Programming VistA

Setting up a program

- Save in user files /r directory (in GT.M)
- Filename must be 8 characters or less, UPPERCASE, with '.m' extension. (Mumps itself will allow lower-case filenames, but to call a function from a menu option VistA only works with uppercase.)
- First 3 characters should be your namespace or ZZZ (to avoid being overwritten by other programmers.)
- Labels (aka 'tags') go at the left margin. Other lines should be indented with spaces.

Overview of M/Mumps key commands

WRITE

writes out to the current i/o channel
example:

```
write "Hello world!",!   ;<----------  the "!" is a linefeed
```

or

```
write "Hello there, ",NameVar,!
```

NEW, scope, and **namespacing**

The scope of ALL variables in M is global scope.

Example:

```
MyProc
        new i
        for  i=1:1:10 do
        . write i, $$MyFn,!
        quit

MyFn
        for i=10:-1:1 do
        . write i
        quit
```

This will go into a endless loop, because changing i in MyFn will affect value in MyProc
Solution: add a "new i" in MyFn, and properly namespace variables.

Variables *will*go out of scope when leaving a DO block where they have been NEW'd

Example

```
        new i
        set i=1
        do
        . new i    ;<--- establishes a new scope for i
        . set i=5
        . write i   ;<--- outputs 5
        write i  ;<------- outputs 1
```

Pearls:

- If you NEW your variables, you can be sure you are not cobbering (overwriting) someone else's variabls. BUT anytime you call someone else's code (i.e. Fileman calls), *that* code could KILL or clobber (overwrite) your variable. Solution? **Namespacing your variables.**

- You only have to namespace the variables that you care about

KILL

KILL will remove a variable from symbol table, within the current scope.
SET and KILL used to be only tools for managing variables.
In my opinion, NEW has made the use of KILL less fundamental.

Example

```
new i
set i=1
do
. new i    ;<--- establishes a new scope for i
. set i=5
. write i,!
. kill i      ;<---- kills new'd variable

write i  ;<------- outputs 1, wasn't killed.
```

SET

assigns a value: if variable didn't exist before, it is created. Scope is global
e.g.'s

```
set MyVar=1
set MyVar("index1",.05,"someOtherIndex")="myValue"
```

Multiple values can be set at once.

```
set MyVar1,MyVar2=2
```

Comparitors

>	greater than
<	less than
=	equals
'=	not equals
(>=)	(use '< instead)
(<=)	(use '> instead)
&	and (boolean only)
!	or (boolean)

Logic evalation

Left-to-right evaluation. I don't think there is any short circuit of logic (?)

```
if (a=1)&(b=2)!(c>5) ...
```

Another construction:

```
if (a-1),(b=2)  <--- this does allow short circuiting of logic.
```

"One-liners"

Much legacy code treats one line of code like a "paragraph" of code in other languages
This was to speed processing due to interpreted nature of Mumps.
Shortening command names to their first letters makes more sense when doing this, as it allows more commands on one line.

```
e.g. W "Starting" F TMGI=1:1:10 W TMGI,$$FN^TMGXYZ(TMGI),!
```

vs.

```
write “Starting”
for TMGI=1:1:10 do
. write TMGI
. write $$FN^TMGXYZ(TMGI),!
```

DO and QUIT

Examples

```
do MyFunct(i)
```

or

```
do  write “Step 4”,!
. write “Step 1”,!
. write “Step 2”,!
. if i>5 quit        ;<--------- quits loop, and continues to write out “Step 4”
. write “Step 3”,!
```

IF, ELSE

Standard if-(then)-else constuction

```
if a>1 write “hello. ” set a=a+1 write a,!

if MyVar=1 do
. write “Success!”,!
else  do            ;<-------------- else MUST have TWO spaces after else command
. write “Failure!”,!
```

Postconditional logic (**:**).

```
write:(a>1) “apple” write:(a=2) “pear” write:(a<4) “grape”
```

This allows for just one part of a line to be skipped, not the entire line. For example, the equivalent of the line above with out a post conditional would have to be:

```
if (a>1) write “apple”  ;<---------- if (a<1) fails, anything after on this line would be skipped.
if (a=2) write “pear”
if (a<4) write “grape”
```

Much of Fileman code uses long one-liners, and postconditionals are used extensively!

FOR

The only looping command used.

Syntax: For [var initiation]:[variable incrimenter]:[comparitor for termination] [code to execute]

Example:

```
for i=1:1:100 write i,!
```

or

```
for i=100:-1:1 do
. write i,!
```

Can leave off parts of this

Example

```
for  i=1:1 write i,! quit:(i>100)
```

or

```
for  do  quit:(i>100)   ;<------- Note: FOR and DO must have TWO spaces after them
. set i=i+1
```

```
. do MyFn(i)
```

My most common loop. This will cycle through an array

```
set i=$order(^GLOBAL(“”))
if i>”” for  do  quit:(i=””)
. do MyFn(i)
. set i=$order(^GLOBAL(i))
```

$ORDER

A very powerful tool for cycling through nodes of a variable

```
set var(“pear”)=”green”
set var(“apple”)=”read”
set var(“bananna”)=”yellow”

set i=$order(var(“”))
if i’=”” for  do  quit:(i=””)
. write “index=”,i,!
. set i=$order(var(i))
```

output:

```
apple
bananna
pear
```

Note that when variables are created, they are automatically sorted. Order returns them in their sorted order, not in the order in which they were created.

$GET

Variables in M can have an undefined state. Attempts to access undefined variables will cause a programming error.

Example:

```
new myVar
write “Apple”,myVar,!
```

-output-

```
“Apple”
<CRASH HERE>
```

Solution? **$get()**

example:

```
write $get(myVar)  ;<--- will return “” if myVar is undefined
```

or `write $get(myVar,DefaultValue)  ;<--- will return DefaultValue if  myVar is undefined.`

$DATA

Provides information as to whether a variable contains information.

example

```
new myVar
write $data(myVar)          -->
set myVar=1
```

```
write $data(myVar)        -->
set myVar("index")=1
write $data(myVar)        -->
```

$PIECE

Allows retrieval/setting a data stored in sections of a string.

example

```
set s="apple^bananna^pear^grape"

write $piece(s,"^",3)   ; -->  outputs "pear"

set $piece(s,"^",2)="strawberry"
write s  ;--> outputs  "apple^ strawberry^pear^grape"
```

$$functions

Passing by value, reference, and by name

Example, passing parameters <u>by value</u>:

```
Mult(x,y)
        new result
        set result=x*y
        quit result

Main
        write $$Mult(5,3)   ;<---- outputs 15
        quit
```

Example, Passing parameters <u>by reference</u>:

```
Concat(data)
        new result
        set result=$get(data(1))_ " "_$get(data(2))_ " "_$get(data(3))
        quit result

Main
        new Var
        set Var(1)="Hello"
        set Var(2)="there"
        set Var(3)="everyone."

        write $$Concat(.Var)   ; <------ outputs "Hello there everyone"
        quit
```

Note: only the ENTIRE array can be passed, not part of it (see example below)

Example, Passing parameters <u>by name</u> (a.k.a. the variable "root")

```
Func(ref)
        new result
        set result=@ref@(1)
        quit result

Main
```

```
new Var
set Var(1)=”value”
write $$Func(“Var”)   ; <----- outputs “value”
quit result
```

This allows passing of *part* of an array. For example, this is NOT allowed:

```
write $$Funct(.Var(1))  ;<--- error
```

But this is allowed:

```
write $$funct(“Var(1)”)
```

or

```
write $$funct($name(Var(1)))
```

Optional parameters.

Any parameter may be considered optional (from the callers point of view).
The function will receive a null variable.
So a robust function will consider situation when passed variables are not defined.

Example:

```
Func(input)
        new result
        set result=+$get(input)+1
        quit result
Main
        write $$Func()    ;<--- will output “1”
        quit
```

The multi-use ^ character

- Used to indicate a “global” (i.e. disk-stored) variable/array
  - e.g. set ^MyVar(1)=”Keep this forever”
- Used to indicate a file that a function exists in (i.e.”global routines”)
  - e.g. do $$MyFunct^MyFile(123)
- Used as a “Naked” indicator
  - M keeps track of the last global referenced
  - example:
    - set ^VAR(“Animals”,”small”)=”mouse”
    - set ^VAR(“Animals”,”big”)=”elephant”
    - write ^Var(“Animals”,”small”) --> outputs “mouse”
    - write ^(“big”) --> outputs “elephant” <---- Shows use of naked indicator
  - “Remembers” up to next-to-the-last node. (e.g. ^VAR(“Animals”))
  - Only works with global variables, not local variables
- Often used as delimiter for Fileman data storage
  - e.g. ^PSNDF(50.6,0) = “VA GENERIC^50.6O^5376^5334” <-- contains 4 fields

MERGE

Merges two arrays
example:

```
set Array1(“Mammal”,”big”)=”elephant”
set Array1(“Mammal”,”small”)=”mouse”

set Array2(“Reptile”,”big”)=”dinosaur”
set Array2(“Reptile”,”small”)=”snake”

merge Array3(“Animals”)=Array1
```

merge Array3(“Animals”)=Array2

- array dump of merged array -
Array3(“Animals”,”Mammal”,”big”)=”elephant”
Array3(“Animals”,”Mammal”,”small”)=”mouse”
Array3(“Animals”,”Reptile”,”big”)=”dinosaur”
Array3(“Animals”,”Reptile”,”small”)=”snake”

$JOB (or $J)
The current job number

How M differs from many other languages.

- Global scope of variables: a KILL in *another* function can crash *your* function
- Need for name spacing.
- Variables can be undefined
- Whitespace IS significant (2 spaces after else, for etc.)
- Left-to-right evaluation → use parentheses!
  4+3*3 = (4+3)*3 , not 4 + (3*3) as would be mathematically expected.
- Somewhat limited Boolean testing from IF, via $T
- Case is not sensative for commands, but is sensative for variable names and functions.
- Abreviations of commands is allowed, using just first letter of command (bad for readability)
- Fantastic global variable type.
  Auto-sorting, permanence forever…
- Variable arrays and naming
  DIC=1 vs. DIC(0)=1 vs. ^DIC=1 vs. do ^DIC

Helpful Tools

VPE

- Open source. Very helpful
- Demonstration

Website for language reference (Mums by Example)

- URL:

TMGIDE debugger

- For GT.M only (Cache’ has its own debugger)
- Demonstration

Fileman/Kernel programmers reference (HTML format)

- Available at VA documentation library
- Demonstrations

Discussion of Fileman file structure
Programmers do not have to store their data in Fileman format, but I strongly recommend it because:

- VA Standards and Conventions (SAC) state that all data will be stored in Fileman format (there are a few exceptions around)
- Non-standard data storage is more difficult to deal with if you need to fix something.... you can’t use Fileman tools to edit the data etc.
- It will be much for difficult for other programmers to use your code if not in Fileman format

Exploration with VPE (vs. Fileman)
Show live demo of VPE and fileman DD tools

Headers, nodes, pieces

Data is stored in a particular node, at a given piece

Example: Field 1 stored at 0;4 means node 0, piece 4

You have to know the reference (or "root" of the file). In this example the file is stored at ^VAR(123,

to read a field value for record# IEN directly, one can do this:

```
set myVar=$piece($get(^VAR(123,IEN,0)),"^",4)
```

This will return the *internal* value of the field

Direct access versus using Fileman calls:

It is OK to *read* data directly as a variable, but *writing* data directly is not good in most circumstances. Doing so will fail to update triggers, cross-references etc, it bypasses security, and data may become out of sync. More about cross references ("xrefs") later

Introduction to programming with Fileman

Classic calls vs. new silent database server calls

Classic calls

- Often directly query the user and write output to the console
- Can often be set to "quiet" mode, provided there are no errors etc.
- Use a different method of passing parameters

  Because all variables are global in scope, the functions calls are set up by just setting variables prior to the call. For example:

```
set DIC=2
set DIC(0)="M"   ; Use multiple xrefs.  More detail avail in Fileman reference.
set X="DOE,JOEN"
do ^DIC    ; parameters in global-scope vars DIC and X
write Y,!    ; return is via global-scope var Y
```

Silent database server calls

- Designed for RPC calls, so no direct user interaction is expected
- Uses parameter passing more like other languages

  For example:

```
set MyVar=$$GET1^DIQ(2,1234,.01)  ; get field .01 of record 1234 in file 2
```

Quick aside... Common terminology/variable names

X and Y – X is input Y is output for classic Fileman calls.

"IEN" (vs. IENS)

- IEN = Internal Entry Number (i.e. record number)
- IENS = Internal Entry Number String – it is like a URL
  - IEN,parent-IEN,grandparent-IEN,greatgrandparent-IEN,
  - Note: **always has terminal comma** (',') even if only 1 IEN given
  - e.g. 15,1234, <---- IEN 15 inside file IEN 1234

DFN = patient number (IEN in file 2, PATIENT file)

U = ^

Internal vs. External format:

- Internal format: data as stored in database. E.g. dates --> date code etc.
- External format: a value as a user might enter it.

"Roots" – the NAME of an array

- an "Open" Root: ^TMG("Animals", <--- note, there is no closing parenthesis
- a "Closed" Root: ^TMG("Animals")

Cross references

These provide an index of data based on a given field for rapid lookup later.

Most (if not all) files have a “B” reference on the .01 field (every file has a .01 field)

Example: Here is the “B” cross reference for the file 50.6, stored in global ^PSNDF(50.6,*)

```
304) ^PSNDF(50.6,"B","ALBUMIN/ALLERGENIC EXTRACT,POLLEN MIX",1240) =
305) ^PSNDF(50.6,"B","ALBUMIN/STANNOUS CHLORIDE",1735) =
306) ^PSNDF(50.6,"B","ALBUMIN/STANNOUS TARTRATE",1756) =
307) ^PSNDF(50.6,"B","ALBUMIN/TECHNETIUM,TC-99M",488) =
308) ^PSNDF(50.6,"B","ALBUTEROL",1141) =
309) ^PSNDF(50.6,"B","ALBUTEROL SULFATE/IPRATROPIUM BROMIDE",5235) =
310) ^PSNDF(50.6,"B","ALBUTEROL/IPRATROPIUM",3416) =
311) ^PSNDF(50.6,"B","ALCLOMETASONE",1296) =
312) ^PSNDF(50.6,"B","ALCLOXA/PRAMOXINE/WITCH HAZEL",2882) =
313) ^PSNDF(50.6,"B","ALCOHOL",454) =
314) ^PSNDF(50.6,"B","ALCOHOL,ISOPROPYL/UNDECYLENIC ACID",2284) =
315) ^PSNDF(50.6,"B","ALCOHOL/ALUMINUM CHLORIDE HEXAHYDRATE",5236) =
316) ^PSNDF(50.6,"B","ALCOHOL/ANISE OIL/LICORICE COMPOUND",5237) =
317)  ^PSNDF(50.6,"B","ALCOHOL/ARNICA",2448) =
```

So, to find IEN of “ALBUTEROL”, one could use this:

```
set drugName=”ALBUTEROL”
set IEN=$order(^PSNDF(50.6,”B”,drugName,””))
```

More often, indexes are passed to fileman by name (e.g. “B”), directing how to look up data.

Lookup with ^DIC

Performs a lookup, using indexes (not a brute-force search)

Input parameters:

DIC file number or an explicit global root in the form ^GLOBAL(or GLOBAL(X,Y,.

DIC(0) Options, as follows: If null or undefined, no terminal output will be generated

The acceptable characters are:

A	Ask the entry; if erroneous, ask again.
B	Only the B index is used when doing lookup to files pointed-to by starting file.
E	Echo information.
L	Learning a new entry is allowed.
M	Multiple-index lookup allowed.
N	Uppercase "N" -- Internal Number lookup allowed (but not forced).
O	Only find one entry if it matches exactly.
Q	Question erroneous input (with two ??).
T	ConTinue searching all indexes until user selects an entry or enters ^^ to get out.
U	Untransformed lookup.
V	Verify that looked-up entry is OK.
X	EXact match required.

Example: if user interaction for lookup is wanted, I use “MAQE”

X The value to lookup, unless DIC(0) contains an “A”

DIC("A") (Optional) A prompt that is displayed for user input.

For example: set DIC("A")="Enter Employee to edit: "

DIC("S") (Optional) A string of M code that DIC executes to screen an entry from selection.

DIC("S") must contain an IF statement to set the value of $T. Those entries that the IF sets as $T=0 will not be displayed or selectable. When the DIC("S") code is executed, the local variable Y is the internal number of the entry being screened and the M naked indicator is at the global level @(DIC_"Y,0)"). Therefore, to use the previous example again, if you wanted to find a male employee whose name begins with SMITH, you would:

```
set DIC="^EMP("
set DIC(0)="QEZ"
set X="SMITH"
set DIC("S")="IF $PIECE(^(0),U,2)=""M"""
 do ^DIC
```

To execute: DO ^DIC

Similar functions: IX^DIC (allows specifying which index to start search with), MIX^DIC (use user-specified indexes for search)

Lookup/Data Retriever with FIND^DIC (a silent DBS API call)

FIND^DIC(file,IENS,returnFields,flags,lookupValue,maxNum,indexes,screenCode, identifierCode,targetRoot,msgRoot)

Input Parameters

file number of the file or subfile to search.

IENS (Optional) The IENS that identifies the subfile, if FILE is a subfile number.

returnFields (Optional) The fields to return with each entry found.
e.g. "@;.01;.02;.03I"

flags (Optional) Flags to control processing. (edited details)
A --Allow pure numeric input to always be tried as an IEN
B – B index used on lookups to pointed-to files.
C -- Use the Classic way of performing lookups on names. E.g. lookup value of "Smi,J", the Finder will find "Smith,John" but also "Smiley,Bob J."
K -- Primary Key used for starting index.
M -- Multiple index lookup allowed.
O -- Only find exact matches if possible. Otherwise returns all matches, partial and exact.
P -- Pack output.
Q -- Quick lookup. Finder assumes the passed value is in internal format.
U -- Unscreened lookup. Makes the Finder ignore screen stored at ^DD(file#,0,"SCR"))
X -- EXact matches only.

lookupValue -- (Required) Should be in external format unless the Q flag is used.

maxNum -- (Optional) The maximum number of entries to find. "*" designates all entries.

indexes -- (Optional) The indexes to search, as a list of index names separated by ^ characters.
E.g. "B^C^ZZALBERT^D" specifies four indexes to check in the order shown.

screenCode -- (Optional) Screening code to apply to each potential entry

identifierCode -- (Optional) Identifier code to set text for accompaning each found entry
E.g. "DO EN^DDIOL(""KILROY WAS HERE!"")" would include that string with each entry returned, as a separate node under the "ID","WRITE" nodes of the output array.

targetRoot -- (Optional) The array that should receive the output list of found entries. This must be a closed array reference and can be either local or global. If not passed, the list is returned descendent from ^TMP("DILIST",$J).

msgRoot -- (Optional) The array that should receive any error messages. This must be a closed array reference and can be either local or global. For example, if MSG_ROOT equals "OROUT(42)", any errors generated appear in OROUT(42,"DIERR"). If the MSG_ROOT is not passed, errors are returned descendent from ^TMP("DIERR",$J).

Output

targetRoot (and possible msgRoot) are filled. See format in Fileman reference manual.

Example

Do a lookup on the Option file, using the "C" index (Upper Case Menu Text). We'll let the Finder return default output, so we get the .01 field, the IEN, and the Identifier field (#1, Menu Text).

```
DO FIND^DIC(19,"","","","STAT","","C","","","TMGOUT")

-output of TMGOUT -
TMGOUT("DILIST",0)=2^*^0^
TMGOUT("DILIST",0,"MAP")=FID(1)
TMGOUT("DILIST",1,1)=DISTATISTICS
TMGOUT("DILIST",1,2)=ZISL STATISTICS MENU
TMGOUT("DILIST",2,1)=15
TMGOUT("DILIST",2,2)=187
TMGOUT("DILIST","ID",1,1)=Statistics
TMGOUT("DILIST","ID",2,1)=Statistics Menu
```

Other examples can be found in the Fileman manual...

Discussion of FDA's and IENS's formats

IENS = Internal Entry Number String – it is like a URL

- IEN,parent-IEN,grandparent-IEN,greatgrandparent-IEN,
- Note: **always has terminal comma** (',') even if only 1 IEN given
- e.g. 15,1234, <---- IEN 15 inside file IEN 1234
- Sometimes a IEN number is not known. For example when ADDING a new record, then special code ('+') is used.
  - e.g. "+1,1234," <-- means to add a new subrecord inside record 1234

FDA – Fileman Data Array. It is an array to pass to fileman containing fields and values

- e.g. for FILE# 1234
  - TMGFDA(1234,IENS,.01)="APPLE"
  - TMGFDA(1234,IENS,.02)="RED"
  - TMGFDA(1234,IENS,.03)="SWEET"
- Should be a name-spaced variable. Don't just use "FDA" --> funny errors

Examples:

Adding records with UPDATE^DIE

UPDATE^DIE(flags,fdaRoot,ienRoot,msgRoot)

flags -- (Optional) Flags to control processing. The possible values are:

E -- External values are processed. If this flag is set, the values in the FDA must be in the format input by the user. The Updater validates all values and converts them to internal format. Invalid values cancel the entire transaction. If the flag is not set, values must be in internal format and must be valid.

K -- the primary Key fields, not the .01 field, are used for lookup for Finding

S -- Saves the FDA instead of killing it at the end.

U -- Don't check key integrity. (CAUTION)

fdaRoot – The NAME of an FDA, which describes the entries to add to the database.

ienRoot -- (Optional) The NAME of an IEN Array. This should be a closed root. This array has two functions:

1) Requesting Record Numbers for New Entries. Examples:

- set TMGIENA(1)=10, try convert "+1" (from FDA) --> record# 10
- set TMGIENA(2)=11, try convert "+2" (from FDA) --> record# 11

2) Locating Feedback on What the Updater Did

- TMGIENA(1)=10, means that "+1" (from FDA) was put into record# 10
- TMGIENA(2)=11, means that "+2" (from FDA) was put into record# 11

msgRoot -- (Optional) the NAME of the array that should receive any error messages. This must be a closed array reference and can be either local or global. For example, if MSG_ROOT equals "TMGERROR("MyErrors")", any errors generated appear in MGERROR("MyErrors","DIERR"). If not passed, errors are returned descendent from ^TMP("DIERR",$J).

**Note: Don't try to add a record and a subrecord at the same time. It may be possible, but I have wasted much time hassling with this. Do it as two separate transactions.

Example:

```
new TMGFDA,TMGIENA,TMGMSG
set TMGFDA(22702,"+1,",.01)="A DESCRIPTION"
set TMGFDA(22702,"+1,",.02)="NOW"
do UPDATE^DIE("E","TMGFDA","TMGIENA","TMGMSG")
```

Modify existing records with FILE^DIE

FILE^DIE(flags,fdaRoot,msgRoot)

Input Parameters

flags -- (Optional) Flags to control processing. The possible values are:

E -- the values in the FDA must be in the format input by the user. The value is validated and filed if it is valid. If not set, values must be in internal format and must be valid; no validation or transformation is done by the Filer, but key integrity is enforced.

K -- LocKing is done by the Filer. (See discussion of Locking.)

S -- Save FDA. If not set and there were no errors during the filing process, the FDA is deleted. If this flag is set, the array is never deleted.

T -- Transaction is either completely filed or nothing is filed. If you use the "T" flag, you must also pass the "E" flag, and pass values in external format. If any value is invalid, nothing is filed, and the error array will specify which fields were invalid. Without this flag, valid values are filed and only the invalid ones are not.

U -- Don't enforce key Uniqueness or completeness. Without the U flag, the values in the FDA are checked to ensure that the integrity of any key in which an included field participates is not violated. (CAUTION)

fda_root -- (Required) The NAME OF (i.e. root) of the FDA that contains the data to file.

msg_root -- (Optional) The NAME OF (root) of an array (local or global) into which error messages are returned. If this parameter is not included, error messages are returned in the default array-^TMP("DIERR",$J).

Example: -- Alter record# 12 in FILE 22702
new TMGFDA,TMGIENA,TMGMSG
set TMGFDA(22702,"12,",.01)="A DESCRIPTION"
set TMGFDA(22702,"12,",.02)="NOW"
do FILE^DIE("EK","TMGFDA","TMGIENA","TMGMSG")

Data retrieval

$$GET1^DIQ vs. Direct access.

Example:
write $$GET1^DIQ(22702,"12,",.01) ; --> outputs "A DESCRIPTION"
Easier to use, but slower (more instructions to achieve)

write $piece($get(^TMG(22702,12,0)),"^",1) ; --> outputs "A DESCRIPTION"
Harder to use, required knowledge of where data is stored.

Access restrictions

- Many fields have restrictions so that data may not be read or written with Fileman.
- A user's file access code is stored in DUZ(0)
- @ is the highest level of access, but it will NOT supercede ^ access.
- i.e. if DUZ(0)="@" then user should be able edit any field, unless field access holds a ^
- The locks come in a variety of forms, and are stored different places. Here are field-level locks, stored in the data dictionary (^DD):
  - character will be compared to user's file access code.
  - ^DD(File#,Field#,8) Read access character for the field.
  - ^DD(File#,Field#,8.5) Delete access character for the field.
  - ^DD(File#,Field#,9) Write access character for the field.
  - ^DD(File#,Field#,12.1) Contains the code which sets DIC("S") if a screen has been written for a pointer or a set of codes. Might be usable to prevent selection of a record in Fileman.
  - ^DD(File#,Field#,"DEL") Contains code that determines if the field can be deleted. If $T is set to 1, the field cannot be deleted.
  - ^DD(File#,Field#,"LAYGO") Contains code that determines if an entry can be added. If $T is set to 0, the entry cannot be added.
- Here are file-level locks stored in the Dictionary of Files (^DIC)
  - ^DIC(File#,0,"AUDIT") -- Audit Access
  - ^DIC(File#,0,"DD") -- Data Dictionary Access
  - ^DIC(File#,0,"DEL") -- Delete Access
  - ^DIC(File#,0,"LAYGO") -- LAYGO Access
  - ^DIC(File#,0,"RD") -- Read Access
  - ^DIC(File#,0,"WR") -- Write Access

IO discussion

$I[O] = the current IO device.
$P[RINCIPLE] = the initial IO device

Commands OPEN, USE, CLOSE control devices. However, don't use OPEN, CLOSE directly

Here is an example of selection an IO channel
write "Select device to output data to.",!
write "HFS (i.e. Host File System) will allow output to a file.",!
write "(A file name will be asked after HFS is chosen)."

```
set %ZIS("A")="Enter Output Device: "
set %ZIS("B")="HFS"
do ^%ZIS  ;standard device call (a Kernal API call)
if POP write “ERROR”,! quit
use IO   ;”will change $IO=IO
         do MyOutputFunction()
do ^%ZISC ;" Close the output device
```

HFS entry in DEVICE file.

```
   Here is the entry from my system
  .01-NAME : GTM-UNIX-HFS
  .02-LOCATION OF TERMINAL : Host File Server (GT.M)
  .03-MNEMONIC :
     Multiple Entry #1
     .01-MNEMONIC : HFS
     Multiple Entry #2
     .01-MNEMONIC : GTM-LINUX-HFS
   1-$I : /tmp/vistahfs.txt       ;<------ this is default entry, but user can change.
   2-TYPE : HOST FILE SERVER
   3-SUBTYPE : P-OTHER
   4-ASK DEVICE : YES
   5-ASK PARAMETERS : NO
  5.1-ASK HOST FILE : YES
  5.2-ASK HFS I/O OPERATION : NO
```

Real examples:
Adding (registering) a patient

RPC Communications.
RPC server code.
Review help file showing client vs. Server code
Show some of my server code.
Example of binary upload /download