# Programming Hooks 103 Lookup: Part I

*by Rick Marshall*

## Introduction

This is the third in a series of articles exploring the use of the programming hooks available to applications based on FileMan. For purposes of this series, a programming hook will be considered to be any significant point at which you (a programmer) can insert M code into the sequence of events that makes up a standard database activity. The programming hook is one of the most powerful but one of the least understood features of FileMan.

Much of application programmers' understanding of these hooks has come from oral tradition and folklore, secrets shared over lunch, at conventions, and in phone conversations with programmers. The goal of this series is to bring the written and oral traditions back in synch, to clarify what these hooks should do, and to examine the strengths and weaknesses of some common coding tricks.

## What Is a Lookup?

This article investigates the hooks available in FileMan's Lookup module. FileMan refers to the standard process of picking a record in a file as a "lookup." Although FileMan provides similar, more powerful features in the Search module, your users will usually pick records with the Lookup module. There are actually two different kinds of lookups.

General lookups are when FileMan directly asks the user to pick a record from a file; this could be for any of a number of different reasons. For example, when a user performs an inquiry on a file, FileMan first asks the user to perform a general lookup on the file, so FileMan knows which records to display.

Pointer lookups take place when FileMan asks the user to enter the value for a field that happens to be a pointer to another file. In order to ensure that the user enters a valid value for the field, FileMan must actually perform a lookup on that pointed-to file. The various characteristics of the field being edited act as a kind of filter on the lookup, as we will see when we examine the programming hooks available in lookups.

Lookups can have different characteristics, such as which of the files' indexes can be used to help the users pick a record. Since the focus of this article is on programming hooks, we will cover these distinctions only where they actually affect the hooks in question.

Finally, many procedures informally called lookups are actually composite activities. For example, any lookup performed with LAYGO permitted (in which users may add new entries to the file) is actually a lookup followed by an optional add. Many LAYGO lookups also prompt the user to fill in field values, which adds editing to the composition. In the "Programming Hooks" series we will decompose these composite calls as much as possible. So, for example, our discussions of lookup hooks will not include adding or editing.

Let's begin our examination of the lookup hooks.

## Special Lookup Routine

(AKA: LOOK-UP PROGRAM or Special Look-up Program)

Syntax: Free text, 3-6 characters, routine name, no ^, no line reference, routine name cannot begin with `DI`.

Input & Output: same as `DIC` entry point.

Creation: LOOK-UP PROGRAM prompt, EDIT FILE, UTILITY FUNCTIONS.

Data Dictionary Location: `^DD(filenumber,0,"DIC")`.

You may need to write your own lookup routines to handle an unusual file. This happens only in very rare circumstances, because FileMan is flexible enough to support most standard variations on its typical structure. However, if your file is different enough you may decide that getting FileMan's Lookup to support your file is more trouble than it is worth.

FileMan lets you name a replacement for `DIC`, which you must then write to perform lookups with the same input and output parameters as `DIC`. In essence, you must write a black box that behaves exactly like `DIC` in every respect, except that it works differently on the inside, and only applies to a single file. The calls to `DO^DIC1`, `DQ^DICQ`, and `FILE^DICN` should help you write it; you will need help because this is a lot of work.

While this feature is slated for permanent support, we are investigating the feasibility of retiring it and moving applications back to `DIC`. Any users of this hook should contact the FileMan development team to discuss it.

The best example of this is the Special Lookup Routine `DPTLK`, on the Patient file. It illustrates both the kinds of circumstances under which someone might need this hook, and the complexity of the undertaking.

## Field Executable Help

(AKA: XECUTABLE 'HELP')

Syntax: M code, 1-245 characters, checked by FileMan's `DIM` routine.

Input & Output: none in documentation (see below).

Creation: XECUTABLE 'HELP' prompt, MODIFY FILE ATTRIBUTES.

Data Dictionary Location: `^DD(filenumber,.01,4)`.

You may need to give your users special help with some pointer lookups. The most common way to do this is to add some additional text to the help messages usually shown. If the text of that help is not static, or if the help you want to display in response to a user entering a single question mark will not fit in the normal help prompt field (i.e., it's longer than 245 characters), then you will need to use this hook.

Whenever the extra help you want to provide involves more sophisticated processing, you need this hook. For example, help systems based around statistical analysis of the kinds of mistakes your users tend to make will always need this hook. We will look at some other examples of processed help below.

### When Does Executable Help Hook into Lookups?

In general, FileMan will execute this hook when the user requests help with the field you place it on. Since most fields (even the .01) are not involved in strict lookups, it turns out that there is only one place where Executable Help does hook into the lookup process. During pointer lookups, your hook on the pointer field fires off whenever your users enter question marks.

FileMan does not execute the Executable Help of the pointed-to file unless LAYGO is permitted, and general lookups without LAYGO don't execute your hook at all. Also (no real surprise here), when your pointer lookups are silent (that is, when they do not interact with your users), none of your Executable Help hooks into the lookup process.

### Position

To customize your help display properly, you will need to consider where FileMan places the cursor before and after executing this hook. The following example should help illustrate the sequence. Notice the differences between ?-help and ??-help. I have filled in text to help clarify the sources of the various pieces of help shown. The Executable Help performs no formatting of its own; it just outputs a line of text, so any formatting you see comes from FileMan. This example shows a pointer lookup from the Musician field of the Song file, which points to the Musician field:

```
MUSICIAN: ?

     Help Prompt for Song's Musi-
     cian (pointer): Answer
     should be the name of the musi-
     cian who recorded this
     song.

Field Executable Help for Song's Mu-
sician (pointer).

Answer with MUSICIAN NAME

Choose from:

     AMOS,TORI
     ANDERSON,LORI
     BUSH,KATE
     MERCHANT,NATALIE
     SHOCKED,MICHELLE
     VEGA,SUZANNE

MUSICIAN: ??

Field Executable Help for Song's Mu-
sician (pointer).

     Field Description for Song's
     Musician (pointer): This
     field stores the name of the mu-
     sician who recorded this
     song. Because song titles are
     not unique, this field
     helps identify the song.

Choose from:

     AMOS,TORI
     ANDERSON,LORI
     BUSH,KATE
     MERCHANT,NATALIE
     SHOCKED,MICHELLE
     VEGA,SUZANNE

MUSICIAN:
```

## Formatting

The example above shows that FileMan will position the cursor at the start of a new line before executing your code, and will perform a line feed and carriage return afterward. In fact, if you provide no executable help FileMan leaves a blank line where it would have appeared. This means you need to format your text a little to make it match the flavor of the surrounding text.

Canned text messages should indent five spaces to the sixth character. Since they will be placed differently in LAYGO lookups and may not flow directly into or out of the adjacent text and prompts, you should put a line feed before and after.

Some terminals will not wrap for you, but instead will truncate any text beyond the right margin, so you must handle the wrapping of your text. If your application is developed for a Kernel environment, you may use the Device Handler variable IOM to tell you the right margin width. If your application lacks Kernel support, you have several choices. You can take steps independently to ensure a local variable will equal the right margin width, call %ZIS each time to get IOM, or assume a standard width of eighty characters. If you choose the last option, be aware that your text will probably display incorrectly on some terminals.

## How to Avoid Direct WRITEs: EN^DDIOL

MWAPI, ScreenMan, and client/server configurations are among the new computer science technologies that will bring a low tolerance for spontaneous READs and WRITEs to the M world. Since the FileMan data dictionaries are littered with direct READs and WRITEs, we have a problem. FileMan needs to somehow intercept that I/O and redirect it to the appropriate user interface, but since such activities bypass FileMan and directly execute M commands, there is no way to do so.

Our solution is to introduce tools that will perform READs and WRITEs for the applications, allowing us to intercept I/O whenever it is inappropriate. READs are not a major problem because very few programming hooks perform READ commands, and the Reader utility has been available for several years now.

To help solve the WRITE problem, FileMan version 20 introduced the Writer: EN^DDIOL. In version 20 this call simply WRITEs the value or array passed to it, but in version 21 it begins its transformation into the solution to the I/O problem. When called in a ScreenMan environment, the version 21 Writer will direct the I/O down to the help portion of the screen, preventing the WRITEs from inadvertently disrupting the interactive portion of the screen. When called in an environment that requires no direct I/O, it loads the text passed to it into standard output arrays, where interface handling tools can then manipulate and display the text in whatever fashion is appropriate for the current user interface.

What this means for you as application developers is that you need to begin converting your database hooks from direct WRITEs to calls to the Writer. Doing so in version 20 will result in no direct change to your hook's behavior, but will position your files to make the move forward to graphical user interface and client/server compatibility. These are relatively simple changes for you to make, and biting the bullet now will save you one more piece of work to coordinate later on when the major work of converting to GUI is upon you. Even if you plan to leave your application in scrolling mode in perpetuity, converting to the Writer will allow local sites to build ScreenMan, MWAPI, and client/server applications based around your files.

Executable Help will be one of the main places where this conversion is necessary, since it usually consists of WRITE commands. In the box is a before and after example of the code for an Executable Help, and how it looks when executed.

```
>ZP XHELP2:END
XHELP2  ;Executable Help for File # 3298428.7, Field # 1
        W !?5,"Here is your executable help."
        W !?5,"I hope you like it.",!
        Q
        ;
XHELP3  ;New Help for File # 3298428.7, Field # 1
        N ZZA
        S ZZA(1)="     Here is your executable help."
        S ZZA(2)="     I hope you like it."
        S ZZA(3)=""
        D EN^DDIOL(.ZZA)
        Q
        ;
>W !,"BEFORE:",! D XHELP2 W !,"AFTER:",! D XHELP3 W !,"NEXT
PROMPT"
```

```
BEFORE:

   Here is your executable help.
   I hope you like it.

AFTER:

   Here is your executable help.
   I hope you like it.

NEXT PROMPT

   >
```

## Distinguishing Between ? and ??

You may want to tailor your Executable Help to respond differently to different levels of help requested by the user. For example, you may be happy with the ?-help, but want to add something powerful to the ??-help. Although it has not been documented so far, a variable is available to help with this problem, has been available for a long time, and is even used by FileMan itself in dealing with this issue. X will equal the user's input, either "?" or "??".

## Using `DQ^DICQ` in Executable Help

One common trick applications can use to generate a list of entries to display, to make the field's help resemble that of a pointer field, is to call the FileMan utility `DQ^DICQ`. You will need to do something like this any time you want to create a free text pointer, in which the field must behave to the user like a pointer but must store the external, free text value instead of the internal record number. Free text pointers are especially useful wherever you must archive the current resolved value in case it changes later, but it has other uses as well. So, for example, a pointer to the device file might store "DIRECTOR'S PRINTER" instead of "42" (assuming the Director's Printer was the forty-second entry in the Device file). Free text pointers are just one situation in which you might want to alter the executable help in this way.

In this example, which we will look at again when we discuss input transforms, our sample file has a free text pointer field, DEVICE, which records the name of the device used to print this monthly report. This field is a free text pointer to the Device file, and has an input transform set up to ensure proper free text field behavior. To make the field's help behave properly, we have entered the following executable help:

```
N D,DO,DIC,DZ,X,Y
S DIC="^%ZIS(1,",DIC(0)="M",D="B
  "
D DQ^DICQ
```

Notice the care we are taking to protect the key variables here; we have `NEW`ed all the input variables to the `DQ^DICQ` call along with the two common variables X and Y in an attempt to help FileMan recover from our external call. `DQ^DICQ` will change a lot of variables, and being older code for now it will clean up after itself with `KILLS` rather than with `NEW` commands. This gives it a wide footprint, and makes it impossible to use in a hook that requires reentrancy. Fortunately, this use of our Executable Hook in a pointer lookup does not require reentrancy, so simply cleaning up our symbol table with a `NEW` command is enough to let FileMan recover from the call.

## Using `^DIC`, `IX^DIC`, and `MIX^DIC1` in Executable Help

For more specialized help similar to that provided by `DQ^DICQ`, you may need to call `DIC` (or one of its two other incarnations, `IX^DIC` and `MIX^DIC1`) itself; the control it gives you over indexes, screens, and identifiers may be exactly what you need. In this example, the Patient file has a field called Attending Physician. This field is a pointer to the New Person file, with a screen to ensure that the only entries selectable are medical providers of some kind. Since the New Person file also includes nonphysician computer users, the Executable Help for this field has been set up to ensure that only providers are shown when the user asks for help:

```
    N D,DO,DIC,X
+1  S X="??",DIC("S")="I$D(^VA(2
    00,""AK.PROVIDER"",$P(^(0),U
    ),+Y)),$S('$G(^VA(200,+Y ,""
    I"")):1,DT'>+^(""I""):1,1:0)
    "
+2  S DIC="^VA(200,",DIC(0)="EQ"
    ,D="AK.PROVIDER"
+3  D IX^DIC
```

By calling `IX^DIC`, the developer restricted the lookup to an index that contains only medical providers, and the input value of "??" ensures that `IX^DIC` will provide the correct kind of help rather than performing an actual lookup. Notice again the careful `NEW`ing of the input variables to avoid disrupting the call outside of this programming hook. Similar calls can be made for the other two `DIC` variants, for much the same purpose.

## Limitations of I/O

One drawback to these various DIC calls you have seen is that they generate direct I/O. This restricts the files that use these calls to a scrolling mode interface. When the Patient file developers move it forward to a GUI interface, they will need to convert all programming hooks that use this kind of direct I/O to use tools compatible with a GUI interface, tools that issue no direct `READS` or `WRITES`. As you will see in the months ahead, FileMan version 21 offers a wide range of tools designed to provide the equivalent features of these calls, but in a silent fashion compatible with GUI. Tools such as the Lister, Helper, and Retriever will eventually become commonplace in FileMan programming hooks as these files move forward technologically.

## Alternative: Prompting Yourself

An alternative to using Executable Hooks to provide the help you need for pointer lookups is to prompt the user yourself. This option offers you complete control over the interaction, including the help processing, and also allows you to intervene in more lookup situations, since Executable Help is limited to pointer lookups. Whether GUI, screen-driven, or roll and scroll, once your interface has collected the user's input values, you can pass them into a silent lookup call.

## Conclusion

Next time we will continue examining the features and special uses of other programming hooks available during lookups. **M**

Forward your FileMan questions or topics you would like to see addressed in this column to the mail group FILEMAN DEVELOPMENT TEAM on the VA's FORUM system, or write to: VAISC6-San Francisco, Suite 600, 301 Howard Street, San Francisco, CA 94105.

Rick Marshall works at the Seattle office of VA's San Francisco Information Systems Center. He is a member of the VA FileMan development team, teaches the VA Kernel class at the MTA Annual Meeting each year, and is active in MDC.

# CALENDAR

### June 13–17, 1994

**MTA–NA 23rd Annual Meeting—Windows of Opportunity, Reno, Nevada.**

### October 26–28, 1994

USENIX Symposium on Very High Level Languages, El Dorado Hotel, Sante Fe, New Mexico. Call 714-588-8945 for details.

### November 7–11, 1994

MTA–Europe Annual Meeting, Luxembourg.

### January 16–20, 1995

USENIX Winter 1995 Technical Conference, Marriott Hotel, New Orleans, Louisiana. Call 714-588-8943 for details.

To have your organization's meeting(s) listed in the *M Computing* Calendar, send details to *M Computing*, Managing Editor, M Technology Association, Suite 205, 1738 Elton Road, Silver Spring, Maryland 20903-1725. Materials must be received four months prior to the date of the event(s).

# An Invitation for Authors and Would-Be Authors

*M Computing* editors welcome articles and news items from our readers. Tell us about M application areas, new systems and installations, interfacing, techniques, technology challenges, or something else related to the world of M. Send book reviews, product announcements, press releases, other materials, or a brief description of your proposed article to Marsha Ogden, Managing Editor, MTA, Suite 205, 1738 Elton Road, Silver Spring, Maryland 20903-1725. Phone: 301-431-4070; Fax: 301-431-0017; or use FORUM.

JUST TWO WRITING OPPORTUNITIES LEFT IN 1994!
BOOT UP YOUR WORD PROCESSOR!
September Focus: *Interoperability*
Deadline: July 1, 1994
November Focus: *Client/Server Technologies*
Deadline: August 17, 1994