines and parse
⇕
 ∅
```

tabit *verb* promotes atoms and lists to tables

```
tabit =: ]`,:@.(1&>:@(#@$))^:2
```

testw *verb* creates a dictionary test case

```
  ∅
 ⇕
 m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
 m99 [ $.=.>(>{.m99=.jnt x.){’’;$.
 m99 =. wd (’ctc’ ddereq x.),’,’,(namemask y.-.’ ’),’;’
 (’!’~:{.m99) ; m99 =. jddedata m99
```

toupper *verb* to upper case

```
 (y.i.~’abcdefghijklmnopqrstuvwxyz’,a.){’ABCDEFGHIJKLMNOPQRSTUVWXYZ’,a.
⇕
  ∅
```

ttac *verb* tests for tacit

```
 1 [ $.=.>(2=class <y.){$.;’’
 1 [ $.=.>(3 ~: #jc =. (5!:2) <y.){$.;’’
 1 [ $.=.>(-. (,’:’) -: ,>1{jc){$.;’’
 1 [ $.=.>(32 e. type&> jc=. ht jc){$.;’’
 1 [ $.=.>(1 e. (#@,)&> jc){$.;’’
 0  NB. explicit
⇕
  ∅
```

type *verb* data type of J word

```
 type =: 3!:0
```

uses *verb* lists words used by a given word

```
 ’*one’ uses y.
⇕
 m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
 x. =. }. > (’*all’;’*one’) {~ ’*one’ -: x.
 m99 =. jddedata wd (x. ddereq y.),’;’
 $. =. $. {~ 0 i. #m99
   1;’’
   jddecut2 m99
```

wcopy *verb* copies a dictionary word

```
  ∅
 ⇕
 m99 [ $.=.>(>{.m99=.jnt y.){’’;$.
 m99 [ $.=.>(>{.m99=.jnt x.){’’;$.
 m99 =. wd (’wjc’ ddereq x.),’,’,(namemask y.-.’ ’),’;’
 (’!’~:{.m99) ; m99 =. jddedata m99
```

wd *verb* window driver

```
 wd =: 11!:0
```

wget *verb* extracts from wd result

```
    wget =: ({."1@] i. <@[) {:@{ ]
```

Appendix 6 - PROFILE.JS

The following J profile.js establishes the jd locale. A key part of the definition of locale jd is the application of expose. expose uses the SCRIPTNAMES noun in locale jd to define a set of base locale “alias” verbs.

```
NB. Dictionary J profile:  (July 1994)

GrPath =: ’C:\JDICT\mine\gr\’     NB. group path noun
JdPath =: ’C:\JDICT\’             NB. dictionary system directory

NB. defines base locale interface aliases
expose =: 0 : 0
0;’!arg: not character’ [ $.=.(’’;$.)>@{˜ 2 = 3!:0 y.
0;’!arg: not list’ [ $.=.(’’;$.)>@{˜ 1>:#$ y.
lfx99 =. ’_’ ,˜ ’_’ , y. -. ’ ’
w99 =. ’i.0’  ". ’SCRIPTNAMES’ , lfx99
0;’!err: no interface’ [ $.=.>(0 = #w99){$.;’’
w99 =. <;._1 w99
lfx99 =. w99 ,&.> <lfx99
0;’!err: missing interface words’ [ $.=.>(’’;$.){˜ *./ 0 < 4!:0 lfx99
(<’i.0’) ".&.> w99 ,&.> (<’ =: ’) ,&.> lfx99
$. =. $. }.˜ *./ 0 < 4!:0 w99
  0;’!err: missing base locale words’ [ $.=.’’
  1;w99
:
)

NB. gets a J script
getsc =: 0 : 0
$. =. $. }.˜ 2=(4!:0) < ’GrPath’
  GrPath =: ’C:\JDICT\mine\gr\’ NB. default path
GrPath getsc y.
:
SCRIPTNAMES =: i.0 0
0!:3 <x.,y.,’.js’
’__’;SCRIPTNAMES
)

SCRIPTNAMES=:’ GrPath JdPath expose getsc’

NB. define FoxPro dictionary locale (jd)
in_jd_ =. 0!:3
in_jd_ <JdPath,’jdict.js’
NB. define dictionary interface
expose ’jd’
```

Appendix 7 - J8 Support

J has been evolving and changing from its beginning. J8 is more than an incremental change or refinement to the J language. In many ways it is a new language.

J language changes have major consequences for `JDict` .

1. **Old J scripts will not run!**

   `JDICT.JS` cannot be loaded into J8 because `LABELS)` and the suite `$.` have been dropped. May they rest in peace. J8 introduces standard *control structures*: see `c:\jdict\j8sup\j8dict.js` It is interesting to compare the old and new client scripts. The new script is much easier to read.

   Note: `J8DICT.JS` can be safely loaded into the `z` locale. Inspect the files in `c:\jdict\j8sup` to see how this can be done.

2. **J name class codes have changed.**

   To handle name class code changes compiler directives were inserted into `JDICT.PRG` to generate a J8 version of the server.

   To use `c:\jdict\j8sup\j8dict.exe` copy `J8DICT.EXE` to the `c:\jdict` directory. Use the file manager to drag `J8DICT.EXE` into a program manager group.

   Start `J8DICT.EXE` and use the corresponding client `J8DICT.JS` to create J8 dictionaries.

Index