

SHELL and SQL Extensions

1.1 shell (abbreviation: none)

1.1.1 shell/g

1.1.2 shell/p

The **shell** command passes the remainder of the line to a shell for execution (**sh** in Linux). Shell output will appear on **stdout**. The command sets **\$test** to false if the *fork()* fails, true otherwise.

This command is not presently available in the DOS version.

The **shell/p** form passes the remainder of the line to a shell for execution but opens a pipe from the shell to Mumps/II unit number 6. All output from the shell is directed to unit number 6 and can be read with any of the input commands or functions in association with the **use** command.

The **shell/g** form passes the remainder of the line to a shell for execution (**sh** in Linux) and opens a pipe from the Mumps/II program to the shell as Mumps/II unit number 6. Data written to this unit becomes **stdin** to the shell. Remember to **close** unit number 6 to signal end-of-file to the shell.

For example:

```
1 shell sort dictionary.tmp | uniq -c | sort -nr > dictionary.s
```

The Linux **sh** shell will do the following:

1. File *dictionary.tmp*, a collection of words, will be sorted by **sort** and the output piped to **uniq**
2. **uniq** counts duplicate entries and pipes its output consisting of a count and a word to **sort**
3. **sort** sorts the result numerically by number of duplicates in reverse order and writes its output to *dictionary.s*.

```
1 shell/p sort dictionary.tmp | uniq -c | sort -nr
2 open 1:"dictionary.s,new"
3 for do
4 . use 6
5 . read line
6 . if '$test break
7 . use 1
8 . write line,!
9 close 1
```

The above does the same but the output will be presented to Mumps/II unit 6 which reads and writes the result to the file named *dictionary.s*.

1.2 **sql** (abbreviation: none)

1.2.1 **sql/c**

1.2.2 **sql/f**

1.2.3 **sql/o=exp**

1.2.4 **sql/t=table,size**

The **sql** command is used to pass commands to the SQL relational database server.

By default, this version of Mumps/II uses a native global array handler which is very fast but not fault tolerant.

PostgreSQL, on the other hand, is a widely used, thoroughly fault tolerant and completely free, open-source relational database management package which in many tests outperforms its commercial counterparts. PostgreSQL has the advantages of being fault-tolerant, multi-user, network accessible, accessible from C/C++, Java, .Net, Perl, Python, Ruby, Tcl, and ODBC, among others.

As a configuration option, global arrays from Mumps/II may be stored in a PostgreSQL database. Details are described in the distribution package. When globals are stored in PostgreSQL, they are accessible not only to local and remote Mumps/II applications, but also SQL queries processed without the use of Mumps/II code.

Mumps/II can also be configured for native btree storage of globals along with **SQL** command access to PostgreSQL.

The basic **sql** command passes the remainder of the line to the PostgreSQL backend. If successful, **\$test** is set to true, false otherwise.

Mumps/II expressions may be embedded in the SQL command in the same manner as done in the **html** and **shell** commands (see above).

The **sql/c** command disconnects from the SQL server. This is done automatically when a program terminates if PostgreSQL is being used to store the global array database. If not, you need to execute this command before the program terminates or the server will generate a warning message. No other commands may appear on this line.

The form: **sql/f** instructs the PostgreSQL backend to clear all contents of the Mumps/II global array database. No other commands may appear on this line.

The form **sql/o=exp** is designed for SQL commands passed to the backend which result in tuples, for example, as a result of the **select** command. The value of **exp** is taken as the name of a file (may be prefixed with directory information) into which the tuples will be written. Items in a tuple written to this file will be separated from one another by the <tab> character.

For example:

```
1  #!/usr/bin/mumps
2
3  sql/f
```

```

4
5  set ^lab(1111,$zd1,"hct",44)=""
6  set ^lab(2222,$zd1,"hct",45)=""
7  set ^lab(3333,$zd1,"hct",46)=""
8  set ^lab(4444,$zd1,"hct",47)=""
9  set ^lab(5555,$zd1,"hct",48)=""
10
11 set ^bp(1111,$zd1,128,70)=""
12 set ^bp(2222,$zd1,127,71)=""
13 set ^bp(3333,$zd1,126,72)=""
14 set ^bp(4444,$zd1,125,73)=""
15 set ^bp(5555,$zd1,124,74)=""
16
17 set ^prob(1111,$zd1,"123.45")=""
18 set ^prob(2222,$zd1,"223.45")=""
19 set ^prob(3333,$zd1,"323.45")=""
20 set ^prob(4444,$zd1,"423.45")=""
21 set ^prob(5555,$zd1,"523.45")=""
22
23 sql/o=$j_".tmp" select * from mumps where a1='3333';
24   open 1:$j_".tmp,old"
25     if '$test write "file not found",! halt
26
27     for do
28       . use 1
29       . read a
30       . if '$test break
31       . use 5
32       . for i=1:1 do
33         .. set b=$piece(a,"      ",i)
34         .. if b="" break
35         .. write b," ... "
36         . write !

```

output:

```

lab ... 3333 ... 1213545964 ... hct ... 46 ...
bp ... 3333 ... 1213545964 ... 126 ... 72 ...
prob ... 3333 ... 1213545964 ... 323.45 ...

```

The command on line 3 empties the global array database of all contents. Lines 5 through 21 populate the Mumps/II global arrays with a simple set of example clinical data. Line 23 passes a SQL query to the database server as specifies that the output be written to a file whose name is composed of the current process's PID (**\$job**) with the *.tmp* extension.

The resulting file name is *547D.tmp* (the Linux PID when this example was run). It is read and printed by lines 27 through 36. Note that the quoted value in the **\$piece()** function on line 33 is a `<tab>` character. Be careful that your editor does not convert `<tab>`s to blanks (*set noexpandtab* in *vi*).

For PostgreSQL purposes, the database used is *mumps* and the table in which all global arrays are stored is also named *mumps*.

When Mumps/II globals are mapped to relational tables, the global array name has the columns named *gbl* and the remaining columns are named *a1*, *a2*, *a3*, ... up to *a10*.

If a data value is stored for a global array node (none are stored in the example above), it will be found in the column with the name *a11*.

Consequently, global arrays are restricted to ten levels of indexing but this is a parameter which can be changed as needed.

The **sql/t=table,size** command is used to switch to a different PostgreSQL table. This command only applies if PostgreSQL is being used as the backend storage facility for the global arrays. Both **table** and **size** must be valid Mumps/II expressions. **table** is the name of the PostgreSQL table which will be used for global array references and **size** gives the number of columns, exclusive of the global array name column (always column one).

For example, assuming the following SQL data definitions exist and that the tables are populated:

```
create table problems (ptid text, icd text, problem text,
    onset text, resolved text, dxphys text);

create table pname (ptid text, namefirst text, namelast text,
    namemiddle text, nameprefix text, namesuffix text);
```

The following Mumps/II code uses views constructed from the above:

```
1  #!/usr/bin/mumps
2
3  sql create temp view nv (gbl,a1,a2,a3,a4) as select
    text 'nv', ptid, namelast, namefirst, namemiddle from pname;
4
5  sql/t="nv",4
6
7  for ptid="":$order(^nv(ptid)):" do
8  . for nlast="":$order(^nv(ptid,nlast)):" do
9  .. for nfirst="":$order(^nv(ptid,nlast,nfirst)):" do
10 ... for nmid="":$order(^nv(ptid,nlast,nfirst,nmid)):" do
11 .... write ptid," ",nlast," ",nfirst," ",nmid,!
12
13 sql create temp view pv (gbl,a1,a2,a3) as select text 'pv',
    ptid, icd, problem from problems;
14
15 sql/t="pv",3
16
17 for ptid="":$order(^pv(ptid)):" do
18 . for icd="":$order(^pv(ptid,icd)):" do
19 .. for prob="":$order(^pv(ptid,icd,prob)):" do
20 ... write ptid," ",icd," ",prob,!
21 ... write "val=",^pv(ptid,icd,prob),!
22
23 sql/t="mumps",11
24
25 halt
```

example output (all data fictitious):

```
1001 Jones John James
1002 Smith Charles James
1003 Smith Sara Mary
1004 Jones Jane Sharon
1005 Adams William Michael
1001 265 THIAMINE/NIACIN DEFIC
val=THIAMINE/NIACIN DEFIC
1001 286 COAGULATION DEFECTS
```

```

val=COAGULATION DEFECTS
1001 311 DEPRESSIVE DISORDER NEC
val=DEPRESSIVE DISORDER NEC
1001 345 EPILEPSY
val=EPILEPSY
1001 373 INFLAMMATION OF EYELIDS
val=INFLAMMATION OF EYELIDS
.
.
.

```

On line 3 (which is one line but has been split onto two due to space limitations) an SQL command is executed that creates a five columned temporary view named **nv** consisting of four columns from the table **ptname**. Note that the first column of the view contains the constant value **nv** (due to the *text 'nv'* following the *select*). The first column will be the global array name used for this table in the Mumps/II program. Note that the names of the columns in the created view are *gbl*, *a1*, *a2*, *a3* and *a4*. These are required.

On line 5 the Mumps/II environment is instructed to switch to the **nv** table for future global array references and the number of columns (*a1*, *a2*, ...) is given as 4.

Lines 7 through 11 access the global array *nv* in the usual manner.

On line 13 (also shown split) a new temporary view is created named **pv**. Its name and size are given in line 15 and it is accessed on lines 17 through 21. The Mumps/II environment is then returned to the default table **mumps**.

Note: the **mumps** table is always size 11. There are ten columns which may be used as indices and one column (*a11*) which contains stored data, if any.

Note line 21. The value printed is that of the final column. If a value is stored, it will be stored into the final column but since these are temporary views, they may not be modified.

PostgreSQL has many tuning parameter that are not covered here. See the PostgreSQL documentation. However, in cases where speed is important, a series of inserts into the database that are done as one transaction is faster than individual transactions (the default. For example:

```
1  #!/usr/bin/mumps
2
3  sql/f
4
5  set k=0
6
7  sql SET LOCAL synchronous_commit TO OFF;
8  sql begin;
9
10 for i=1:1:100 do
11 . s k=k+1
12 . s ^a(i)=k
13 . for j=1:1:100 do
14 .. set k=k+1
15 .. set ^a(i,j)=k
16 .. for m=1:1:10 do
17 ... s k=k+1
18 ... s ^a(i,j,m)=k
19 sql commit
20 halt
```

One line 3 the database is cleared and initialized. On line 7 the SQL command disables the server's waits for the transaction's records to be flushed to permanent storage before returning a success indication to the client. This causes the inserts to proceed very much faster but at some risk of data loss (but not data corruption) should the system fail during updates.

Line 8 starts a transaction and line 19 commits the transaction which finalizes the values inserted in the intervening lines.