Programming Hooks 102: Rules for Use

by Rick Marshall

Introduction

This is the second in a series of articles that examines programming hooks available in VA FileMan. A programming hook is a significant point in the sequence of events that makes up a standard database activity at which a programmer can insert M code. To use VA FileMan programming hooks effectively, developers must abide by some simple rules. Some rules always have existed but either have not been effectively communicated to developers, or have been missed by developers in the flight to experience. Other rules arise only now from changes in M Technology and the changes that must follow in FileMan as a result. Programmers who do not learn and apply these rules will doom their databases to stagnation or obsolescence. This time, *Focus on FileMan* spotlights these all-important rules for using programming hooks.

Unsupported Hooks

Rule: Programmers must not use unsupported hooks. While this may seem to be a simple proposition, it is not. Although novice programmers can adhere easily to this rule by sticking to the hooks described in the FileMan manuals, more experienced programmers have two challenges facing them.

First, experienced FileMan programmers may have unsupported hooks in their databases without realizing it. FileMan's early developers (George Timson and the late Michael Distaso) had the freedom to release new versions with little overhead, and frequently released versions of FileMan with hooks and other features added for specific programmers. They easily could remember all of the agreements they made regarding which hooks were supported for use by programmers who were not part of the FileMan team, and which were for FileMan's internal use only.

The advantages were great for custom-fit features, but several factors have complicated this practice. The number of FileMan programmers and hooks, and the size of FileMan have all become too cumbersome for human memory to track.

Now FileMan development involves a team of several developers, making written documentation essential even within the team, let alone for the outside world. We undertook a survey of undocumented features developers use and documented what we found. Thus, as far as we could, we solved this problem by legitimizing existing practice (a strategy known as grandfathering).

A problem facing experienced programmers comes from their key advantage over novices, which is their programming habits. Good programmers develop personal styles, consisting of standard ways to solve standard problems. Unique styles enable them to program rapidly, without struggling over every aspect of the problem at hand, and thereby letting them focus only on what's new about the problem. Styles retain the history and experience of their users, and experienced FileMan developers may retain among their programming habits the reflexive use of features they do not realize lack documentation.

With the relatively new reliance on written documentation to determine which features are supported or not, experienced programmers may be building their software houses on foundations that unintentionally will be pulled out from under them.

There is an easy solution. Reread the FileMan documentation, and then reread your software. The documentation will have changed considerably since you learned to use FileMan, and you will find features and hooks of value that were not documented (or did not exist) when you started out. Rereading your software —particularly looking at data dictionaries, templates, and routines—will help identify features to use that are not documented. If you find any, contact us by e-mail or at the address following this column, and we will help you by either grandfathering in your feature or by phasing you into something newer.

Unsupported Variables

Rule: Don't use unsupported variables within a programming hook. First, don't assume that variables supported for one hook are supported for any other hook; they must be treated on a case-by-case basis. Second, don't assume that any variable defined at a given hook is available for use by that hook. I know it is common practice for novice programmers to issue `ZWRITE` commands within an M cross-reference to learn "what's

available," and then to pick out variables for the cross-reference's code to use from that list. This is a setup for disaster and M programmers should know better than to get into one.

If M means never having to say you're sorting, it also means not doing everything you can. We believe that the M approach of opening the world to you makes you a better programmer because you avoid problems by choice. Here's a place to apply that discretion. No matter how useful a variable in the symbol table looks, if it is not documented as supported for use, it may disappear from the symbol table when the VA releases a newer version of FileMan. More insidiously, a useful variable may only exist under the right conditions. Stick to the supported variables, and if what you need is not available, contact the FileMan team to get a reliable, supported way to achieve what you want.

As with unsupported hooks, experienced programmers need to refamiliarize themselves with the documentation and their own code, and identify those unsupported variables used in their own applications.

Symbol Table Cleanup

Rule: Clean up after yourself. (Remember what you learned in kindergarten.) The MUMPS Development Committee has given you the NEW command, so use it to NEW your variables. The only variables that should have changed when your hook completes execution are those documented for output from the hook to FileMan and your own package-wide variables.

Reentrance and Compatibility

Rule: Don't assume everything is reentrant and compatible. If within a programming hook you make a call to FileMan, you are nesting FileMan calls. This will work only if the calls in question are reentrant (in the case of nesting a call within itself) or compatible (in the case of different calls being nested).

In fact, no code is reentrant or compatible unless you code it to be that way. The documentation is your only guide here. Remember, too, that reentrant calls may nest your programming hook. Is your programming hook reentrant? You should probably document the reentrance of your own package as well. For example, if your hook is an input transform, then FileMan's standard lookup program, DIC, will call it. If your input transform contains *another* DIC call, then it, too, *may* call your input transform. Unless you write your input transform properly, you will step on your own variables.

Flow of Control

For M, as with most programming languages, the default flow of control is strictly sequential, executing commands left to right, top to bottom, in order. Certain M commands permanently alter that flow in the current line or routine: BREAK, ELSE, FOR, GOTO, HALT, IF, and QUIT. For example, given the following line, M will execute every command:

```
S DIC-19,DIC(0)="M",X="ZZ TEST
OPTION",Y=0 D ^DIC S ZZTOAD_Y
```

but if we replace the DO with a GOTO then ZZTOAD will never be set, and replacing D ^DIC with a FOR command will cause an infinite loop.

In FileMan programming hooks, use these commands with caution. Some programming hooks expect you to alter the flow of control. For example, DIC("S") expects you to set $TEST, which usually requires an IF command (see *VA FileMan Version 20.0 Programmer Manual,* page 42). Others, though, expect no alteration of the flow of control. FileMan appends more commands to the end of some progrmming hooks, so if you alter the flow of control FileMan may never execute these appended commands, or may repeat them unexpectedly. For example, if you include an IF command in a field's output transform, and then attempt to print a report in which you sort by that field, under the right conditions your report will include entries you do not want (see *VA FileMan Version 20.0 Programmer Manual,* pages 170 to 171).

Read the documentation carefully. If it disallows use of certain commands, avoid them. If it explicitly permits the use of certain commands, feel free to so so. If it does not discuss whether you may use a certain flow control command, assume you cannot.

Rule: If you must use a flow control command in a restricted programming hook, bury it in the execution stack so it does not alter the hook's flow of control. You can do this by wrapping the command up in an XECUTE statement. For example, turn

```
F ZZA=1:1:10 S ZZB(ZZA)=ZZA
```

into

```
X "F ZZA=1:1:10 S ZZB(ZZA)=ZZA"
```

Alternately, you can do this by putting the risky code in a routing, and using the DO command to call it. By replacing the restricted flow control commands with a DO or XECUTE, you will avoid changing the new hook's flow. Eventually, FileMan developers hope to restructure FileMan to permit developers to use these flow control commands more freely, but for now you should take precautions.

Segregating Input/Output

One of the greatest challenges is the move to a graphical user interface (GUI). No longer will you control the order of events. Instead, your users will choose which gadgets to invoke and in which order. In such an environment, you do not have READ/WRITE control. Instead, the window driver will perform all the input/output with the user. This has serious implications for most M programmers used to mixing freely input/output with database activities and computations. For GUI, this must stop.

Rule: You must segregate your input/output from your other activities in preparation for GUI and the sooner you start, the easier the transition will go when the M windowing application program interface arrives. (See p. 45 for an article on MWAPI.) In particular, it is time to review all your FileMan programming hooks to locate READs and WRITEs, or DOs or GOTOs in order to code them as READs and WRITEs. Our surveys of the Decentralized Hospital Computer Program (DHCP) have confirmed that some input and output transforms issue WRITEs, but we have found no output transforms with READ commands. Compile a list of places where input/output is embedded within a programming hook. We will provide you a tool to issue WRITEs or to pass them off to the windowing environment, depending on the current input/output handling with the user. We recognize this creates work for all of us, but the move to GUI will gain M greater acceptance with new markets and increased appreciation from our current users.

Interface versus Database

Transaction processing and screen-oriented interfaces present their own challenges. One key challenge is to separate database activities from user-interface activities. For example, it is common to use an M cross-reference for one field to set a value into another field so it will be presented as a default value when FileMan prompts the user for that second field. In a transaction-oriented environment such as ScreenMan, the database is not updated until after the user completes the form. Database changes will not take effect until it is too late for them to function as defaults, so that existing schemes for getting the right defaults will no longer work.

Fundamentally, this is a problem of improperly mixing the database with the interface. It has become a serious problem only with the advent of new technology (specifically, screens and transaction processing) that modernized our approach to database and interfaces, sharpening the distinctions between them. When solving interface problems, look for new ways to solve them and call on us for the tools you need where they do not already exist.

A Job Well Done

Considering the complexity of database management, this is actually a surprisingly short list of rules that are key for using FileMan's programming hooks. Any progress comes with a price, of course, and the move to a modern standard and a modern database management system is no exception. Remember that once you have completed your work in this area, it will remain done. Documented hooks and their features will remain documented. Likewise, segregated input/output, interface, and computation will remain so. Applying these rules will help you make the most of the FileMan programming hooks we will be describing in later issues of *M Computing*.

Forward your FileMan questions to the FILEMAN DEVELOPMENT TEAM on the Veterans Affairs' FORUM system, or write to: VAISC6-San Francisco, Suite 600, 301 Howard Street, San Francisco, CA 94105.

Rick Marshall works at the Seattle Development Satellite office of the VA's San Francisco Information Center. He is a member of the FileMan development team.
