VA FILEMAN

PROGRAMMER MANUAL

Version 22.0

March 1999

Department of Veterans Affairs
V*IST***A** Software Development
Infrastructure Product Line

Preface

The *VA FileMan Programmer Manual* is designed to provide you, the Veterans Health Information Systems and Technology Architecture (**V***IST***A**) developer, with information about the programming functions of VA FileMan. This manual covers the APIs (Application Programming Interfaces) and using VA FileMan's developer tools. VA FileMan is **V***IST***A**'s database management system.

This manual is a full reference for all entry points in VA FileMan's APIs:

- Classic FileMan
- Database Server (DBS)
- ScreenMan API
- Browser
- Import Tool
- Extract Tool
- Filegrams

This manual shows how to use features of VA FileMan that are likely to be used by developers and IRM staff. In most cases you must have programmer access (DUZ(0)="@") to use these features:

- Global File Structure
- Advanced File Definition
- ScreenMan Forms and using the ScreenMan Form Editor
- VA FileMan Functions
- DIALOG File
- DIFROM

The *VA FileMan Programmer Manual* is available in two formats:

- Adobe Acrobat Portable Document Format (PDF), and
- Hypertext Markup Language (HTML) format

Both are available at the FileMan Home Page:

http://www.vista.med.va.gov/softserv/infrastr.uct/fileman/index.html

Manuals in HTML

Why produce an HTML (Hypertext Markup Language) edition of the VA FileMan Programmer Manual?

- The HTML versions of the VA FileMan manuals are useful as online documentation support as you use VA FileMan. HTML manuals allow you to instantly jump (link) to specific topics or references online.
- The VA FileMan HTML manuals are "living" documents that are continuously updated with the most current VA FileMan information (unlike paper or printed documentation). They are updated based on new versions, patches, or enhancements to VA FileMan.
- Presenting manuals in an HTML format on a web server also gives new opportunities, such as accessing embedded multimedia training material (e.g., movies) directly in the manuals themselves.
- Providing manuals in a native online format (HTML) also helps introduce HTML and web servers to the **V***IST***A** user community as documentation platforms for VHA.
- As more user workstations become network-capable, access to information in these HTML manuals is increased by making them available over the VA network.

Manuals in PDF

Why also produce a PDF (Portable Document Format)?

- Adobe Acrobat's PDF is a means of universal document exchange. Documents in PDF format look the same and contain the same information as the original.
- PDF electronic documents offer the advantage of being easily being distributed or stored on the web.
- They can be viewed on any computer or printed when desired.

Feedback

The VA FileMan Development Team encourages you to send them questions and comments about the completeness and accuracy of this manual and suggestions for its improvement. Send us feedback at:

G.FMTEAM@FORUM.VA.GOV.

Table of Contents

Orientation.....Orientation-1

Introduction Introduction-1

What is VA FileMan?.....1

Functional Description 1

Standalone VA FileMan 2

Part I: Major APIs.....I-1

1. Classic VA FileMan API.....1-1

Introduction.....1-1

Classic Calls Cross-referenced By Category.....1-3

Classic Calls Presented in Alphabetical Order.....1-5

X ^DD("DD"): Internal to External Date.....1-6
EN^DDIOL: Loader.....1-8
^DIAC: File Access Determination.....1-12
EN^DIB: User Controlled Editing.....1-13
^DIC: Lookup.....1-14
IX^DIC: Lookup/Add.....1-32
DO^DIC1: File Info Setup.....1-35
MIX^DIC1: Lookup/Add.....1-36
WAIT^DICD: Wait Messages 1-39
FILE^DICN: Add.....1-40
YN^DICN: Yes/No.....1-43
DQ^DICQ: Entry Display for Lookups.....1-44
DT^DICRW: FM Variable Setup 1-45
EN^DID: Data Dictionary Listing.....1-46
^DIE: Edit Data.....1-47
^DIEZ: Input/Compile.....1-59
EN^DIEZ: Input/Compile 1-60
^DIK: Delete Entries.....1-61
EN^DIK: Reindex.....1-63
EN1^DIK: Reindex.....1-65
ENALL^DIK: Reindex.....1-67
IX^DIK: Reindex 1-69

IX1^DIK: Reindex 1-71
IXALL^DIK: Reindex 1-73
^DIKZ: Cross-reference Compilation....1-75
EN^DIKZ: Compile 1-76
$$ROUSIZE^DILF: Routine Size 1-77
^DIM: M Code Validation 1-78
DT^DIO2: Date/Time Utility 1-79
^DIOZ: Sort/Compile 1-80
EN1^DIP: Print Data 1-81
^DIPT: Print/Display....1-101
DIBT^DIPT: Sort/Display 1-102
^DIPZ: Compile 1-103
EN^DIPZ: Print/Compile 1-104
D^DIQ: Display 1-105
DT^DIQ: Display 1-106
EN^DIQ: Display....1-107
Y^DIQ: Display....1-108
EN^DIQ1: Data Retrieval....1-109
^DIR: Reader 1-113
EN^DIS: Search File Entries....1-129
EN^DIU2: Data Dictionary Deletion....1-130
EN^DIWE: Text Editing 1-132
^DIWF: Form Document Print 1-135
EN1^DIWF: Form Document Print....1-137
EN2^DIWF: Form Document Print....1-139
^DIWP: Formatter....1-141
^DIWW: WP Print 1-143
%DT: Introduction to Date/Time Formats....1-144
^%DT: Internal to External Date 1-145
DD^%DT: Internal to External Date....1-151
^%DTC: Date/Time Utility....1-153
C^%DTC: Date/Time Utility 1-154
COMMA^%DTC: Date/Time Utility 1-155
DW^%DTC: Date/Time Utility 1-157
H^%DTC: Date/Time Utility....1-158
HELP^%DTC: Date/Time Utility 1-159
NOW^%DTC: Date/Time Utility....1-160
S^%DTC: Date/Time Utility 1-161
YMD^%DTC: Date/Time Utility 1-162
YX^%DTC: Date/Time Utility....1-163
%XY^%RCR: Array Moving 1-164

2. Database Server (DBS) API 2-1

Introduction....2-1

How to use the DBS calls.....2-2

Format and Conventions of the Calls.....2-2
IENS: To Identify Entries and Subentries.....2-3
FDA: Format of Data Passed to and from VA FileMan.....2-5
Documentation Conventions.....2-6

How the Database Server (DBS) communicates.....2-7

Overview.....2-7
How Information Is Returned.....2-7
Contents of Arrays.....2-8
Obtaining Formatted Text From The Arrays.....2-11
Cleaning Up the Output Arrays.....2-11
Example of Call to VA FileMan DBS.....2-12

DataBase Server Calls Cross-referenced by Category.....2-15

Database Server (DBS) Calls Presented in Alphabetical Order).....2-16

DELIX^DDMOD: Traditional Cross-reference Deleter.....2-17
DELIXN^DDMOD: New-Style Index Deleter.....2-21
BLD^DIALOG( ): DIALOG Extractor.....2-24
$$EZBLD^DIALOG( ): DIALOG Extractor (Single Line).....2-31
MSG^DIALOG( ): Output Generator.....2-33
FIND^DIC( ): Finder.....2-38
$$FIND1^DIC( ): Finder (Single Record).....2-65
LIST^DIC( ): Lister.....2-81
FIELD^DID( ): DD Field Retriever.....2-105
FIELDLST^DID( ): DD Field List Retriever.....2-108
FILE^DID( ): DD File Retriever.....2-109
FILELST^DID( ): DD File List Retriever.....2-112
$$GET1^DID( ): Attribute Retriever.....2-114
CHK^DIE( ): Data Checker.....2-117
FILE^DIE( ): Filer.....2-120
HELP^DIE( ): Helper.....2-125
$$KEYVAL^DIE( ): Key Validator.....2-129
UPDATE^DIE( ): Updater.....2-131
VAL^DIE( ): Validator.....2-140
VALS^DIE( ): Fields Validator.....2-145
WP^DIE( ): Word Processing Filer.....2-149
CLEAN^DILF: Array and Variable Clean-up.....2-151
$$CREF^DILF( ): Root Converter (Open to Closed Format).....2-152
DA^DILF( ): DA( ) Creator.....2-153
DT^DILF( ): Date Converter.....2-154
FDA^DILF( ): FDA Loader.....2-158
$$HTML^DILF( ): HTML Encoder/Decoder.....2-161
$$IENS^DILF( ): IENS Creator.....2-162

$$OREF^DILF( ): Root Converter (Closed to Open Format) 2-163
$$VALUE1^DILF( ): FDA Value Retriever (Single).....2-164
VALUES^DILF( ): FDA Values Retriever.....2-166
$$EXTERNAL^DILFD( ): Converter to External.....2-168
$$FLDNUM^DILFD( ): Field Number Retriever 2-174
PRD^DILFD( ): Package Revision Data Initializer 2-175
RECALL^DILFD( ): Recall Record Number 2-176
$$ROOT^DILFD( ): File Root Resolver 2-178
$$VFIELD^DILFD( ): Field Verifier 2-180
$$VFILE^DILFD( ): File Verifier 2-181
$$GET1^DIQ( ): Single Data Retriever.....2-182
GETS^DIQ( ): Data Retriever.....2-186

Part II: ScreenMan..... II-1

3. ScreenMan Forms.....3-1

Introduction.....3-1

Form Layout: Forms and Pages 3-1

Form Structure 3-1
Linking Pages of a Form 3-2

Features.....3-5

Displaying Multiples in Repeating Blocks 3-5
Form-Only Fields.....3-7
Relational Navigation: Forward Pointers.....3-9
Relational Navigation: Backward Pointers 3-12
Computed Fields.....3-13
The DDSBR Variable 3-17
The DDSSTACK Variable 3-18
Data Filing (When Is It Performed?) 3-19

Form Property Reference.....3-20

Form Properties 3-20
Page Properties.....3-22
Block Properties.....3-25
Field Properties 3-28

ScreenMan Menu Options 3-35

Edit/Create a Form.....3-35
Run a Form 3-35
Delete a Form 3-36
Purge Unused Blocks 3-37

Callable Routines.....3-39

Programmer Mode Utilities....3-39

^DDGF....3-39
CLONE^DDS....3-40
PRINT^DDS....3-42
RESET^DDS....3-43

4. ScreenMan Form Editor....4-1

Introduction....4-1

Invoking the Form Editor....4-1

Command Summary....4-2

Navigating on the Main Screen and Block Viewer Screen....4-2
Quick Page Navigation....4-3
Moving Screen Elements....4-3
Adding, Selecting, and Editing....4-4

The Main Screen....4-5

Exiting, Quitting, Saving, and Obtaining Help....4-6

The Block Viewer Screen....4-7

Navigating on the Form Editor Screens....4-8

Going to Another Page....4-9

Adding Pages, Blocks, and Fields....4-9

Adding Pages....4-9
Adding Blocks....4-10
Adding Fields....4-11

Selecting and Moving Screen Elements....4-11

Selecting Screen Elements....4-11
Moving Screen Elements....4-12

Editing Properties....4-14

Editing Field Properties....4-14
Editing Block Properties....4-17
Editing Page Properties....4-18
Editing Form Properties....4-20

Choosing Another Form....4-21

Deleting Screen Elements (Fields, Blocks, Pages, and Forms)....4-22

5. ScreenMan API 5-1

Introduction 5-1

Invoke ScreenMan 5-1

^DDS 5-1

Retrieve/Stuff Fields 5-5

$$GET^DDSVAL( ) 5-5
PUT^DDSVAL( ) 5-7
$$GET^DDSVALF( ) 5-10
PUT^DDSVALF( ) 5-12

Help Messages 5-14

HLP^DDSUTL( ) 5-14
MSG^DDSUTL( ) 5-15

Refresh Screen 5-16

REFRESH^DDSUTL( ) 5-16

Run-Time Field Status 5-17

REQ^DDSUTL( ) 5-17
UNED^DDSUTL( ) 5-18

Part III: Other APIs III-1

6. Browser API 6-1

Browser (DDBR) 6-1

EN^DDBR 6-1
BROWSE^DDBR 6-3
WP^DDBR 6-7
DOCLIST^DDBR 6-10
$$TEST^DDBRT 6-13
CLOSE^DDBRZIS 6-14
OPEN^DDBRZIS 6-15
POST^DDBRZIS 6-16

7. Import and Export Tools 7-1

Introduction 7-1

FILE^DDMP: Data Import 7-2
EXPORT^DDXP: Data Export 7-11

8. Extract Tool 8-1

Introduction 8-1

EN^DIAXU: Extract Data 8-1
EXTRACT^DIAXU: Extract Data 8-4

9. Filegrams API 9-1

Introduction 9-1

^DIFG: Installer 9-1
EN^DIFGG: Generator 9-5

Part IV: Developer Tools IV-1

10. ^DI: Programmer Access 10-1

11. Global File Structure 11-1

Introduction 11-1

Data Storage Conventions 11-1

File's Entry in the Dictionary of Files 11-1

File Header 11-2

Attribute Dictionary 11-4

File Characteristics Nodes 11-4
Field Definition 0-Node 11-8
Other Field Definition Nodes 11-11
How to Read the Attribute Dictionary: An Example 11-14
INDEX File 11-16
KEY File 11-16
File Entries (Data Storage) 11-16
Cross-references 11-17

12. Advanced File Definition 12-1

Introduction 12-1

File Global Storage 12-1

Storing Data in a Global other than ^DIZ 12-1

Field Global Storage 12-3

Assigning a Location for Fields Stored within a Global 12-3
Storing Data by Position within a Node 12-4

Assigning Sub-Dictionary Numbers....12-5

Computed Expressions 12-6

MUMPS Data Type....12-6

Screened Pointers and Set of Codes 12-6

INPUT Transform....12-7

INPUT Transforms and the Verify Fields Option....12-8

OUTPUT Transform 12-9

Special Lookup Programs 12-9

Post-Selection Action 12-10

Audit Condition....12-10

Editing a Cross-reference 12-11

Executable Help 12-11

13. Trigger Cross-references....13-1

Introduction....13-1

A Trigger on the Same File....13-2

Triggers for Different Files....13-4

14. DIALOG File....14-1

DIALOG File: User Messages 14-1

Introduction 14-1
Use of the DIALOG File 14-1
Creating DIALOG File Entries 14-2

Internationalization and the Dialog File 14-4

Role of the VA FileMan DIALOG File in Internationalization 14-4
Use of the DIALOG File in Internationalization 14-4
Creating Non-English Text in the DIALOG File 14-4
Example 14-5

VA FileMan LANGUAGE File 14-6

Introduction 14-6
Use of the LANGUAGE File 14-6

Creating LANGUAGE File Entries 14-7

15. VA FileMan Functions (Creating) 15-1

Introduction 15-1

Function File Entries 15-1

16. DIFROM 16-1

Introduction 16-1

Exporting Data 16-2

Order Entry and DIFROM 16-9

Running DIFROM (Steps 1-17) 16-11

Importing Data 16-19

DIFROM: Running an INIT (Steps 1-16) 16-19

Glossary Glossary-1

Appendix A—VA FileMan Error Codes ..... Appendix A-1

Index Index-1

Orientation

Installation of VA FileMan in the Veterans Health Information Systems and Technology Architecture (**V***IST***A**) environment is described in the *VA FileMan Installation Guide.*

New features and functionality are outlined in the *VA FileMan Release Notes* and are discussed at more length in this manual.

How to Use this Manual

Frequently, throughout this manual, you are shown a simulation of your interaction with the computer. In order to distinguish computer-supplied prompts from your responses, responses will be in bold type, like this:

```
COMPUTER'S PROMPT:  USER'S RESPONSE
```

The Return or Enter key, when it is shown, is shown as <RET>.

Assumptions About the Reader

This manual is written with the assumption that the reader is familiar with the **V***IST***A** computing environment. If you need more information, we suggest you look at the various VA home pages on the World Wide Web (WWW) for a general orientation to **V***IST***A**. You might want to begin here**:**

- **V***IST***A** Software Development Home Page: http://www.vista.med.va.gov/

Related Manuals and Other References

Readers who wish to learn more about VA FileMan should consult the VA FileMan Home Page at the following web address:

http://www.vista.med.va.gov/softserv/infrastr.uct/fileman/index.html

You may want to look at the following:

- *VA FileMan V. 22.0 Release Notes* (PDF format)
- *VA FileMan V. 22.0 Installation Guide* (PDF format)
- *VA FileMan V. 22.0 Technical Manual* (PDF format)
- *VA FileMan V. 22.0 Getting Started Manual* (PDF and HTML format)

  http://www.vista.med.va.gov/softserv/infrastr.uct/file/u1/index.html

- *VA FileMan V. 22.0 Advanced User Manual* (PDF and HTML format)

  http://www.vista.med.va.gov/softserv/infrastr.uct/fileman/u2/index.html

VA FileMan documentation is made available online (**HTML** format) and in Adobe Acrobat Portable Document Format (**.PDF**).

The **HTML** documents must be read using a web browser (e.g., Microsoft Explorer or Netscape Navigator, both of which are freely distributed). Using the web browser, open the "table of contents" page (i.e., INDEX.HTML). The distinguishing characteristic of manuals in this format are the *hypertext jumps* contained within the text. Clicking on a hypertext jump causes your browser to jump to the location or document described in the jump. The *VA FileMan Getting Started Manual*, the *VA FileMan Advanced User Manual*, and the *VA FileMan Programmer Manual* are all linked together so that you can jump from one to another easily.

The **.PDF** documents must be read using the Adobe Acrobat Reader (i.e., ACROREAD.EXE), which is also freely distributed by Adobe Systems Incorporated at the following web address:

http://www.adobe.com/

NOTE: For more information on the use of the Adobe Acrobat Reader, please refer to the "Adobe Acrobat Quick Guide" at the following web address:

http://www.vista.med.va.gov/softserv/infrastr.uct/acrobat/index.html

Programming Conventions

NOTE: A knowledge of the M programming language is presumed throughout this manual. VA FileMan V. 22.0 is written following the 1995 ANSI MUMPS standard, with several Type A extensions.

Nonstandard M Features

Z-commands and Z-functions are avoided throughout VA FileMan routines. For certain purposes [such as allowing terminal breaking and spooling to a Standard Disk Processor (SDP) disk device], FileMan executes lines of nonstandard M code out of the MUMPS OPERATING SYSTEM file (#.7). The nonstandard code used (if any) depends on the answer to the prompt:

```
TYPE OF MUMPS SYSTEM YOU ARE USING:
```

This prompt appears during the DINIT initialization routine. Answering OTHER to this question will ensure that VA FileMan uses only standard M code.

VA FileMan also makes use of nonstandard M code that is stored in the %ZOSF global. If FileMan is installed on a system that contains Kernel, it uses the %ZOSF global created by Kernel. If it is being used without Kernel (i.e., standalone), the necessary %ZOSF nodes are set for many operating systems by running DINZMGR in the manager account. See the "System Management" chapter of the *VA FileMan Advanced User Manual* for details.

String-valued subscripts (up to 30 characters long) are used extensively but only in the $ORDER collating sequence approved by the MUMPS Development Committee (MDC). Non-negative integer and fractional canonic numbers collate ahead of all other strings.

The $ORDER function is used at several points in VA FileMan's code. FileMan routines assume that reference to an undefined global subscript level sets the naked indicator to that level, rather than leaving it undefined. In all other respects, the FileMan code conforms to the 1995 ANSI Standard for the M language with Type A extensions.

Routine, Variable, and Global Names

In keeping with the convention that all programs which are a part of the same application or utility package should be namespaced, all VA FileMan routine names begin with DI or DD. (The "Device Handling for Standalone VA FileMan" section of the *VA FileMan Advanced User Manual* explains that some DI* routines are renamed in the management account.) The DINIT routine initializes FileMan. The DI routine itself is the main option reader (see the "^DI: Programmer Access" chapter in this manual).

Except in DI, the routines do not contain unargumented or exclusive KILL commands. All multi-character local variable names created by VA FileMan routines begin with % or the letter D, or consist of one uppercase letter followed by one numeral (except that IO(0), by convention, contains the $I value of the signon device). Since FileMan uses single character variable names extensively, do *not* use them in code that is executed from within FileMan programming hooks unless their use is documented in the hook's description or you New them. Also, do not expect single character variables to return unchanged after calling a FileMan entry point.

The following local variables are of special importance in the routines:

DT if defined, is assumed to be the current date. For example, June 1, 1987 is DT=2870601.

DTIME if defined, is the integer value of the number of seconds the user has to respond to a timed read.

DUZ if defined, is assumed to be the User Number, a positive number uniquely identifying the current user.

DUZ(0) if defined, is assumed to be the FileMan Access Code, a character string describing the user's security clearance with regard to files, to templates, and to data fields within a file. See the "Data Security" chapter in the *VA FileMan Advanced User Manual*. Setting DUZ(0) equal to the @-sign overrides all security checks and allows special programmer features which are described later. If the user's M implementation supports terminal break, a programmer is allowed to break execution at any point, whereas a user who does not have programmer access can only break during output routines.

U if defined, is equal to a single up-arrow (^) character.

If not defined, the default values set for these variables are:

DT today's date, derived from $H

DTIME 300

DUZ 0

DUZ(0) ""

U "^"

VA FileMan routines explicitly refer to the following globals:

^DD	All attribute dictionaries
^DDA	Data dictionary audit trail
^DI	Data types
^DIA	Data audit trail
^DIAR	Archival activity and Filegrams
^DIBT	Sort templates and the results of file searches
^DIC	Dictionary of files
^DIE	Input templates
^DIPT	Print templates and Filegram templates
^DIST	ScreenMan forms and blocks and Alternate Editors
^DISV	Most recent lookup value in any file or subfile (by DUZ)
^DIZ	Default location for new data files as they are created
^DOPT	Option lists
^DOSV	Statistical results
^%ZOSF	M vendor-specific executable code

The routines use the ^UTILITY and ^TMP globals for temporary scratch space. The ^XUTL global is also used if you are running some M implementations.

Delimiters within Strings

The up-arrow (^) character is conventionally used to delimit data elements which are strung together to be stored in a single global node. A corollary of this rule is that the routines almost never allow input data to contain up-arrows; the user types an up-arrow (^) to change or terminate the sequence of questions being asked. Within ^-pieces, semicolons are usually used as secondary delimiters and colons as tertiary delimiters.

VA FileMan routines use the local variable **U** as equal to the single up-arrow (^) character.

Canonic Numbers

VA FileMan recognizes only canonic numbers. A canonic number is a number that does **not** begin or end with meaningless zeroes. For example, 7 is a canonic number, whereas 007 and 7.0 are not.

Introduction

WHAT IS VA FILEMAN?

VA FileMan creates and maintains a database management system that includes features such as:

- A report writer
- A data dictionary manager
- Scrolling and screen-oriented data entry
- Text editors
- Programming utilities
- Tools for sending data to other systems
- File archiving

VA FileMan can be used as a standalone database, as a set of interactive or "silent" routines, or as a set of application utilities; in all modes, it is used to define, enter, and retrieve information from a set of computer-stored files, each of which is described by a data dictionary.

VA FileMan is a public domain software package that is developed and maintained by the Department of Veterans Affairs. It is widely used by VA medical centers and in clinical, administrative, and business settings in this country and abroad.

FUNCTIONAL DESCRIPTION

VA FileMan functions as a database management system with powerful Application Programmer Interfaces(APIs) and provides useful utilities to package application developers and programmers. VA FileMan can be used as a database management system for data entry and output and its DBS calls are utilized in application packages with tools like Filegrams, auditing, archiving, and statistics.

As a *database management system*, VA FileMan supports the entering, editing, printing, searching, inquiring, transferring, cross-referencing, triggering, and verifying of information. It includes special functions to create new files, modify an existing file, delete entire files, re-index files, and create or edit templates.

As an *application programmer interface*, the Database Server routines manage interactions between the application software and the database management system "silently," that is, without writing to the current device. Package developers use DBS calls to update the database in a non-interactive mode. Information needed by the FileMan routines is passed through parameters rather than through

interactive dialog with the user. Information to be displayed to the user is passed by FileMan back to the calling routine in arrays. This separation of data access from user interaction makes possible the construction of alternative front-ends to the FileMan database [e.g., a windowed Graphical User Interface (GUI)].

As a set of *utilities*, VA FileMan provides tools like the Filegram, which is a tool that moves file records from one computer to another; archiving, which is a tool that stores data onto an offline storage medium; auditing, which is a tool that tracks changes to data in a field or to the file's structure (the data dictionary); and statistics, which is a tool that accumulates totals and subtotals of individual fields.

VA FileMan has several levels of users, ranging from a data entry person who enters, edits, inquires or prints information, to a software application developer or Information Resource Management (IRM) staff member who uses all of its database management system features and utilities.

Programmers should consider this manual the list of VA FileMan-supported ("documented") routines and calls eligible for programmer use. These routines and calls provide the following (to list a few):

- File lookup and re-indexing
- Data edit, print, display, and retrieval
- Filegrams
- File entry deletion
- A reader program
- Data dictionary deletion
- Word processing
- Conversion of date and time values
- Software package export
- Linked option processing

STANDALONE VA FILEMAN

VA FileMan is designed to be used either with Kernel or as a standalone application running under a variety of implementations of ANSI standard M. If VA FileMan is used without Kernel, the basic DBMS features of VA FileMan all work as described in the manuals. However, there are some features (e.g., bulletin-type cross references, print queuing, and Filegrams) that do not work without portions of Kernel. Whenever Kernel is needed to support a particular VA FileMan feature, that fact is mentioned in the manuals.

The installation of VA FileMan V. 22.0 is not integrated with the installation of Kernel. The *VA FileMan Installation Guide* contains instructions on how to install FileMan, both for standalone sites and for sites running Kernel.

For specific information regarding standalone VA FileMan (i.e., device handling, setting IO variables, manually setting ^%ZOSF nodes, and setting up a minimal NEW PERSON file), please refer to the "FileMan System Management" section of the *VA FileMan Advanced User Manual*.

Part I: Major APIs

1. Classic VA FileMan API

INTRODUCTION

Certain modules within VA FileMan are callable by other M routines. This is true of the Classic FileMan routines which are referred to as "Callable Routines" and are described in this chapter.

Database Server (DBS) calls are also callable by other M routines. However, these "silent" calls differ from the Classic FileMan routines in that they separate interaction with the database from interaction with the end-user. In Classic FileMan's roll and scroll mode, interaction with the end-user was closely tied to the code that actually changed the database, but, with FileMan's DBS calls, no Writes to the current device are done. Package developers, calling FileMan from within their own code, manage interaction with the user whenever interaction with the database is needed. These DBS calls are described in the "Database Server (DBS)" chapter in this manual.

When using both the Classic FM callable routines and the DBS calls, you must keep in mind the variable-naming conventions described below. If you have local variables that you wish to preserve by a call to any of the routines described here, you should be sure to give them multi-character names beginning with letters other than D.

It is your responsibility, as a programmer, to clean up (Kill) documented input and output variables used in a FileMan call, when the call is finished. The few situations in which your input variables are killed during the FileMan call are mentioned in the following sections.

After making a programmer call, always check for failed calls. For example, when using ^DIC for lookups, always check for the error condition Y=-1 before doing anything else; when using the reader, always check DUOUT, DIRUT, and DTOUT before doing anything else. When a call provides a way to check for error conditions, it means that there are some circumstances where the call will not succeed! Checking for errors after such a call allows you to handle the errors gracefully.

CLASSIC CALLS CROSS-REFERENCED BY CATEGORY

Lookup/Adding Entries

^DIAC	File Access
^DIC	Lookup/Add
IX^DIC	Lookup/Add
MIX^DIC1	Lookup/Add
FILE^DICN	Add
DQ^DICQ	Entry Lists

Entry Editing

^DIE	Edit Data
EN^DIB	User Controlled
^DIK	Delete Entries
EN^DIQ1	Data Retrieval
EN^DIWE	Text Edit

Prompting/Messages

^DIR	Reader
EN^DDIOL	Msg Loader
WAIT^DICD	Wait Msgs
YN^DICN	Yes/No
HELP^%DTC	Date

Printing

EN1^DIP	Print
D^DIQ	Display
DT^DIQ	Display
EN^DIQ	Display
Y^DIQ	Display
EN^DIS	Search
^DIWF	Form Doc.
EN1^DIWF	Form Doc.
EN2^DIWF	Form Doc.
DIWP	WP Print
DIWW	WP Print

Templates

^DIEZ	Input/Compile
EN^DIEZ	Input/Compile
^DIOZ	Sort/Compile
^DIPT	PRINT/Display
DIBT^DIPT	Sort/Display
^DIPZ	PRINT/Compile
EN^DIPZ	PRINT/Compile

Cross-References

EN^DIK	Reindex
EN1^DIK	Reindex
ENALL^DIK	Reindex
IX^DIK	Reindex
IX1^DIK	Reindex
IXALL^DIK	Reindex
^DIKZ	Compile
EN^DIKZ	Compile

Date/Time Utilities

X ^DD("DD")
DT^DIO2
^%DT
DD^%DT
^%DTC
C^%DTC
DW^%DTC
H^%DTC
NOW^%DTC
S^%DTC
YMD^%DTC
YX^%DTC

Classic Calls Cross-referenced By Category (continued)

Utilities

DO^DIC1	File Info Setup
DT^DICRW	FM Var Setup
EN^DID	Print DDs
$$ROUSIZE^DILF	Routine Size
^DIM	M Validation
COMMA^%DTC	Number Format
EN^DIU2	DD Deletion
%XY^%RCR	Array Moving

CLASSIC CALLS PRESENTED IN ALPHABETICAL ORDER

This section lists and describes the VA FileMan Classic Calls in alphabetical order. The table above cross-references the Classic Calls by category.

X ^DD("DD"): Internal to External Date

Introduction to Date/Time Formats: %DT

This introduction pertains to this and the %DT calls.

%DT is used to validate date/time input and convert it to VA FileMan's conventional internal format: "YYYMMDD.HHMMSS", where:
YYY is number of years since 1700 (hence always 3 digits)
MM is month number (00-12)
DD is day number (00-31)
HH is hour number (00-23)
MM is minute number (01-59)
SS is the seconds number (01-59)

This format allows for representation of imprecise dates like JULY '78 or 1978 (which would be equivalent to 2780700 and 2780000, respectively). Dates are always returned as a canonic number (no trailing zeroes after the decimal).

There are two ways to convert a date from internal YYYMMDD format to external format—this call and DD^%DT. (This is the reverse of what %DT does.) Simply set the variable Y equal to the internal date and execute ^DD("DD").

Example

```
>S Y=2690720.163 X ^DD("DD") W Y
JUL 20,1969@1630
```

This results in Y being equal to JUL 20,1969@16:30. (No space before the 4-digit year.)

Input Variable

Y	(Required) This contains the internal date to be converted. If this has five or six decimal places, seconds will automatically be returned.

Output Variable

Y	Y is returned as the external form of the date.

See also DT^DIO2, which takes an internal date in the variable Y and *writes out* its external form.

EN^DDIOL: Loader

EN^DDIOL is designed as a replacement for simple WRITE statements in any part of the data dictionary that has a programming 'hook', such as executable help. As alternate user interfaces are developed for accessing VA FileMan databases, developers are faced with the issue of removing embedded WRITE statements from their data dictionaries. Direct writes should be removed since they might cause the text to display improperly in the new interface. This separation of the user interface from the database definition helps you to prepare your databases for access by any new interface, such as a Graphical User Interface (GUI).

The environment in which the Loader is called determines how it processes the text it is passed.

Mode	How the Text Is Processed
Scrolling mode	Text is written to the screen
ScreenMan mode	Text is written in ScreenMan's Command Area
DBS mode	Text is loaded into an array

In DBS mode, the specific array where the text is placed depends on which DBS call is made and whether an output array was specified in the DBS call. For example, if a call is made to the Validator (VAL^DIE), and the INPUT transform of the field makes a call to the Loader, the text is placed into ^TMP("DIMSG",$J). If a call is made to the Helper (HELP^DIE), and the executable help of the field makes a call to the Loader, the text is placed into ^TMP("DIHELP",$J). If the call to Validator or the Helper uses the MSG_ROOT parameter, the text is placed in the array specified by MSG_ROOT.

Recommendation: no line of text passed to the Loader should exceed 70 characters in length.

Formats

```
EN^DDIOL(VALUE,"",FORMAT)

EN^DDIOL(.ARRAY)

EN^DDIOL("",GLOBAL_ROOT)
```

Input Parameters

VALUE (Optional) If there is just one line of text to output, it can be passed in the first parameter.

.ARRAY (Optional) If there is more than one line of text to output, stored in a local array, then the first parameter of the call is the name of the local array passed by reference and that contains string or numeric literals, where:

```
ARRAY(1) = string 1
ARRAY(2) = string 2 ...
ARRAY(n) = string n
```

Formatting instructions can also be included in this array. See Formatting for Arrays in *Details and Features* below.

GLOBAL_ROOT (Optional) An alternate way to pass the text to the call is in a global root. In that case, the first parameter is null, and the second parameter contains the name of the global root that contains string or numeric literals, where:

```
@GLOBAL_ROOT@(1,0) = string 1
@GLOBAL_ROOT@(2,0) = string 2 ...
@GLOBAL_ROOT@(n,0) = string n
```

or

```
@GLOBAL_ROOT@(1) = string 1
@GLOBAL_ROOT@(2) = string 2 ...
@GLOBAL_ROOT@(n) = string n
```

Formatting instructions can also be included in this global array. See Formatting for Arrays in *Details and Features* below.

FORMAT (Optional) Formatting instructions controlling how the string is written or placed in the array. You can specify:

- One or more new lines before the string (!, !!, !!!, etc.)
- Horizontal position of string (?n)

FORMAT can be any number of "!" characters optionally followed by "?n", where n is an integer expression. The default FORMAT is "!".

This parameter can only be used when call format is used to pass a *single* string or numeric literal to EN^DDIOL. To pass formatting instructions when text is passed as an *array* or *global*

to EN^DDIOL, see Formatting for Arrays in *Details and Features* below.

Examples

Example 1

Suppose a Write Identifier node contains the following WRITE statement:

```
^DD(filenumber,0,"ID","W1")=W "   ",$P(^(0),U,2)
```

An equivalent statement converted to use EN^DDIOL is:

```
^DD(filenumber,0,"ID","W1")=D EN^DDIOL("   "_$P(^(0),U,2),"","?0")
```

Example 2

The executable help of a field passes one line of text by value to the Loader as illustrated below:

```
>D EN^DDIOL("This is one line of text.","","!!?12")
```

If the call is made in scroll mode (e.g., ^DIE executes the executable help), below is an example of what the Loader writes to the screen:

```
This is one line of text.
```

If the call is made in DBS mode, the Helper (HELP^DIE) executes the executable help. The text is placed into the ^TMP global as shown below:

```
^TMP("DIHELP",$J,1)=""
^TMP("DIHELP",$J,2)="        This is one line of text."
```

Example 3

Below is an example of passing an array of text to the Loader:

```
>S A(1)="First line."
>S A(2)="Second line, preceded by one blank line or node."
>S A(2,"F")="!!"
>S A(3)="More text on second line."
>S A(3,"F")="?55"
>D EN^DDIOL(.A)
```

Example 4

Below is an example of passing a global that contains text to the Loader:

```
>S ^GLB(1)="First line."
>S ^GLB(2)="Second line, preceded by one blank line or node."
>S ^GLB(2,"F")="!!"
>S ^GLB(3)="More text on second line."
>S ^GLB(3,"F")="?55"
>D EN^DDIOL("","^GLB")
```

Details and Features

Formatting for Arrays

When you pass an array or a global to EN^DDIOL, you can also pass formatting instructions for each line of text in your array or global. These instructions control how the string is written or placed in the output array. You can specify:

- One or more new lines before the string (!, !!, !!!, etc.)
- Horizontal position of string (?n)

Place the formatting instructions for a line of text in an "F" node descendent from the node containing the text. The value of each "F" node can be any number of "!" characters optionally followed by "?n", where n is an integer expression. The default FORMAT is "!".

For example:

```
A(1) = string 1
A(1,"F") = format (e.g., "!?35", "?10", etc.)
^G(1,0) = string 1

^G(1,"F") = format


^G(1) = string 1
^G(1,"F") = format
```

NOTE: If you use format (1) to pass a single string of text to EN^DDIOL, you can pass the formatting instructions in the third parameter FORMAT.

^DIAC: File Access Determination

This entry point determines if a user has access to a file.

Input Variables

DIFILE (Required) The file number of the file on which you want to verify file access.

DIAC (Required) Use one of the values listed below to verify the specified type of file access:

"**RD**"	Verify Read access to a specific file.
"**WR**"	Verify Write access to a specific file.
"**AUDIT**"	Verify Audit access to a specific file.
"**DD**"	Verify DD access to a specific file.
"**DEL**"	Verify Delete access to a specific file.
"**LAYGO**"	Verify LAYGO access to a specific file.

Output Variables

DIAC DIAC returns either a 0 or a 1:

1 Indicates that the user has that type of access to the file.

NOTE: If the user's DUZ(0)="@", the value 1 is always returned.

0 Indicates that the user does not have access of that type to the file.

% The % variable returns exactly the same values as DIAC.

EN^DIB: User Controlled Editing

Invokes the Enter or Edit File Entries option of VA FileMan to edit records in a given file, allowing the user to select which fields to edit.

Input Variables

DIE (Required) The global root of the file in the form ^GLOBAL(or ^GLOBAL(# or the number of the file.

DIE("NO^") (Optional) Allows the programmer control of the use of the up-arrow in an edit session. If this variable does not exist, unrestricted use of the up-arrow for jumping and exiting is allowed.

The variable may be set to one of the following:

"**OUTOK**"	Allows exiting and prevents all jumping.
"**BACK**"	Allows jumping back to a previously edited field and does not allow exiting.
"**BACKOUTOK**"	Allows jumping back to a previously edited field and allows exiting.
"**Other value**"	Prevents all jumping and does not allow exiting.

DIDEL (Optional) Allows you to override the Delete Access on a file or subfile. Setting DIDEL equal to the number of the file before calling DIE allows the user to delete an entire entry from that file even if the user does not normally have the ability to delete. This variable does not override the DEL-nodes described in the Global File Structure chapter.

^DIC: Lookup

Given a lookup value, this entry point searches the requested Index(es) on a file and either finds a matching entry, adds an entry to the file, or returns a condition indicating that the lookup was unsuccessful.

See also IX^DIC and MIX^DIC1 for a comparison of how they each perform lookups.

Except for the DIC("W") variable, which is killed, the DIC input array is left unchanged by ^DIC.

Input Variables

DIC

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

DIC(0)

(Optional) A string of alphabetic characters which alter how DIC responds. At a minimum this string must be set to null. A detailed description of these characters can be found later in this section, under DIC(0) Input Variables in Detail.

NOTE: If DIC(0) is null or undefined, no terminal output will be generated by the DIC routine.

The acceptable characters are:

Flag	Short Description
A	**A**sk the entry; if erroneous, ask again.
B	Only the **B** index is used when doing lookup to files pointed-to by starting file.
C	**C**ross-reference suppression is turned off.
E	**E**cho information.
F	**F**orget the lookup value.

I	**I**gnore the special lookup program.
K	Primary **K**ey is used as starting index for the lookup.
L	**L**earning a new entry is allowed.
M	**M**ultiple-index lookup allowed.
N	Internal **N**umber lookup allowed (but not forced).
O	**O**nly find one entry if it matches exactly.
Q	**Q**uestion erroneous input (with two ??).
S	**S**uppresses display of .01 (except B cross-reference match) and of any Primary Key fields.
T	Con**T**inue searching all indexes until user selects an entry or enters ^^ to get out.
U	**U**ntransformed lookup.
V	**V**erify that looked-up entry is OK.
X	E**X**act match required.
Z	**Z**ero node returned in Y(0) and external form in Y(0,0).

X If DIC(0) does not contain an A, then the variable X must be defined equal to the value you want to find in the requested Index(es).

If the lookup index is compound (i.e., has more than one data subscript), then X can be an array X(n) where "n" represents the position in the subscript. For example, if X(2) is defined, it will be used as the lookup value to match to the entries in the second subscript of the index. If only the lookup value X is passed, it will be assumed to be the lookup value for the first subscript in the index, X(1).

DIC("A")

(Optional) A prompt that is displayed prior to the reading of the X input. If DIC("A") is not defined, the word Select, the name of the file, [i.e., $P(^GLOBAL(0),"^",1)], a space, the LABEL of the .01 field, and a colon will be displayed. If the file name is the same as the LABEL of the .01 field, then only the file name will be displayed. DIC(0) must contain an A for this prompt to be issued. For example, if the EMPLOYEE file had a .01 field with the LABEL of NAME, then FileMan would issue the following prompt:

```
Select EMPLOYEE NAME:
```

By setting DIC("A")="Enter Employee to edit: ", the prompt would be:

```
Enter Employee to edit:
```

Notice that it is necessary for the prompt in DIC("A") to include the colon and space at the end of the prompt if you want those to be displayed.

If the lookup index is compound (i.e., has more than one data subscript), then DIC("A") can be an array DIC("A",n) where "n" represents the position in the subscript. For example, DIC("A",2) will be used as the prompt for the second subscript in the index. If only the single prompt DIC("A") is passed, it will be assumed to be the prompt for the first subscript in the index DIC("A",1).

If DIC("A",n) is undefined for the "**n**th" subscript, then the "Lookup Prompt" field for that subscript from the INDEX file will be used as the prompt, or if it is null, the LABEL of the field from the data dictionary.

DIC("B")

(Optional) The default answer which is presented to the user when the lookup prompt is issued. If a terminal user simply presses the Enter/Return key, the DIC("B") default value will be used, and returned in X. DIC("B") will only be used if it is non-null.

If the lookup index is compound (i.e., has more than one data subscript), then DIC("B") can be an array DIC("B",n) where "**n**" represents the position in the subscript. For example, DIC("B",2) will be used as the default answer for the prompt for the second subscript in

the index. If only the single default answer DIC("B") is passed, it will be assumed to be the default answer for the prompt for the first subscript in the index DIC("B",1).

DIC("DR")

When calling DIC with LAYGO allowed, you can specify that a certain set of fields will be asked for in the case where the user enters a new entry. This list is specified by setting the variable DIC("DR") equal to a string that looks exactly like the DR string of fields that is specified when calling ^DIE. Such a list of what VA FileMan calls forced identifiers overrides any identifiers that would normally be requested for new entries in this file.

DIC("P")

NOTE: As of Version 22 of FileMan, the developer is no longer required to set DIC("P"). The only exception to this is for a few files that are not structured like a normal FileMan file, where the first subscript of the data is variable in order to allow several different 'globals' to use the same DD. An example of this is the FileMan Audit files where the first subscript is the file number of the file being audited.

This variable is needed to successfully add the FIRST subentry to a multiple when the descriptor (or header) node of the multiple does not exist. In that situation, DIC("P") should be set equal to the subfile number and subfile specifier codes for the multiple. (See the File Header section of the Global File Structure chapter.) If the descriptor node for the multiple already exists, DIC("P") has no effect.

In order to automatically include any changes in the field's definition in DIC("P"), it is best to set this variable to the second ^-piece of the 0-node of the multiple field's definition in the DD. (See the Field Definition section of the Global File Structure chapter.)

Thus, for example, if file 16150 had a multiple field #9, set DIC("P") like this:

```
S DIC("P")=$P(^DD(16150,9,0),"^",2)
```

For more information, see Adding New Subentries to a Multiple below.

DIC("PTRIX",f,p,t)=d

DIC("PTRIX",f,p,t)=d where

f is the from (pointing) file number,

p is the pointer field number,

t is the pointed-to file number, and

d is an "^" delimited list of index names.

When doing a lookup using an index for a pointer or variable pointer field, this new array allows the user to pass a list of indexes that will be used when searching the pointed-to file for matches to the lookup value. For example, if your file (662001) has a pointer field (5) to file 200 (NEW PERSON), and you wanted the lookup on file 200 to be either by name ("B" index), or by the first letter of the last name concatenated with the last 4 digits of the social security number ("BS5" index): DIC("PTRIX",662001,5,200)="B^BS5". Note that if the call allows records to be added to a pointed-to file, then the list in the "PTRIX" entry should contain the "B" index. However, the "B" index would not need to be included in the list if the first index in the "PTRIX" array entry is a compound index whose first subscript is the .01 field.

DIC("S")

(Optional) DIC("S") is 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:

```
S DIC="^EMP(",DIC(0)="QEZ",X="SMITH"
S DIC("S")="I $P(^(0),U,2)=""M"""
D ^DIC
```

DIC("T")

(Optional) Present every match to the lookup value, quitting only when user either selects one of the presented entries, enters ^^ to quit, or there are no more matching entries found.

Currently, if one or more matches are found in the first pass through the indexes, then FileMan quits the search, whether or not one of the entries is selected. Only if no matches are found in the first pass does FileMan continue on to try transforms to the lookup value. This includes transforms to find internal values of pointers, variable pointers, dates or sets.

Another feature of the “T” flag is that indexes are truly searched in the order requested. If, for example, an index on a pointer field comes before an index on a free-text field, matches from the pointer field will be presented to the user before matches to the free-text field.

When used in combination with the “O” flag, all indexes will be searched for an exact match. Then, only if none are found, will FileMan make a second pass through the indexes looking for partial matches.

DIC("V")

If the .01 field is a variable pointer, it can point to entries in more than one file. You can restrict the user's ability to input entries from certain files by using the DIC("V") variable. It is used to screen files from the user. Set the DIC("V") variable to a line of M code that returns a truth value when executed. The code is executed after someone enters data into a variable pointer field. If the code tests false, the user's input is rejected; FileMan responds with ?? and a "beep."

If the lookup index is compound (i.e., has more than one data subscript), and if any of the subscripts index variable pointer fields, then DIC("V",n) can be passed where "n" represents the subscript position of the variable pointer field in the index. For example, if DIC("V",2) is passed in, it will be used as the screen for

files pointed-to by the variable pointer field indexed in the second subscript of the index. If only the entry DIC("V") is passed, it will be assumed to be the variable pointer file screen for the first subscript in the index, DIC("V",1).

When the user enters a value at a variable pointer field's prompt, VA FileMan determines in which file that entry is found. The variable Y(0) is set equal to information for that file from the data dictionary definition of the variable pointer field. You can use Y(0) in the code set into the DIC("V") variable. Y(0) contains:

^-Piece	**Contents**
Piece 1	File number of the pointed-to file.
Piece 2	Message defined for the pointed-to file.
Piece 3	Order defined for the pointed-to file.
Piece 4	Prefix defined for the pointed-to file.
Piece 5	y/n indicating if a screen is set up for the pointed-to file.

All of this information was defined when that file was entered as one of the possibilities for the variable pointer field.

For example, suppose your .01 field is a variable pointer pointing to files 1000, 2000, and 3000. If you only want the user to be able to enter values from files 1000 or 3000, you could set up DIC("V") like this:

```
S DIC("V")="I +Y(0)=1000!(+Y(0)=3000)"
```

DIC("W")

(Optional) An M command string which is executed when DIC displays each of the entries that match the user's input. The condition of the variable Y and of the naked indicator is the same as for DIC("S"). If DIC("W") is defined, it overrides the display of any identifiers of the file. Thus, if DIC("W")="", the display of identifiers will be suppressed.

NOTE: DIC("W") is killed by ^DIC calls.

DIC("?N",file#) =n

(Optional) The number "**n**" should be an integer set to the number of entries to be displayed on the screen at one time when using "**?**" help in a lookup. Usually, file# will be the number of the file on which you're doing the lookup. However, if doing a lookup using an index on a pointer field, and if DIC(0) contains "L", then the user also is allowed to see a list of entries from the pointed-to file, so in that case file# could be the number of that pointed-to file. For example, when doing a lookup in test file 662001, if the developer wants only five entries at a time to be displayed in question-mark help, set DIC("?N",662001)=5

DIC("?PARAM", file#,"INDEX")= Index name

(Optional) Used to control entries displayed during online "**?**" help only. If provided, this index will be used to display the entries from the file specified by file#. Otherwise, VA FileMan uses the first lookup index specified for the ^DIC call. This value is used as the INDEX parameter to the Lister call to display the entries. See documentation for LIST^DIC for more information.

DIC("?PARAM", file#,"FROM",n) =value

(Optional) Used to control entries displayed during online "**?**" help only. This array can be set to define a starting value for an entry in the lookup index used to list entries from the file. Integer value "**n**" is associated with the "**n**th" data value subscript in the index (e.g., regular old-style indexes always have just one indexed data value so "**n**" would be 1). If a starting value is defined for subscript "**n**," then starting values must also be defined for all of the subscripts preceding "**n**."

This information is used to set the FROM parameter for a call to LIST^DIC in order to display the entries in the file specified by file#. Therefore, the entries must meet the same rules as the FROM parameter described in that call. See documentation for LIST^DIC for detailed information.

If DIC(0) contains an "L" and the first indexed field is a pointer, then after displaying the current entries on the file, VA FileMan allows the user to see entries on the pointed-to file. In that case, the developer may request starting values for any pointed-to file in the pointer chain. If the user enters "^value" when asked whether

they wish to see the entries in the file, the value entered by the user will override the starting list value passed by the developer in this array.

DIC("?PARAM", file#,"PART",n) =value

(Optional) Used to control entries displayed during online"**?**" help only. This array can be set to define partial match value(s) for each of the "**n**" subscripts on the lookup index used during online help. The information is used to set the PART parameter for a Lister call to display the entries. See documentation for LIST^DIC for more information. As with DIC("?PARAM",file#,"FROM",n), if DIC(0) contains "L", the developer can define partial match values for any pointed-to file in the pointer chain.

DLAYGO

(Optional) If this variable is set equal to the file number, then the users will be able to add a new entry to the file whether or not they have LAYGO access to the file. This variable, however, does not override any LAYGO node that may exist on the .01 field, i.e. ^DD(file#,.01,"LAYGO",#,0). M code in the LAYGO node will still be executed, and must set the truth value to TRUE for an entry to be added.

NOTE: In addition, DIC(0) must contain L to allow addition of entries to the file.

Output Variables

Y

DIC always returns the variable Y. The variable Y is returned with one of these three formats:

Y=-1 The lookup was unsuccessful.

Y=N^S N is the internal number of the entry in the file and S is the value of the .01 field for that entry.

Y=N^S^1 N and S are defined as above and the 1 indicates that this entry has just been added to the file.

Y(0) This variable is only set if DIC(0) contains a Z. When the variable is set, it is equal to the entire zero node of the entry that was selected.

Y(0,0) This variable also is only set if DIC(0) contains a Z. When the variable is set, it is equal to the external form of the .01 field of the entry.

The following are examples of returned Y variables based on a call to the EMPLOYEE file stored in ^EMP(:

```
S DIC="^EMP(",DIC(0)="QEZ",X="SMITH"
D ^DIC
```

Returned are:

```
Y      = "7^SMITH,SAM"
Y(0)   = "SMITH,SAM^M^2231109^2
Y(0,0) = "SMITH,SAM"
```

If the lookup had been done on a file whose .01 field points to the EMPLOYEE file, the returned variables might look like this:

```
Y      = "32^7"  [ Entry #32 in this file and #7
                 in EMPLOYEE file.]
Y(0)   = "7^RX 2354^ON HOLD"
Y(0,0) = "SMITH,SAM"  [.01 field of entry 7 in
                       EMPLOYEE file]
```

X Contains the value of the field where the match occurred.

If the lookup index is compound (i.e., has more than one data subscript), and if DIC(0) contains "A" so that the user is prompted for lookup values, then X will be output as an array X(n) where "n" represents the position in the subscript and will contain the values from the index on which the entry was found. Thus, X(2) would contain the value of the second subscript in the index. If possible, the entries will be output in their external format (i.e., if the subscript is not computed and doesn't have a transform). If the entry is not found on an index (example, when lookup is done with X=" " (the space-bar return feature)), then X and X(1) will contain the user input, but the rest of the X array will

be undefined.

DTOUT This is only defined if DIC has timed-out waiting for input from the user.

DUOUT This is only defined if the user entered an up-arrow.

DIC(0) Input Variables in Detail

The effects of the various characters which can be contained in DIC(0) are described below:

A DIC asks for input from the terminal and asks again if the input is erroneous. A response of null or a string containing ^ is accepted. Input is returned in X when DIC quits. If DIC(0) does not contain the character A, the input to DIC is assumed to be in the local variable X.

B Without the B flag, if there are cross-referenced pointer or variable pointer fields in the list of indexes to use for lookup and if DIC(0) contains "M" and there is no screening logic on the pointer that controls the lookup on the pointed-to file, then:

1. For each cross-referenced pointer field, FileMan checks ALL lookup indexes in each pointed-to file for a match to X (time-consuming);

2. If X matches any value in any lookup index (not just the "B" index) on the pointed-to file and the IEN of the matched entry is in the home file's pointer field cross-reference, FileMan considers this a match. This may perhaps not be the lookup behavior you wanted, see Examples section.

The B flag prevents this behavior by looking for a match to X only in the B index (.01 field) of files pointed to by cross-referenced pointer or variable pointer fields. This makes lookups quicker and avoids the risk of FileMan matching an entry in the pointed-to file based on some unexpected indexed field in that file.

C Normally, when DIC does a lookup and finds an entry that matches the input, that entry is presented to the user only once even if the entry appears in more than one cross-reference. This is called cross-reference suppression and can be

overridden by including a C in DIC(0). If, for example, a person with the name ZACHARY,DAVID is an entry in a file, then his name will appear in the B cross-reference of the file. If he has a nickname of ZACH, which is in the C cross-reference of the file, then when a user enters ZACH as a lookup value, the name, ZACHARY,DAVID, will appear only once in the choices. But if there is a C in DIC(0), then ZACHARY,DAVID will appear twice in the choices; once as a hit in the B cross-reference and again as a hit in the C cross-reference.

F Prevents saving the entry number of the matched entry in the ^DISV global. Ordinarily, the entry number is saved at ^DISV(DUZ,DIC). This allows the user to do a subsequent lookup of the same entry simply by pressing the space bar and Enter/Return key. To avoid the time cost of setting this global, include an F in DIC(0).

I If DIC(0) contains I, any special user-written lookup program for a file will be ignored and DIC will proceed with its normal lookup process.

You can write a special lookup program to be used to find entries in a particular file. This special program can be defined by using the Edit File option of the Utility Functions submenu (see the Special Lookup Programs section in the Advanced File Definition chapter.) When a lookup program is defined, VA FileMan will bypass the normal lookup process of DIC and branch to the user written program. This user written lookup program must respond to the variables documented in this section and provide the functionality of DIC as they pertain to the file.

K This flag causes ^DIC to use the Uniqueness index for the Primary Key as the starting index for the lookup, rather than starting with the B index. (If developers want to specify some other index as the starting index, then they can specify the index by using the "D" input variable, and either the IX^DIC or the MIX^DIC1 call instead of ^DIC.)

L If DIC(0) contains L and the user's input is in valid format for the file's .01 field, then DIC will allow the user to add a new entry to the file at this point (Learn-As-You-GO), as long as at least one of these four security-check conditions is true:

The local variable DUZ(0) is equal to the @-sign.

If Kernel's File Access Security System (formerly known as Kernel Part 3) is being used for security, the file is listed in the user's record of accessible files with LAYGO access allowed.

If file access management is not being used, a character in DUZ(0) matches a character in the file's LAYGO access code or the file has no LAYGO access code.

The variable DLAYGO is defined equal to the file number.

NOTE: Even if DIC(0) contains L and one of these security checks is passed, LAYGO will not be allowed if a test in the data dictionary's LAYGO node fails.

M If DIC(0) contains M, DIC will do a multiple lookup on all of the file's cross-references from B on to the end of the alphabet. For example, if a given file is cross-referenced both by Name and by Social Security Number, and the user inputs 123-45-6789, DIC, failing to find this input as a Name, will automatically go on to look it up as a Social Security Number.

NOTE: For finer control in specifying the indexes used for lookup, see the alternate lookup entry points IX^DIC and MIX^DIC1.

N If DIC(0) contains N, the input is allowed to be checked as an internal entry number even if the file in question is not normally referenced by number. However, input is only checked as an IEN if no other matches are found during regular lookup.

If DIC(0) does not contain an N, the user is still allowed to select by entry number by preceding the number with the accent grave character (` ). When a ` is used, the lookup is limited to internal entry numbers only.

Placing N in DIC(0) does not force IEN interpretation; it only permits it. In order to force IEN interpretation, you must use the accent grave (`) character.

NOTE: With this flag, when DIC(0) contains an L, users may be allowed to force the internal entry number when adding new entries to the file. If the user enters a number N that is not found on any of the cross-references, and if the .01 field is not numeric and the file is not DINUMed, and if FileMan can talk to the users (DIC(0)["E"), then the user will be asked

whether they want to add the new entry, and will be prompted for the value of the .01 field. The entry will be added at the record number N that was originally entered by the user. Note that if there is a .001 field on the file, the number N must also pass the INPUT transform for the .001 field.

O If DIC(0) contains the letter O, then for each index searched, FileMan looks first for exact matches to the lookup value before looking for partial matches. If an exact match is found, then FileMan returns only that match and none of the partial matches on the index. Thus if an index contained the entries 'SMITH,SAM' and 'SMITH,SAMUEL' and if the user typed a lookup value of 'SMITH,SAM', then only the 'SMITH,SAM' entry would be selected, and the user would never see the entry 'SMITH,SAMUEL'. Note that if partial matches but no exact matches are found in the first index(es) searched, but if exact matches are found in an index searched later, then the partial matches from the first index(es) are returned along with the exact match from the later index(es).

Q If DIC(0) contains Q and erroneous input is entered, two question marks (??) will be displayed and a "beep" will sound.

S If DIC(0) does not contain S, the value of the .01 field and Primary Key fields (if the file has a Primary Key) will be displayed for all matches found in any cross-reference. If DIC(0) does contain S, the .01 field and Primary Key fields will not be displayed unless they are one of the indexed fields on which the match was made.

T "T flag in DIC(0). Present every match to the lookup value, quitting only when user either selects one of the presented entries, enters ^^ to quit, or there are no more matching entries found.

Currently, if one or more matches are found in the first pass through the indexes, then FileMan quits the search, whether or not one of the entries is selected. Only if no matches are found in the first pass does FileMan continue on to try transforms to the lookup value. This includes transforms to find internal values of pointers, variable pointers, dates or sets.

Another feature of the "T" flag is that indexes are truly searched in the order requested. If, for example, an index on a pointer field comes before an index on a free-text field, matches

from the pointer field will be presented to the user before matches to the free-text field. When used in combination with the "O" flag, all indexes will be searched for an exact match. Then, only if no matches are found, will FileMan make a second pass through the indexes looking for partial matches.

U Normally the lookup value is expected to be in external format (for dates, pointers and such). FileMan first searches the requested index for a match to the user input as it was typed in. Then, if no match is found, FileMan automatically tries certain transforms on the lookup value.

For instance, if one of the lookup indexes is on a date field, FileMan tries to transform the lookup value to an internal date, then checks the index again. The U flag causes FileMan to look for an exact match on the index and to skip any transforms. Thus the lookup value must be in internal format. This is especially useful for lookups on indexed pointer fields, where the internal entry number (i.e., internal pointer value) from the pointed-to file is already known.

Ordinarily this flag would not be used along with the "A", "B", "M", "N" or "T" flags. In many cases it makes sense to combine this with the "X" flag.

V If DIC(0) contains V and only one match is made to the user's lookup value, then they will be asked "OK?" and they will have to verify that the looked-up entry is the one they wanted. This is an on the fly way of getting behavior similar to the permanent flag that can be set on a file by answering "YES" to the question "ASK 'OK' WHEN LOOKING UP AN ENTRY?" (See the EDIT FILE option within the FileMan UTILITY option, described in the Advanced User Manual).

X If DIC(0) contains X, for an exact match, the input value must be found exactly as it was entered. Otherwise, the routine will look for any entries that begin with the input X. Unless 'X-act match' is specified, lowercase input that fails in the lookup will automatically be converted to uppercase, for a second lookup attempt. The difference between X and O (described above) is that X requires an exact match. If there is not one, either DIC exits or tries to add a new entry. With O, if there is not an exact match, DIC looks for a partial match beginning with the input.

Z If DIC(0) contains Z and if the lookup is successful, then the variable Y(0) will also be returned. It will be set equal to the entire zero node of the entry that has been found. Another array element, Y(0,0), is also returned and will be set equal to the printable expression of the .01 field of the entry selected. This has no use for Free Text and Numeric data types unless there is an OUTPUT transform. However, for Date/Time, Set of Codes and Pointer data types, Y(0,0) will contain the external format.

Adding New Subentries to a Multiple

You can use ^DIC or FILE^DICN to add new subentries to a multiple. In order to add a subentry, the following variables need to be defined:

DIC Set to the full global root of the subentry. For example, if the multiple is one level below the top file level: file's_root,entry#,multiple_field's_node,

DIC(0) Must contain "L" to allow LAYGO.

DIC("P") Set to the 2nd piece of 0-node of the multiple field's DD entry. **NOTE:** As of Version 22 of FileMan, the developer is no longer required to set DIC("P"). The only exception to this is for a few files that are not structured like a normal FileMan file, where the first subscript of the data is variable in order to allow several different 'globals' to use the same DD. An example of this is the FileMan Audit files where the first subscript is the file number of the file being audited.

DA(1)...

DA(n) Set up this array such that DA(1) is the IEN at the next higher file level above the multiple that the lookup is being performed in, DA(2) is the IEN at the next higher file level (if any), ... DA(n) is the IEN at the file's top level.

NOTE: The value of the unsubscripted DA node should not be defined when doing lookups in a subfile—that's the value you're trying to obtain!

A.) Below is an example of code that:

1. Uses ^DIC to interactively select a top-level record.
2. Uses ^DIC to select or create a **subentry** in a multiple in that record.
3. Uses ^DIE to edit fields in the selected or created subentry.

The file's root in this example is '^DIZ(16150,', the multiple's field number is 9, and the multiple is found on node 4. The code for this example follows:

```
; a call is made to DIC so the user can select an entry in the file
;
S DIC="^DIZ(16150,",DIC(0)="QEAL" D ^DIC
I Y=-1 K DIC Q  ;quit if look-up fails
;
; a second DIC call is set up to select the subentry
;
S DA(1)=+Y ;+Y contains the internal entry number of entry chosen
S DIC=DIC_DA(1)_",4," ;the root of the subfile for that entry
S DIC(0)="QEAL" ;LAYGO to the subfile is allowed
S DIC("P")=$P(^DD(16150,9,0),"^",2) ;returns the subfile# and specifiers
D ^DIC I Y=-1 K DIC,DA Q  ;user selects or adds subentry
;
; a DIE call is made to edit fields in subfile
;
S DIE=DIC K DIC ;DIE now holds the subfile's root
S DA=+Y ;+Y contains the internal entry number of subentry chosen
S DR="1;2" D ^DIE ;edit fields number 1 and 2
K DIE,DR,DA,Y Q
```

B) File #662002 has a .01 field that points to the NEW PERSON file (#200). In this example, we'll use input arrays in DIC("?PARAM",662002,"FROM",1) to start the list of entries in the "B" index of File #662002 with the letter "M". Since DIC(0) contains "L" (user can add entries to the pointed-to File #200), VA FileMan will also display entries from File #200, so we use DIC("?PARAM",200,"PART",1) to display only entries that start with the letter "S".

```
>S DIC=^DIZ(662002,DIC(0)="AEQZL"
>S DIC("?PARAM",200,"PART",1)="S"
>S DIC("?PARAM",662002,"FROM",1)="M"

>D ^DIC

Select ZZTAMI POINT TO NEW PERSON PERSON NAME: ??

   Choose from:
   MANNERS,JULIA      NOV 11, 1961     ANOTHER GREAT PROGRAMMER     JM
 PROGRAMMER
   MARSHALL,DELBERT MAY 05, 1965     WIZARD     TOAD          PROGRAMMER
   OGDEN,MARSHALL JUL 07, 1977     GREAT PROGRAMMER     MO PROGRAMMER
   RETROMAN,USER K M JR      JAN 01, 1969     COOLDUDE     UR
   TIMOTHY,S J      APR 03, 1948     KOOL KAT     SJT PROGRAMMER
```

```
  WERLY,BIRD      JUN 12, 1955     GROOVY GUY     BW PROGRAMMER
  WINNER,BIG      AUG 28, 1949     COMPUTER SPECIALIST     BW
PROGRAMMER
  WINNER,SMALL      AUG 28, 1948     SECOND PLACE     SW PROGRAMMER
  ZERO,BOB      MAR 02, 1948     VERY GOOD PROGRAMMER     BZ
IRMFO     PROGRAMMER

       You may enter a new ZZTAMI POINT TO NEW PERSON, if you wish

  Choose from:
  SHARED,MAIL
  SMITH,JOHN HOWARD STEVEN II
  STRALL,SEG        SAS
  SUPERMAN,JOE X Y JR
```

C.) In this example we are using the same files as in example "B", we will display entries from the pointing File #662002, using the "AC" index, which sorts the entries by TITLE, then by NAME. In this case, we will limit the number of entries displayed at one time from both File #662002 and File #200 to 5.

```
>S DIC="^DIZ(662002,",DIC(0)="AEQZL"
>S DIC("?PARAM",662002,"INDEX")="AC"
>S DIC("?N",662002)=5
>S DIC("?N",200)=5

>D ^DIC

Select ZZTAMI POINT TO NEW PERSON PERSON NAME: ??

   Choose from:
   A STATE   CALIFORNIA,MR      MAR 01, 1875     A STATE     MC     ABCD
   ANOTHER GREAT PROGRAMMER   MANNING,DARYL      NOV 11, 1961     ANOTHER
GREAT
 PROGRAMMER     DM          PROGRAMMER
   BROKER WHIZ   CROSS,BOB      FEB 05, 1950     BROKER WHIZ     BC
   COMPUTER SPECIALIST   WILLY,TED     AUG 28, 1949 COMPUTER SPECIALIST TW
   COOLDUDE   RETROMAN,USER K M JR      JAN 01, 1969     COOLDUDE     UR

                 ^

     You may enter a new ZZTAMI POINT TO NEW PERSON, if you wish

 Answer with NEW PERSON NAME
 Do you want the entire NEW PERSON List? Y <RET>  (Yes)
Choose from:
   ATESTMAN,BOB K III        BKA
   CALIFORNIA,MR        MC     ABCD
   CLARK,KENT        KC
   CROSS,BOB        BC          PROGRAMMER
   DELANCY,NAN        ND          PROGRAMMER
```

IX^DIC: Lookup/Add

This entry point is similar to ^DIC and MIX^DIC1, except for the way it uses cross-references to perform lookup. The three entry points perform lookups as follows:

^DIC	Starts with the B cross-reference, or uses only the B cross-reference [unless K is passed in DIC(0)].
IX^DIC	Starts with the cross-reference you specify or uses only the cross-reference you specify.
MIX^DIC1	Uses the set of cross-references you specify.

Input Variables (Required)

NOTE: All of the input variables described in ^DIC can be used in the IX^DIC call. The following variables are required.

DIC	The global root of the file, e.g., ^DIZ(16000.1,.
DIC(0)	The lookup parameters as previously described for ^DIC.
D	The cross-reference in which to start looking. If DIC(0) contains M, then DIC will continue the search on all other lookup cross-references, in alphabetical order. If it does not, then the lookup is only on the single cross-reference. This variable is killed by VA FileMan; it is undefined when the IX^DIC call is complete. If DIC(0) contains "L", (i.e., user will be allowed to add a new entry to the file), then either a) D should be set to "B" or b) D should be set to an index that alphabetically comes before "B" and DIC(0) should contain "M" or c) D should contain the name of a compound index.
X	If DIC(0) does not contain an A, then the variable X must be defined equal to the value you want to look up. If the lookup index is compound (i.e., has more than one data subscript), then X can be an array X(n) where "n" represents the position in the subscript. For example, if X(2) is passed in, it will be used as the lookup value to match to the entries in the

second subscript of the index. If only the lookup value X is passed, it will be assumed to be the lookup value for the first subscript in the index, X(1).

Input Variables (Optional)

All of the ^DIC input variables can be used in the IX^DIC call. These variables below are optional.

DIC("A"), **DIC("B"),** **DIC("DR"),** **DIC("P"),** **DIC("PTRIX",f,p,t)=d** **DIC("S"),** **DIC("V"),** **DIC("W")** **DIC("?N",file#)=n**	This set of input variables affects the behavior of lookup as described for ^DIC.

Output Variables

Y DIC always returns the variable Y. The variable Y is returned in one of these three formats:

- **Y=-1** The lookup was unsuccessful.
- **Y=N^S** N is the Internal Entry Number of the entry in the file and S is the value of the .01 field for that entry.
- **Y=N^S^1** N and S are defined as above and the 1 indicates that this entry has just been added to the file.

Y(0) This variable is only set if DIC(0) contains a Z. When the variable is set, it is equal to the entire zero node of the entry that was selected.

Y(0,0) This variable also is only set if DIC(0) contains a Z. When the variable is set, it is equal to the external form of the .01 field of the entry.

The following are examples of returned Y variables based on a call to the EMPLOYEE file stored in ^EMP(:

```
S DIC="^EMP(",DIC(0)="QEZ",X="SMITH"
D ^DIC
```

Returned are:

```
Y      = "7^SMITH,SAM"
Y(0)   = "SMITH,SAM^M^2231109^2
Y(0,0) = "SMITH,SAM"
```

If the lookup had been done on a file whose .01 field points to the EMPLOYEE file, the returned variables might look like this:

```
Y      = "32^7"  [ Entry #32 in this file and #7 in
                   EMPLOYEE file.]
Y(0)   = "7^RX 2354^ON HOLD"
Y(0,0) = "SMITH,SAM"  [.01 field of entry 7 in
                        EMPLOYEE file]
```

X Contains the value of the field where the match occurred.

If the lookup index is compound (i.e., has more than one data subscript), and if DIC(0) contains an A so that the user is prompted for lookup values, then X will be output as an array X(n) where "n" represents the position in the subscript and will contain the values from the index on which the entry was found. Thus, X(2) would contain the value of the second subscript in the index. If possible, the entries will be output in their external format (i.e., if the subscript is not computed and doesn't have a transform). If the entry is not found on an index (for example, when lookup is done with X=" " [the space-bar return feature]), then X and X(1) will contain the user input, but the rest of the X array will be undefined.

DTOUT This is only defined if DIC has timed-out waiting for input from the user.

DUOUT This is only defined if the user entered an up-arrow.

DO^DIC1: File Info Setup

This entry point retrieves a file's file header node, code to execute its identifiers and its screen (if any), and puts them into local variables for use during lookup into a file.

If $D(DO) is greater than zero, DO^DIC1 will QUIT immediately. If DIC("W") is defined before calling DO^DIC1, it will not be changed.

Input Variables

DIC The global root of the file, e.g., ^DIZ(16000.1,.

DIC(0) The lookup parameters as previously described for ^DIC.

Output Variables

DO File name^file number and specifiers. This is the file header node.

NOTE: Use the letter O, not the number zero, in this variable name.

DO(2) File number and specifiers. This is the second ^piece of DO. +DO(2) will always equal the file number.

DIC("W") This is an executable variable which contains the write logic for identifiers. When an entry is displayed, the execution of this variable shows other information to help identify the entry. This variable is created by $ORDERing through the data dictionary ID level, for example:

```
^DD(+DO(2),0,"ID",value)
```

NOTE: The specifier, I, must be in DO(2) for VA FileMan to even look at the ID-nodes.

DO("SCR") An executable variable which contains a file's screen (if any). The screen is an IF-statement that can screen out certain entries in the file. This differs from DIC("S") in that it is used on every lookup regardless of input or output; that is, the screen is applied to inquiries and printouts as well as to lookups. The value for this variable comes from ^DD(+DO(2),0,"SCR") and the specifier "s" must be in DO(2).

MIX^DIC1: Lookup/Add

This entry point is similar to ^DIC and IX^DIC, except for the way it uses cross-references to do lookup. The three entry points perform lookups as follows:

^DIC	Starts with the B cross-reference or uses only the B cross-reference (unless K is passed in DIC(0)).
IX^DIC	Starts with the cross-reference you specify or uses only the cross-reference you specify.
MIX^DIC1	Uses the set of cross-references you specify.

Input Variables (Required)

NOTE: All of the input variables described in ^DIC can be used in the MIX^DIC1 call. The following variables are required.

DIC	The global root of the file, e.g., ^DIZ(16000.1,.
DIC(0)	The lookup parameters as previously described for ^DIC.
D	The list of cross-references, separated by up-arrows, to be searched, e.g., D="SSN^WARD^B". This variable is killed by VA FileMan; it is undefined when the MIX^DIC1 call is complete. If DIC(0) contains "L", meaning that the user can add a new entry to the file, then either a) the "B" index should be included in the list contained in D, or b) D should be set to the name of a compound index. Make sure DIC(0) contains M; otherwise, only the first cross-reference in D will be used for the lookup.
X	If DIC(0) does not contain an A, then the variable X must be defined equal to the value you want to look up. If the lookup index is compound (i.e., has more than one data subscript), then X can be an array X(n) where "n" represents the position in the subscript. For example, if X(2) is passed in, it will be used as the lookup value to match to the entries in the second subscript of the index. If only the lookup value X is

passed, it will be assumed to be the lookup value for the first subscript in the index, X(1).

Input Variables (Optional)

All of the ^DIC input variables can be used in the MIX^DIC1 call. The variables below are optional.

DIC("A"), **DIC("B"),** **DIC("DR"),** **DIC("P"),** **DIC("PTRIX",f,p,t)=d** **DIC("S"),** **DIC("V"),** **DIC("W")** **DIC("?N",file#)=n**	This set of input variables affects the behavior of lookup as described for ^DIC.

Output Variables

Y

DIC always returns the variable Y. The variable Y is returned in one of the three following formats:

Y=-1 The lookup was unsuccessful.

Y=N^S N is the Internal Entry Number of the entry in the file and S is the value of the .01 field for that entry.

Y=N^S^1 N and S are defined as above and the 1 indicates that this entry has just been added to the file.

Y(0)

This variable is only set if DIC(0) contains a Z. When the variable is set, it is equal to the entire zero node of the entry that was selected.

Y(0,0)

This variable also is only set if DIC(0) contains a Z. When the variable is set, it is equal to the external form of the .01 field of the entry.

The following are examples of returned Y variables based on a call to the EMPLOYEE file stored in ^EMP(:

```
S DIC="^EMP(",DIC(0)="QEZ",X="SMITH"
D ^DIC
```

Returned are:

```
Y      = "7^SMITH,SAM"
Y(0)   = "SMITH,SAM^M^2231109^2
Y(0,0) = "SMITH,SAM"
```

If the lookup had been done on a file whose .01 field points to the EMPLOYEE file, the returned variables might look like this:

```
Y      = "32^7"  [ Entry #32 in this file and #7 in
                   EMPLOYEE file.]
Y(0)   = "7^RX 2354^ON HOLD"
Y(0,0) = "SMITH,SAM"  [.01 field of entry 7 in
                        EMPLOYEE file]
```

X Contains the value of the field where the match occurred.

If the lookup index is compound (i.e., has more than one data subscript), and if DIC(0) contains an A so that the user is prompted for lookup values, then X will be output as an array X(n) where "n" represents the position in the subscript and will contain the values from the index on which the entry was found. Thus, X(2) would contain the value of the second subscript in the index. If possible, the entries will be output in their external format (i.e., if the subscript is not computed and doesn't have a transform). If the entry is not found on an index (for example, when lookup is done with X=" " [the space-bar return feature]), then X and X(1) will contain the user input, but the rest of the X array will be undefined.

DTOUT This is only defined if DIC has timed-out waiting for input from the user.

DUOUT This is only defined if the user entered an up-arrow.

WAIT^DICD: Wait Messages

Use this entry point to display VA FileMan's informational messages telling users that the program is working and they must wait a while. The selection of the phrase is random. There are no input or output variables.

Some sample messages are:

```
...EXCUSE ME, I'M WORKING AS FAST AS I CAN...

...SORRY, LET ME THINK ABOUT THAT A MOMENT...
```

FILE^DICN: Add

This entry point adds a new entry to a file. The INPUT transform is **not** used to validate the value being added as the .01 field of the new entry. This call does not override the checks in the LAYGO nodes of the data dictionary; they must still prove true for an entry to be added.

FILE^DICN can also be used to add subentries in multiples. See the Adding New Subentries to a Multiple discussion in the description of ^DIC.

Variables to Kill

DO If DO is set, then FileMan assumes that all of the variables described as output in the call to DO^DIC1 have been set as well and that they describe the file to which you wish to add a new record. If you're not sure, then DO should be killed and the call will set it up for you based on the global root in DIC.

NOTE: This variable is D with the letter O, not zero.

Input Variables

DIC The global root of the file.

DIC(0) (Required) A string of alphabetic characters which alter how DIC responds. At a minimum this string must be set to null. The characters you can include are:

E **E**cho back information. This tells DIC that you are in an interactive mode and are expecting to be able to receive input from the user. If there are identifiers when adding a new entry, for example, the user can edit them as the entry is added if the E flag is used.

F Prevents saving the entry number of the matched entry in the ^DISV global. Ordinarily, the entry number is saved at ^DISV(DUZ,DIC). This allows the user to do a subsequent lookup of the same entry simply by pressing the space bar and the Enter/Return key. To avoid the time cost of setting this global, include an F in DIC(0).

Z **Z**ero node returned in Y(0) and external form in Y(0,0).

DIC("P") **NOTE:** Beginning with Version 22.0 of VA FileMan, the developer is no longer required to set DIC("P").

The only exception to this is for a few files that are not structured like a normal VA FileMan file, where the first subscript of the data is variable in order to allow several different "globals" to use the same DD. An example of this is the VA FileMan Audit files where the first subscript is the file number of the file being audited.

Used when adding subentries in multiples. See description in ^DIC section.

DA Array of entry numbers. See the Adding New Subentries to a Multiple discussion in the description of ^DIC.

X The internal value of the .01 field, as it is to be added to the file. The programmer is responsible for ensuring that all criteria described in the INPUT transform have been met. That means that the value X must be in internal format as it would be after executing the input transform. For example, a date must be in FileMan internal format '2690302', not 'March 02, 1969'. Also local variables set by the input transform code must be set. For example, if the input transform sets DINUM, then DINUM must be set to the record number at which the entry must be added.

DINUM (Optional) Identifies the subscript at which the data is to be stored, that is, the internal entry number of the new record, shown as follows. (This means that DINUM must be a canonic number and that no data exists in the global at that subscript location.)

```
$D(@(DIC_DINUM_")"))=0
```

If a record already exists at the DINUM internal entry number, no new entry is made. The variable Y is returned equal -1.

DIC("DR") (Optional) Used to input other data elements at the time of adding the entry. If the user does not enter these elements, the entry will not be added. The format of DIC("DR") is the same as the variable DR described under the discussion of ^DIE.

If there are any required Identifiers for the file or if there are keys defined for the file (in the KEY file), and if DIC(0) does not

contain an E, then the identifier and key fields MUST be present in DIC("DR") in order for the record to be added. If DIC(0) contains E, the user will be prompted to enter the identifier and key fields whether or not they are in DIC("DR").

Output Variables

Y DIC always returns the variable Y, which can be in one of the two following values:

- **Y=-1** Indicates the lookup was unsuccessful; no new entry was added.
- **Y=N^S^1** N is the internal number of the entry in the file, S is the value of the .01 field for that entry, and the 1 indicates that this entry has just been added to the file.

Y(0) This variable is only set if DIC(0) contains a Z. When it is set, it is equal to the entire zero node of the entry that was selected.

Y(0,0) This variable is also only set if DIC(0) contains a Z. When it is set, it is equal to the external form of the .01 field of the entry.

DTOUT This is only defined if DIC has timed-out waiting for input from the user.

DUOUT This is only defined if the user entered an up-arrow.

X The variable X will be returned unchanged from the input value.

YN^DICN: Yes/No

This entry point is a reader for a YES/NO response. You must display the prompt yourself before calling YN^DICN. YN^DICN displays the question mark and the default response, reads and processes the response, and returns %.

Recommendation: Instead of using this entry point, it is suggested that you use the generalized reader ^DIR. ^DIR gives you greater flexibility in displaying prompts and help messages and also presents more information about the user's response.

Input Variables

% Determines the default response as follows:

% = 0 (zero)	No default
% = 1	YES
% = 2	NO

Output Variables

% The processed user's response. It can be one of the following:

% = -1	The user entered an ^ (up-arrow).
% = 0 (zero)	The user pressed the Enter/Return key when no default was presented OR the user entered a ? (question mark).
% = 1	The user entered a YES response.
% = 2	The user entered a NO response.

%Y The actual text that the user entered.

DQ^DICQ: Entry Display for Lookups

This entry point displays the list of entries in a file a user can see. It can be used to process question mark responses directly. If DO is not defined, the first thing that DQ^DICQ does is call DO^DIC1 to get the characteristics of the selected file.

Input Variables

DIC	(Required) The global root of the file.
DIC(0)	(Required) The lookup input parameter string as described for ^DIC.
DIC("S")	(Optional) Use this variable in the same way as it is described as an input variable for ^DIC.
DIC("?N",file#)=n	(Optional) Use this variable in the same way it is described as input to ^DIC.
DIC("?PARAM",file#,"INDEX")=index name	(Optional) Use this input array in the same way it is described as input to ^DIC.
DIC("?PARAM",file#,"FROM",n)=value	(Optional) Use this input array in the same way it is described as input to ^DIC.
DIC("?PARAM",file#,"PART",n)=value	(Optional) Use this input array in the same way it is described as input to ^DIC.
D	(Required) Set to "B".
DZ	(Required) Set to "??". This is set in order to prevent VA FileMan from issuing the "DO YOU WANT TO SEE ALL nn ENTRIES?" prompt.

DT^DICRW: FM Variable Setup

Sets up the required variables of VA FileMan. There are no input variables; simply call the routine at this entry point.

NOTE: This entry point kills the variables DIC and DIK.

Output Variables

DUZ	Set to zero if it is not already defined.
DUZ(0)	Set to null if not already defined. If DUZ(0)="@", this subroutine will enable terminal break if the operating system supports such functionality.
IO(0)	Set to $I if IO(0) is not defined. Therefore, this program should not be called if the user is on a device different from the home terminal and IO(0) is undefined.
DT	Set to the current date, in VA FileMan format.
U	Set to the up-arrow (^).

EN^DID: Data Dictionary Listing

This entry point prints and/or displays a file's data dictionary listing by setting the input variables (the same as the output from the List File Attributes option described in the VA FileMan Advanced User Manual).

Input Variables

DIC Set to the data dictionary number of the file to list.

DIFORMAT Set to the desired data dictionary listing format. Must be one of the following strings:

STANDARD

BRIEF

MODIFIED STANDARD

TEMPLATES ONLY

GLOBAL MAP

CONDENSED

INDEXES AND CROSS-REFERENCES ONLY

KEYS ONLY

^DIE: Edit Data

This routine handles input of selected data elements for a given file entry. You should use ^DIE only to edit *existing* records.

NOTE: When you call the DIE routine, it does not lock the record; you must do that yourself. See the discussion of locking below.

To allow the user to interactively choose the fields to edit, use the EN^DIB entry point instead.

Input Variables

DIE

(Required) The global root of the file in the form ^GLOBAL(or ^GLOBAL(#, or the number of the file.

If you are editing a subfile, set DIE to the full global root leading to the subfile entry, including all intervening subscripts and the terminating comma, up to but not including the IEN of the subfile entry to edit.

DA

(Required) If you are editing an entry at the top level of a file, set DA to the internal entry number of the file entry to be edited.

If you are editing an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to edit, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level. See the section below on Editing a Subfile Directly for more information.

NOTE: The variable DA is killed if an entry is deleted within DIE. This can happen if the user answers with the @-sign when editing the entry's .01 field.

DR

(Required) A string specifying which data fields are asked for the given entry. The fields specified by DR are asked *whether or not VA FileMan Write access security protection has been assigned* to the fields.

You can include the following in the DR-string:

Field number: The internal number of a field in a file.

Field with Default Value: A field number followed by // (two slashes), followed by a *default value*. You can make a field with no current data value default to a particular data value you specify. For example, if there is a file entry stored descendent from ^FILE(777), and field #27 for this file is DATE OF ADMISSION, and you want the user to see:

```
DATE OF ADMISSION: TODAY//
```

then the calling program should be:

```
S DR="27//TODAY",DIE="^FILE(",DA=777
D ^DIE
```

If the user just presses the Enter/Return key when seeing the prompt, DIE acts as though the user typed in the word TODAY.

Stuff a Field Value (Validated): A field number followed by /// (three slashes), followed by a value. The value should be the *external* form of the field's value, that is, the format that would be acceptable as a user's response. The value is *automatically inserted* into the database after passing through the INPUT transform. For example:

```
S DR="27///TODAY",DIE="^FILE(",DA=777
D ^DIE
```

The user sees no prompts, and the current date is automatically stuffed into field #27 of entry #777, *even if other data previously existed* there.

In the course of writing a routine, you may want to pass the value contained in a variable to DIE and automatically insert the value into a field. In that case, you would write:

```
S DR="27///^S X=VAR"
```

You can also use the three-slash stuff to automatically add or select an entry in a multiple. For example, if field #60 is a multiple field, and you write:

```
S DR="60///TODAY"
```

the entry in the subfile corresponding to TODAY would be selected, or added if it didn't already exist. Note, however, that if TODAY didn't already exist in the file, but couldn't be added (because LAYGO wasn't allowed, for example), or if more than one TODAY entry already existed in the file (that is, the lookup value was ambiguous), ^DIE will prompt the user to select an entry in the subfile. If you wish to add entries or edit existing entries non-interactively, consider using UPDATE^DIE and FILE^DIE instead.

Stuff a Field Value (Unvalidated): A field number followed by //// (four slashes), followed by a value. The value is *automatically inserted without validation* into the database. For example:

```
S DR="27////2570120",DIE="^FILE(",DA=777
D ^DIE
```

The user sees no prompts, and the value 2570120 is put into field 27 without going through the INPUT transform. When using this form, the data after the four slashes must already be in its internally stored form. *This cannot be used for .01 fields due to the differences between DIE and DIC.*

NOTE: Key uniqueness is not enforced when a 4-slash stuff is used.

Field Value Deletion: A field number followed by three or four slashes (/// or ////) and an @-sign. This *automatically deletes the field value.* For example:

```
S DR="27///@"
```

The user does not see any prompts, and the value for field #27 is deleted.

NOTE: You cannot use this method to delete the value of a required field, an uneditable field, a key field, or a field the user does not have Delete access to.

Field Number Range: A range of field numbers, in the form M:N, where M is the first and N the last number of the *inclusive range.* All fields whose numbers lie within this range are asked.

Placeholder for Branching: A placeholder like @1. See the discussion of branching below.

M Code: A line of M code.

Combination: A sequence of any of the above types, separated by semicolons. If field numbers .01, 1, 2, 4, 10, 11, 12, 13, 14, 15, and 101 exist for the file stored in ^FILE, and you want to have fields 4, .01, 10 through 15, and 101 asked in that order for entry number 777, you simply write:

```
S DIE="^FILE(",DA=777,DR="4;.01;10:15;101"
D ^DIE
```

NOTE: The DR-string contains the semicolon delimiter to specify field numbers and the colon to specify a range of fields. This prevents these two characters from being used as defaults. They can, however, be placed in a variable which is then used as the default instead of a literal, for example:

```
S DR="27///^S X=VAR"
```

INPUT template: An INPUT template name, preceded by an open bracket ([) and followed by a closed bracket (]). All the fields in that template are asked.

DIE("NO^") (Optional) Controls the use of the ^ in an edit session. If this variable does not exist, unrestricted use of the ^ for jumping and exiting is allowed. The variable may be set to one of the following:

"**OUTOK**"	Allows exiting and prevents all jumping.
"**BACK**"	Allows jumping back to a previously edited field and does not allow exiting.
"**BACKOUTOK**"	Allows jumping back to a previously edited field and allows exiting.
"**Other value**"	Prevents all jumping and does not allow exiting.

DIE("PTRIX",f,p,t)=d

DIE("PTRIX",f,p,t)=d where,

f = the from (pointing) file number

p = the pointer field number

t = the pointed-to file number

d = an up-arrow (^) delimited list of index names

This optional input array allows you to control how lookups are done on both multiple and non-multiple pointer and variable pointer fields. Each node in this array is set to a list of index names, separated by up-arrows (^). When the user edits a pointer or variable pointer field, only those indexes in the list are used when searching the pointed-to file for matches to the lookup value.

For example, if your input template contains a field #5 on file #16100 that is a pointer to the NEW PERSON file (#200), and you want the lookup on the NEW PERSON file to be by name ("B" index), or by the first letter of the last name concatenated with the last 4 digits of the social security number ("BS5" index), you would set the following node before the ^DIE call:

```
DIE("PTRIX",16100,5,200)="B^BS"
```

Note that if you allow records to be added to the pointed-to file, you should include a "B" in the list of indexes, since when ^DIE adds an entry, it assumes the .01 field for the new entry is the lookup value. However, the "B" index would not need to be included if the first index in the "PTRIX" node is a compound index whose first subscript is the .01 field.

DIDEL

(Optional) Overrides the Delete access on a file or subfile. Set DIDEL equal to the number of the file before calling DIE to allow the user to delete an entire entry from that file, even if the user does not normally have the ability to delete. This variable does not override the "DEL"-nodes described in the Other Field Definition Nodes of the Global File Structure section.

Output Variables

DTOUT Is set when a time-out has occurred.

NOTE: DA, DIE, DR, DIE("NO^"), and DIDEL are not killed by DIE; however, the variable DA is killed if the entry is deleted within DIE. This can happen if the user answers with an @-sign when editing the entry's .01 field.

Details and Features of Data Editing

1. **Locking**
2. **Edit Qualifiers**
3. **Branching**
4. **Specific Fields in Multiples**
5. **Continuation DR-Strings**
6. **Detecting Up-Arrow Exits**
7. **Editing a Subfile Directly**
8. **Screening Variable Pointers**
9. **Filing**
10. **New Style Compound Indexes and Keys**

1. Locking

If you want to ensure that two users cannot edit an entry at the same time, lock the entry. It is recommended that you use incremental locks.

Here is a simple example of using incremental locks to lock an entry before editing and to remove the lock after:

```
S DIE="^FILE(",DA=777,DR="[EDIT]"
L +^FILE(777):0 I $T D ^DIE L -^FILE(777) Q
W !?5,"Another user is editing this entry." Q
```

NOTE: The DIE call itself does *NO* locking.

2. Edit Qualifiers

In the DR string, you can use edit qualifiers (described in the *VA FileMan Advanced User Manual*) in conjunction with the fields you specify. The possible qualifiers are T, DUP, REQ, and text literal strings in quotes.

In interactive mode, users can combine qualifiers with fields by using semicolon separators. But, in DR-strings, semicolons are already used to delimit individual fields, soy must use a different syntax for DR. Basically, leave out the semicolon and the unnecessary characters. Using field #3 as an example, the syntax for edit qualifiers in DR-strings is:

Interactive Syntax	Syntax for DR-string	Explanation
3;T	3T	The T follows the field number immediately.
3;"xxx"	3xxx	The quotes are removed from the literal and it follows the field number immediately.
3;DUP	3d	The D becomes lowercase and the UP is dropped.
3;REQ	3R	The EQ is dropped and the uppercase R follows immediately.

You can combine specifiers as long as you separate them with tildes (~). For example, if you want to require a response to field #3, and issue the title rather than the prompt, put 3R~T in the DR-string.

3. Branching

You can include **branching logic** within DR. To do this, insert an executable M statement in one of the semicolon-pieces of DR. The M code is executed when this piece of DR is encountered by the DIE routine.

If the M code sets the variable Y, DIE jumps to the field whose number (or label) matches Y. (The field must be specified elsewhere within the DR variable.) Y may look like a placeholder, e.g., **@**1. If Y is set to zero or the null string, DIE exits. If Y is killed, or never set, no branching occurs.

The M code can calculate Y based on X, which equals the internal value of the field previously asked for (as specified by the previous semicolon-piece of DR). Take the

example below and suppose that you do not want the user to be asked for field .01 if the answer to field 4 was YES, you would write the following:

```
S DIE="^FILE(",DA=777
S DR="4;I X=""YES"" S Y=10;.01;10:15;101"
D ^DIE
```

NOTE: The ability to up-arrow jump to specific fields does not take into account previous branching logic. You must ensure that such movements are safe.

4. Specific Fields in Multiples

When you include the field number of a multiple in a DR-string, all the subfields of the multiple are asked. However, suppose you want to edit only selected subfields in the multiple. To do this, set DR in the usual manner and in addition set a subscripted value of DR equal to the subfields to edit. Subscript the additional DR node by file level and then by the multiple's subfile number.

For example, if field #15 is a multiple and the subfile number for the multiple is 16001.02 and you want the user to be prompted only for subfields .01 and 7, do the following:

```
S DR=".01;15;6;8"
S DR(2,16001.02)=".01;7"
```

where the first subscript, 2, means the second level of the file and the second subscript is the subfile number of the multiple field (#15).

5. Continuation DR-Strings

If there are more than 245 characters in a DR-string, you can set continuation strings by defining the DR-array at the third subscript level. These subscripts should be sequential integers starting at 1. For example, the first continuation node of DR(2,16001.02) would be DR(2,16000.02,1); the second would be DR(2,16001.02,2), and so on.

6. Detecting Up-Arrow Exits

You can determine, upon return from DIE, whether the user exited the routine by typing an up-arrow. If the user did so, the subscripted variable Y is defined; if all questions were asked and answered in normal sequence, $D(Y) is zero.

7. Editing a Subfile Directly

You can call ^DIE to directly edit an entry in a subfile; you can descend into as many subfiles as you need to. Set the DIE input variable to the full global root

leading to the subfile entry, including all intervening subscripts and the terminating comma, up to—but not including—the IEN of the subfile entry to edit. Then set an array element for each file and subfile level in the DA input variable, where DA=entry number in the subfile to edit, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level.

For example, suppose that the data in subfile 16000.02 is stored descendent from subscript 20 and you are going to edit entry number 777, subentry number 1; you would write the following:

```
S DIE="^FILE(777,20," ; global root of subfile
    S DA(1)=777 ; entry number in file
    S DA=1 ; entry number in subfile
    S DR="3;7" ; fields in subfile to edit
    D ^DIE
```

NOTE: The internal number of the entry into the file appears in the variable DIE and appears as the value of DA(1). When doing this, it is necessary that the subfile descriptor node be defined. In this example, it would be:

```
^FILE(777,20,0)="^16000.02^last number entered^number of entries"
```

8. Screening Variable Pointers

A variable pointer field can point to entries in more than one file. You can restrict the user's ability to input entries to certain files by setting the DIC("V") variable in a DR-string or in an INPUT template. It screens files from the user. Set DIC("V") equal to a line of M code that returns a truth value when executed. The code is executed after someone enters data into a variable pointer field. If the code tests false, the user's input is rejected; FileMan responds with **??** and a "beep."

The code setting the DIC("V") variable can be put into a DR-string or into an INPUT template. It is **not** a separate input variable for ^DIE or ^DIC. It should be set immediately before the variable pointer field is edited and it should be killed immediately after the field is edited.

When the user enters a value at a variable pointer field's prompt, FileMan determines in which file that entry is found. The variable Y(0) is set equal to information for that file from the data dictionary definition of the variable pointer field. You can use Y(0) in the code set into the DIC("V") variable. Y(0) contains the following:

^-Piece	Contents
Piece 1	File number of the pointed-to file.

Piece 2 Message defined for the pointed-to file.

Piece 3 Order defined for the pointed-to file.

Piece 4 Prefix defined for the pointed-to file.

Piece 5 y/n indicating if a screen is set up for the pointed-to file.

Piece 6 y/n indicating if the user can add new entries to the pointed to file.

All of this information was defined when that file was entered as one of the possibilities for the variable pointer field.

For example, suppose field #5 is a variable pointer pointing to files 1000, 2000, and 3000. If you only want the user to be able to enter values from files 1000 or 3000, you could set up your INPUT template like this:

```
THEN EDIT FIELD: ^S DIC("V")="I +Y(0)=1000!(+Y(0)=3000)"
THEN EDIT FIELD: 5
THEN EDIT FIELD: ^K DIC("V")
```

9. Filing

DIE files data when any one of the following conditions is encountered:

- The field entered or edited is cross-referenced
- A change of level occurs, i.e., either DIE must descend into a multiple or ascend to the level above
- Navigation to another file occurs
- M code is encountered in one of the semicolon-pieces of the DR-string or in a template
- $S becomes less than 2000
- The user up-arrows to a field
- The end of the DR-string or INPUT template is reached
- Templates are compiled and the execution is transferred from one routine to the next

10. New Style Compound Indexes and Keys

^DIE traditionally fires cross-references when the field on which the cross-reference is defined is edited. New-style cross-references that have an execution of "RECORD" (hereafter referred to as record-level indexes) are fired once at the end of the ^DIE call, after all the semicolon pieces of the DR string have been processed.

When record-level uniqueness indexes are fired, the corresponding keys (hereafter called record-level keys) are checked to ensure that they are unique. If edits to a field in a key result in a duplicate key, then changes to that field are backed out and an error message is presented to the user.

You can set the variable DIEFIRE in any of the semicolon-pieces of DR to instruct FileMan to fire the record-level indexes at that point and validate the corresponding record-level keys. You can also control what FileMan does if any of the record-level keys is invalid.

DIEFIRE contains:	**Action:**
M	Print error message to user
L	Return the DIEBADK array (see example immediately below)
R	Restore invalid key fields to their pre-edited values

If DIEFIRE contains an L and a key is invalid, the DIEBADK array is set as follows:

```
DIEBADK(rFile#,key#,file#,IENS,field#,"O") = the original value of the field
DIEBADK(rFile#,key#,file#,IENS,field#,"N") = the new (invalid) value of the
field
```

where,

rFile#	=	the **root file** of the **uniqueness index** of the key. This is the file or subfile number of the fields that make up the key.
key#	=	the internal entry number of the key in the KEY file.
file#	=	the **file** of the **uniqueness index** of the key. This is the file or subfile where the **uniqueness index** resides. For whole file indexes, this is a file or subfile at a higher level than **root file**.
IENS	=	the IENS of the record that—with the edits—would have a non-unique key.
field#	=	the field number of the field being edited.

If any of the Keys is invalid, FileMan sets the variable X to the string "BADKEY", which can be checked by M code in the subsequent semicolon-piece of the DR string.

The variable X and the local array DIEBADK are available for use only in the semicolon piece immediately following the piece where the DIEFIRE was set.

For example:

```
S DIE="^FILE(",DA=777
S DR="@1;.01;.02;S DIEFIRE=""R"";I X=""BADKEY""
         S Y=""@1"";1;2"
D ^DIE
```

Here, the .01 and .02 field makes up a key to the file. After prompting the user for the value of the .02, DIEFIRE is set to force VA FileMan to fire the record-level indexes and validate the key. If the key turns out to be invalid, FileMan sets X equal to "BADKEY" and, since DIEFIRE equals R, restores the fields to their pre-edited values. In the next semicolon-piece, we check if X equals "BADKEY" and, if so, branch the user back to the placeholder @1.

^DIEZ: Input/Compile

Interactively compiles or recompiles an INPUT template.

Compiling an INPUT template means telling VA FileMan to write a hard-coded M routine that will do just what a particular INPUT template tells the Enter or Edit File Entries option to do. This can enhance system performance by reducing the amount of data dictionary lookup that accompanies VA FileMan input. The routines created by DIEZ should run from 20% to 80% more efficiently than DIE does for the same input.

Call ^DIEZ and specify the maximum number of characters you want in your routines, the name of the INPUT template you are using, and the name of the M routine you want to create. If more code is compiled than will fit into a single routine, overflow code will be incorporated in routines with the same name, followed by 1, 2, etc. For example, routine DGT may call DGT1, DGT2, etc.

Once DIEZ has created a hard-coded routine for a particular INPUT template, VA FileMan **automatically** uses that routine in the Enter or Edit File Entries option, whenever that template is specified for input. When definitions of fields used in the EDIT template are altered by the Modify File Attributes or Utility Functions option, the hard-code routine(s) is (are) recompiled immediately.

EN^DIEZ: Input/Compile

This entry point **compiles** or **recompiles** an INPUT template, without user intervention. For more information about compiled INPUT templates, see ^DIEZ.

Input Variables

X	The name of the routine for the compiled INPUT template.
Y	The internal entry number of the INPUT template to be compiled.
DMAX	The maximum size the compiled routines should reach. Consider using the $$ROUSIZE^DILF function to set this variable.

^DIK: Delete Entries

Call DIK at ^DIK to delete an entry from a file.

WARNING: Use DIK to delete entries with extreme caution. It does not check Delete access for the file or any defined "DEL" nodes. Also, it does not update any pointers to the deleted entries. However, it does execute all cross-references and triggers.

Input Variables

DIK

The global root of the file from which you want to delete an entry.

If you are deleting a subentry, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to—but not including—the IEN of the subfile entry to delete.

DA

If you are deleting an entry at the top level of a file, set DA to the internal entry number of the file entry to delete. For example, to delete SAM SMITH, who is entry number 7, from the EMPLOYEE file, stored in the global ^EMP, write the following:

```
S DIK="^EMP(",DA=7
D ^DIK
```

If you are deleting an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to delete, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level. For example, suppose employee JOHN JONES (record #1) has two skill entries (subrecords #1 and #2) in a SKILL multiple. To delete the SKILL multiple's subrecord #2 you would write:

```
S DA(1)=1,DA=2,DIK="^EMP("_DA(1)_",""SX"","
   D ^DIK
```

where DA is the skill entry number in the subfile and DA(1) is the employee's internal entry number in the EMPLOYEE file.

Looping to Delete Several Entries

^DIK leaves the DA-array and DIK defined. So you can loop through a file to delete several entries:

```
S DIK="^EMP(" F DA=2,9,11 D ^DIK
```

This deletes entries 2, 9 and 11 from the EMPLOYEE file.

Deleting Fields from a File

As discussed in the How to Read an Attribute Dictionary section of the Global File Structure chapter, each attribute dictionary is also in the form of a file. You can therefore use the routine DIK to delete a *single-valued field* (i.e., not a multiple) from a file. To do this, the variable DIK is set to the file's data dictionary global node; DA is set to the number of the field to be deleted; and DA(1) is set to the file number. To delete the field SEX from our EMPLOYEE file example, simply write:

```
S DIK="^DD(3,",DA=1,DA(1)=3
D ^DIK
```

When you use ^DIK to delete fields from a file, the data is not deleted.

EN^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

EN^DIK reindexes one or more cross-references of a field for one entry in a file. It executes the KILL logic first and then executes the SET logic of the cross-reference. Before reindexing, you should be familiar with the effects of all relevant cross-references that could be fired (including bulletins, triggers, and MUMPS-type).

Input Variables

DIK If you are reindexing an entry at the top level of a file, set DIK to the global root of the file.

If you are reindexing a subentry, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to—but not including—the IEN of the subfile entry to reindex.

DA If you are reindexing an entry at the top level of a file, set DA to the internal entry number of the file entry to reindex.

If you are reindexing an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to reindex, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level.

DIK(1) Use the field number (to get all indexes) or the field number *and* specific indexes of the cross-reference. See the ENALL^DIK entry point description for examples.

EN1^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

EN1^DIK reindexes one or more cross-references of a field for one entry in a file. It **only** executes the SET logic of the cross-reference.

Before reindexing, you should be familiar with the effects of all relevant cross-references that could be fired (including bulletins, triggers, and MUMPS-type).

Input Variables

DIK If you are reindexing an entry at the top level of a file, set DIK to the global root of the file.

If you are reindexing a subentry, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to—but not including—the IEN of the subfile entry to reindex.

DA If you are reindexing an entry at the top level of a file, set DA to the internal entry number of the file entry to reindex.

If you are reindexing an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to reindex, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level.

DIK(1) Use the field number (to get all cross-references) or the field number *and* specific indexes of the cross-references you want. See the ENALL^DIK entry point description for examples.

ENALL^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

ENALL^DIK reindexes all entries in a file for the cross-references on a specific field. It may also be used to reindex all entries within a single subfile, that is a subfile corresponding to only one of the file's entries. ENALL^DIK only executes the SET logic.

Before reindexing, you should be familiar with the effects of all relevant cross-references that could be fired (including bulletins, triggers, and MUMPS-type).

NOTE: IXALL^DIK, ENALL^DIK, and the Re-Index File option on the Utility Functions menu set the 3rd piece of the 0 node of the file's global root (the file header) to the last internal entry number used in the file. They set the 4th piece to the total number of entries in the file.

Input Variables

DIK If you are reindexing an entry at the top level of a file, set DIK to the global root of the file.

If you are reindexing subentries, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to—but not including—the iens of the subfile entries to reindex.

DA(1..n) If you are reindexing entries in a subfile, set up DA as an array, where DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level. Since ENALL^DIK reindexes all entries at a given file level, don't set the unsubscripted DA node.

DIK(1) Use the field number (to get all indexes) or the field number *and* specific cross-references separated by up-arrows as shown below:

```
S DIK(1)="FLD#" ;Just the field number to get all indexes.
```

OR:

```
;Field number followed by x-ref name or number.
S DIK(1)="FLD#^INDEX"
;See the examples below:

S DIK(1)=".01^B"
S DIK(1)=".01^B^C"
S DIK(1)=".01^1^2"
D ENALL^DIK
```

IX^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

IX^DIK reindexes all cross-references of the file for only one entry in the file. It executes first the KILL logic and then the SET logic. Reindexing occurs at all file levels at or below the one specified in DIK and DA.

Before reindexing, you should be familiar with the effects of all relevant cross-references that could be fired (including bulletins, triggers, and MUMPS-type).

Input Variables

DIK If you are reindexing an entry at the top level of a file, set DIK to the global root of the file.

If you are reindexing only a subentry, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to—but not including—the IEN of the subfile entry to reindex.

DA If you are reindexing an entry at the top level of a file, set DA to the internal entry number of the file entry to reindex.

If you are reindexing an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to reindex, DA(1) is the entry

number at the next higher file level,...DA(n) is the entry number at the file's top level.

IX1^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

IX1^DIK reindexes all cross-references of the file for only one entry in the file. It **only executes the SET logic** of the cross-reference. Reindexing occurs at all file levels at or below the one specified in DIK and DA.

Before reindexing, you should be familiar with the effects of all relevant cross-references that could be fired (including bulletins, triggers, and MUMPS-type).

Input Variables

DIK If you are reindexing an entry at the top level of a file, set DIK to the global root of the file.

If you are reindexing a subentry, set DIK to the full global root leading to the subentry, including all intervening subscripts and the terminating comma, up to but not including the IEN of the subfile entry to reindex.

DA If you are reindexing an entry at the top level of a file, set DA to the internal entry number of the file entry to reindex.

If you are reindexing an entry in a subfile, set up DA as an array, where DA=entry number in the subfile to reindex, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level.

IXALL^DIK: Reindex

Reindexing Quick Reference

Entry Point	Reindexes Entries	Reindexes Xrefs	Executes Logic
EN^DIK	1	Some or all for 1 field	KILL then SET
EN1^DIK	1	Some or all for 1 field	SET
ENALL^DIK	All	Some or all for 1 field	SET
IX^DIK	1	All	KILL then SET
IX1^DIK	1	All	SET
IXALL^DIK	All	All	SET

IXALL^DIK reindexes **all** cross-references for **all** entries in a file. It only executes the SET logic.

Before reindexing, you should be familiar with the effects of all relevant cross-references (including bulletins, triggers, and MUMPS-type) that could be fired.

NOTE: IXALL^DIK, ENALL^DIK, and the Re-Index File option on the Utility Functions menu set the 3rd piece of the 0 node of the file's global root (the file header) to the last internal entry number used in the file. They set the 4th piece to the total number of entries in the file.

Input Variable

DIK The global root of the file to be indexed.

Examples

Example 1

A simple call to reindex the EMPLOYEE file would be:

```
>S DIK="^EMP(" D IXALL^DIK
```

Example 2

The reindexing of data dictionary #3 would be:

```
>S DA(1)=3,DIK="^DD(3," D IXALL^DIK
```

^DIKZ: Cross-reference Compilation

Cross-references can be compiled into M routines by calling ^DIKZ. You will be prompted to specify the maximum routine size and the name or number of the file. If you specify the routine name XXX and more code is generated than can fit into that one routine, overflow routines (XXX1, XXX2, etc.) will be created. Routine XXX may call XXX1, XXX2, etc.

Once DIKZ has been used to create hard-coded cross-reference routines, those routines are used when calls to any entry point in DIK are made. However, if you restrict the cross-references to be reindexed by using the DIK(1) variable, the compiled routines are not used. As soon as data dictionary cross-references are added or deleted, the routines are recompiled. The purpose of this DIKZ code generation is simply to improve overall system throughput.

See the Edit File section of the *VA FileMan Advanced User Manual* for instructions on permanently stopping the use of compiled cross-references, uncompiling cross-references.

EN^DIKZ: Compile

EN^DIKZ recompiles a file's cross-references by setting the input variables without user intervention.

Input Variables

X	The routine name.
Y	The file number of the file for which you want the cross-references recompiled.
DMAX	The maximum size the compiled routines should reach. Consider using the $$ROUSIZE^DILF function to set this variable.

$$ROUSIZE^DILF: Routine Size

This argumentless function returns the maximum routine size that should be used when compiling cross-references, print templates, or input templates.

Format

```
$$ROUSIZE^DILF
```

Input Parameters

None

Output

This function returns the maximum routine size defined in the MUMPS OPERATING SYSTEM file (#.7).

Example

```
>W $$ROUSIZE^DILF
4000
```

^DIM: M Code Validation

Call ^DIM to validate any line of M code. ^DIM checks that code conforms to the 1995 ANSI Standard. Code is also checked against aspects of VHA's Programming Standards and Conventions (SAC).

NOTE: ^DIM does not allow killing an unsubscripted global.

Input Variable

X Invoke ^DIM with the line to be validated in the local variable X.

Output Variable

X ^DIM either kills X or leaves it unchanged. If $D(X) is zero on return from ^DIM, the line of code is invalid. However, the converse is not always true; in other words, ^DIM is not as smart as a real M interpreter and sometimes validates strings when it should not.

DT^DIO2: Date/Time Utility

This entry point takes an internal date in the variable Y and *writes out* its external form.

Example

```
>S Y=2690720.163 D DT^DIO2
JUL 20,1969  1630
```

This results in Y being equal to JUL 20,1969 16:30. (No space before the 4-digit year; 2 spaces before the hours [1630].)

Input Variable

Y (Required) This contains the internal date to be converted. Y is required and it is not changed.

In addition, see X ^DD("DD") and DD^%DT, which also convert a date from internal YYYMMDD format to external format.

^DIOZ: Sort/Compile

This entry point marks a SORT template compiled or uncompiled. The ^DIOZ entry point asks for the name of the SORT template to be used and whether the user wishes (1) to mark it compiled or (2) to uncompile it if it is already marked compiled. Actual compilation occurs at the time the template is used in the sort/print. There are no input or output variables.

SORT templates can be compiled into M routines to increase efficiency of the sort and improve system performance. Good candidates for compilation are sorts with many sort fields or those that sort on fields reached with relational syntax. The process of sort compilation is different from other FileMan compiling activities. SORT templates can be "marked" for compilation, then each time the SORT template is used in a FileMan sort/print, a new compiled routine is created. When the print job finishes, the routine is deleted. The routine is named DISZnnnn where "nnnn" is a four-digit number. The routine names are reused. Routine numbers are taken from the Compiled Routine file (described in the section on the ENRLS^DIOZ utility in the *VA FileMan Advanced User Manual*). Thus, a routine name is not tied to a particular SORT template.

EN1^DIP: Print Data

Use EN1^DIP to print a range of entries, in columnar format.

Input Variables

Required

L (Required) A required variable which should be set to zero or some string whose numeric evaluation is zero, e.g., "LIST DRUGS". If set to a text string, the string is used to replace the word "SORT" in the "SORT BY:" prompts, when FileMan asks the user for sort values:

```
LIST DRUGS BY: NAME//
```

DIC (Required) The open global root of the file in the usual format, e.g., "^DIZ(16540," or the file number.

Optional: Sorting and Print Fields

FLDS (Optional) The various fields to be printed. If this parameter is not sent, the user will be prompted for fields to print. FLDS can contain the following:

- The numbers or names of the fields to be printed, separated by commas. These fields are printed in the order that they are listed. Print qualifiers which determine column width, caption contents, and many other features of the output may be included exactly as they are when answering the "PRINT FIELD:" prompt. (See the Print chapter in the *VA FileMan Getting Started Manual* for details on print qualifiers.) For example:

  ```
  FLDS=".01,.03,1;C20"
  ```

  If there are more fields than can fit on one string, FLDS can be subscripted (FLDS(1), FLDS(2), and so forth), but FLDS as a single-valued variable must exist.

- The name of a PRINT template preceded by an open bracket ([) and followed by a close bracket (]). For example:

  ```
  FLDS="[DEMO]"
  ```

BY

(Optional) The fields by which the data is to be sorted. If BY is undefined, the user is prompted for the sort conditions. You can sort by up to 7 fields; that is, you can have up to a 7-level sort.

You can set BY to:

- The numbers or names of the fields separated by commas. Sort qualifiers which determine aspects of the sort and of the printout may be included exactly as they are when answering the "SORT FIELD:" prompt. For example:

  ```
  BY=".01;C1,1"
  ```

  If one of the comma pieces of the BY variable is the @-sign character, the user will be asked for that SORT BY response. So if you want to sort by DIAGNOSIS but allow the user to order the sort within DIAGNOSIS, set BY="DIAGNOSIS,@".

- The name of a SORT template preceded by an open bracket ([) and followed by a close bracket (]). For example:

  ```
  BY="[DEMOSORT]"
  ```

NOTE: You cannot use the name of a SORT template in the BY variable if the BY(0) input variable has been set. If you want to create such complex sorts, you can include the BY(0) information within the SORT template. See the section Storing BY(0) Specifications in SORT Templates, within the Details and Features section of Controlling Sorts with BY(0) at the end of this call.

The name of a SEARCH template, preceded by an open bracket ([) and followed by a close bracket (]). The SEARCH template must have results stored in it. Only those records in the SEARCH template will print, and they will print in IEN order. For example:

```
BY="[DEMOSEARCH]"
```

NOTE: If more than one field is included in the BY variable, separate the fields with commas. The same comma-pieces will identify the field in the FR and TO variables. If, for example, you wanted a sorted report of entries with DOBs in 1960 and with ZIP CODEs in the 90000s, you could define the variables by writing:

```
BY="DOB,ZIP CODE"
FR="01/01/60,90000"
TO="12/31/60,99999"
```

Since the delimiter of BY is a comma, the value placed in the variable should not contain a comma. Therefore, if your field name contains a comma, use the field number in the BY variable instead of its name. For the same reason, if sort from or to values contain commas, the alternate FR(n) and TO(n) input arrays described below should be used instead of the FR and TO input variables.

FR

(Optional) The START WITH: values of the SORT BY fields. If FR is undefined, the user will be asked the START WITH: question for each SORT BY field. If FR is defined, it consists of one or more comma pieces, where the piece position corresponds to the order of the sort field in the BY variable. Each comma piece can be:

- The value from which the selection of entries will begin.
- Null. If a comma piece of FR is null, then the sort will start from the very beginning of the file for that field.
- ?. The question mark as one of the comma pieces causes the "START WITH:" prompt to be presented to the user for the corresponding SORT BY field.
- @. The at-sign indicates that the sort should begin with null values, that is, with entries that have no data on file. If the corresponding piece of the TO variable or array also is set to @, then only entries with null values for this sort field will be selected during the sort. If TO does not contain @, then after the null values, the sort will start at the first non-null value and will go to the value indicated by TO.

NOTE: If BY contains the name of a SORT template and if the developer answered NO to the question SHOULD TEMPLATE

USER BE ASKED 'FROM'-'TO' RANGE... for a field at the time the template was defined, then the information in the FR and TO variables is ignored for that field. Instead, the from/to ranges stored in the sort template are used.

If you customize sorts using BY(0), see special note on FR in that section at the end of this call.

FR(n) (Optional) An alternate way to provide the START WITH: values of the SORT BY fields. If FR is defined, it will override this array. The subscript n corresponds to the comma piece in the BY variable (i.e., the sort by field number). This alternate way of inputting the from and to values allows the use of values containing commas, such as PATIENT NAMEs. Each nth entry in the array corresponds to, and can have the same value as, the nth comma piece in the FR variable. The only difference is that any nth entry, FR(n), can be undefined, causing the START WITH: question to be asked for the nth SORT FIELD.

For example, if you were using the unsubscripted TO and FR variables to do a sort on two fields, you might do as follows:

```
S FR="A,01/01/95",TO="Zz,01/31/95"
```

To set up the same sort using the subscripted forms of TO and FR, you would set them up as follows:

```
S FR(1)="A",FR(2)="01/01/95"
S TO(1)="Zz",TO(2)="01/31/95"
```

NOTE: If you customize sorts using BY(0), see special note on FR in that section at the end of this call.

TO (Optional) The GO TO: values of the SORT BY fields. Its characteristics correspond to the FR variable. If undefined, the user will be asked the GO TO: questions for each SORT BY field. If TO is defined, it consists of one or more comma pieces. Each comma piece can be:

- The value at which the selection of entries will end.
- Null. If TO is null, then the sort will go from FR to the end of the file.

- ?. The question mark as one of the comma pieces causes the "GO TO:" prompt to be presented to the user for the corresponding SORT BY field.

- @. The at-sign indicates that the sort should include null values, that is, entries that have no data on file. If the corresponding piece of the FR variable or array also is set to @, then only entries with null values for this sort field will be selected during the sort. If FR does not contain @, then after the null values, the sort will start at the FR value and include all other non-null values to the end of the file.

NOTE: If BY contains the name of a SORT template and if the developer answered NO to the question SHOULD TEMPLATE USER BE ASKED 'FROM'-'TO' RANGE... for a field at the time the template was defined, then the information in the FR and TO variables is ignored for that field. Instead, the from/to ranges stored in the SORT template are used.

TO(n) (Optional) An alternate way to provide the GO TO: values of the SORT BY fields. If TO is defined, it will override this array. The subscript "n" corresponds to the comma piece in the BY variable. This alternate way of inputting the from and to values allows the use of values containing commas, such as PATIENT NAMEs. Each nth entry in the array corresponds to, and can have the same value as, the nth comma piece in the TO variable. The only difference is that any nth entry, TO(n), can be undefined, causing the GO TO: question to be asked for the nth SORT BY field.

If you customize sorts using BY(0), see special note on TO(n) in that section at the end of this call.

Optional: Miscellaneous Features

DHD (Optional) The header desired for the output. DHD can be one of the following:

- @ if header is not desired.

- @@ if header **and** formfeed are not desired.

- A literal which will be printed, as is, in the upper left hand corner of the printout. The date, page and field headings will be in their normal places.
- A line of M code which must begin with a write statement, e.g., DHD="W ?0 D ^ZZHDR".
- A PRINT template name preceded by an open bracket ([) and followed by a close bracket (]). In this case, the template replaces all parts of the header that VA FileMan normally generates.
- Two PRINT templates separated by a minus sign. The first will be used as the header and the second will be used as the trailer. For example:

  DHD="[HEADER]-[TRAILER]"

DIASKHD (Optional) If this variable is defined, the user will be prompted to enter a header. Set it equal to null ("""). If this variable is undefined, the user will not have the opportunity to change the header on the print.

DIPCRIT (Optional) If this variable is set to 1, the SORT criteria will print in the header of the first page of the report.

PG (Optional) Starting page number. If variable is undefined, page 1 will be assumed.

DHIT (Optional) A string of M code which will be executed for every entry after all the fields specified in FLDS have been printed.

DIOEND (Optional) A string of M code which is executed after the printout has finished but before returning to the calling program.

DIOBEG (Optional) A string of M code which is executed before the printout starts.

DCOPIES (Optional) If %ZIS chooses an SDP device, and if multiple copies are desired, you can call for them by setting DCOPIES equal to the number (greater than one) of copies desired. For more information about SDP devices, see the *Kernel Systems Manual*.

IOP

(Optional) EN1^DIP calls the ^%ZIS entry point to determine which device output should go to. This requires user interaction unless you preanswer the DEVICE prompt. You can do this by setting IOP equal to the name of the device (as it is stored in the DEVICE file) to which the output should be directed. You can also set IOP in any of the additional formats recognized by ^%ZIS to specify the output device (see the *Kernel Systems Manual* for more information on ^%ZIS and IOP).

If you need to call ^%ZIS beforehand to obtain the name of the device in question from the user, call it with the %ZIS N flag set so that ^%ZIS doesn't actually open the device. The name of the device is then returned in the ION output variable. EN1^DIP will open and close the device you specify in IOP on its own; don't open it yourself beforehand.

In addition to setting IOP equal to a device for printing, you can use this variable (in conjunction with the DQTIME variable described immediately below) to queue the printing of a report. This functionality is only available if Kernel is present. Also, you must set up all of the input variables for EN1^DIP so that the user is not asked any questions. For example, the BY, FR, and TO variables must be defined. To establish queuing, IOP should equal Q;output device. For example:

```
S IOP="Q;MY PRINTER - NLQ".
```

DQTIME

(Optional) If output is queued, this variable contains the time for printing. You can set it equal to any value that %DT recognizes. For example:

```
S DQTIME="NOW"
```

OR:

```
S DQTIME="T@11PM"
```

DIS(0)

(Optional) You can screen out certain entries so that they do not appear on the output by setting the optional array DIS. The first subscript in this array can be 0 (zero). This variable

(as well as all the others) contains an executable line of M code which includes an IF-statement. If the execution of the IF sets

$T to 1, then the entry will print. The internal number of the entry being processed is in D0.

DIS(n) (Optional) You can set other elements in the DIS array: DIS(1), DIS(2), DIS(3), etc. The subscripts must be consecutive integers starting at 1. Again, they must contain M code that sets $T. If many elements are defined, then DIS(0) (if it exists) must be true and any one of the other elements in the array must be true for the entry to print.

DISUPNO (Optional) If this variable is set to 1 and if no records are found within the sort ranges specified for the print, the report header and the "No Records to Print" message is not printed.

DISTOP (Optional) If Kernel is present, by default, prints queued through the EN1^DIP call can be stopped by the user with a TaskMan option. However, if this variable is set to 0, users will not be able to stop their queued prints.

DISTOP can also be set equal to M code that will be executed once near the start of a queued print. If the code sets $T to true, the user will be able to stop the job; if $T is false, the user will not be able to. For example:

```
S DISTOP="I DUZ(0)=""@"""
```

would mean that only those with programmer access could stop the print.

DISTOP("C") (Optional) If the user stops a queued print job by using TaskMan's option, code in this optional variable will be executed before the output device is closed. It might, for example, do clean up necessary because the job did not run to completion.

Optional: Controlling Sorts with BY(0)

BY(0)

L(0)

FR(0,n)

See the section called CONTROLLING SORTS WITH BY(0) (In Detail) at the end of this call for more information.

TO(0,n) **DISPAR(0,n)** **DISPAR(0,n,"OUT")**	

Output Variables

None

NOTE: Unlike most calls, EN1^DIP kills all the input variables before it quits. You do not have to kill them.

Details and Features

Input Variables to Control Sorts

You can use a special set of input variables to:

- Preselect a set of records for printing.
- Preselect the order that these records should be printed in.

The set of variables for controlling sorts is:

BY(0), L(0), FR(0,n), TO(0,n), DISPAR(0,n), and DISPAR(0,n,"OUT")

Please see the Controlling Sorts with BY(0) section at the end of this call for more information.

Setting up BY, FR, and TO Variables to Sort within a Multiple

If you have a file like:

```
  .01 PARENT NAME
 1 SPOUSE (mult.)
    .01 SPOUSE NAME
    1 SPOUSE DOB
    2 CHILDREN (mult.)
          .01 CHILDS NAME
          1 CHILDS DOB
          2 CHILDS SEX
          3 CHILDS NICKNAME
 2 PARENT NICKNAME
```

And you wish to sort on the NICKNAME field for CHILDREN, from "A" to "Z", then by the PARENT

NICKNAME field from "B" to "E". You set:

```
BY = "1,2,3,2
FR = "A,B"
TO = "Z,E"
```

You must put in all field numbers to get down to the multiple in the BY (1,2,3), but then it pops you out of the multiple so that the following number '2' in the BY gets you field 2 at the top level (PARENT NICKNAME), rather than field 2 within the lowest multiple (SEX).

But note the FR and TO: here you just put the starting and ending values for the two fields on which you wish to sort.

NOTE: This same logic does not work on the FLDS multiple. It is suggested that in order to print fields within a multiple, the print logic should be set up in a PRINT template.

Using EN1^DIP to Print Audit Trails

The audit files are structured differently than other FileMan files. To print audit trails for a file's data or Data Dictionary, the DIC variable must contain the global location of the requested audit file and the file number of the file that was audited as the open root.

To print a data audit trail for File #662001, set DIC="^DIA(662001,". To print the DD audit trail, set DIC="^DDA(662001,". The other input variables are set as for a normal print. Remember that the fields being printed and sorted come from the audit files, not from the file for which the audit trail was recorded.

EN1^DIP: CONTROLLING SORTS WITH BY(0) (In Detail)

Ordinarily, you control the way EN1^DIP sorts output using the BY, FR, and TO input variables. This lets you sort based on field values, a previous sort stored in a SORT template, or on the records stored in a SEARCH template.

The BY(0) feature allows you to control the sort. With BY(0), you can force VA FileMan to sort using an existing compound index (i.e., one that indexes more than a single data field) for efficiency. Or, use of BY(0) allows you to pre-sort a list of record numbers in a global and pass this pre-sorted list to EN1^DIP. This lets you pre-sort reports in any way that you can use subscripts to sort a global. The only

limitation is that the total number of subscripts in the global that you sort by must be seven or less.

The two main ways in which the BY(0) feature should be used are as follows:

- Set BY(0) to the global location of an **existing FileMan index**. In particular, this lets you sort based on a MUMPS cross-reference or a compound cross-reference defined on the INDEX file (not possible otherwise). Since the sorting is already done in advance, any such prints are very fast.
- Set BY(0) to the global location of a list of records you create "**on the fly**." This lets you sort the records in any order you want, and also lets you easily limit the number of records by pre-selecting them.

Input Variables for Sorting with BY(0)

BY(0)

(Optional; Required for BY(0) sorts) Set this variable to an open global root. The open global root should be the static part of a global; a list of record numbers must be stored at a descendent subscript level.

```
^DIZ(662001,"E","ALBERT",1009)
^DIZ(662001,"E","ANDREA",339)
^DIZ(662001,"E","ANDREW",552)
--------------- ----------------
<-static part-> <-dynamic part->
```

In the example just above, you would set BY(0) to '^DIZ(662001,"E",'.

There can be intervening subscript levels between the static, fixed global root and the subscript level where the list of records numbers is stored. Any intervening subscript levels define a sort order. Use the L(0) input variable to tell FileMan the number of dynamic subscript levels it needs to sort through (see L(0) description below).

Alternatively, you can set BY(0) to the name of a SEARCH template, in [brackets]. This tells VA FileMan to sort on the list of record numbers contained in the corresponding SEARCH template entry in the ^DIBT global.

BY(0) affects your sorts as follows:

It restricts the possible records for printing to those in the specified list.

When you set BY(0) to a static global reference, each intervening subscript level (between the static part of the global reference and the subscript level containing record numbers) defines a sort level, starting from the highest intervening subscript level.

BY(0) for a VA FileMan Index

If you set BY(0) to sort based on an existing FileMan-maintained cross-reference, make sure the subscript you set L(0) to point to is in fact the location where FileMan stores its list of records (when sorting on a regular single-field index, L(0) should be 2).

BY(0) for a List of Records "On the Fly"

If you build your own list of sorted records on the fly in a temporary global (as opposed to setting BY(0) to a VA FileMan-maintained cross-reference) it's best not to let the final subscript of your static global reference be "B". For more information, see the discussion in the Details and Features section below.

NOTE: If you are using both the BY and BY(0) input variables, don't set BY to the name of a template; an error message will print or hard errors could result.

L(0)

(Optional; Required if BY(0) is set to an open global root.)

Use L(0) to specify the number of dynamic subscript levels that exist beyond the static global root, including the subscript level containing the list of record numbers. The minimum value of L(0) is 1.

EN1^DIP lets you sort by up to 7 subscripts; therefore the maximum value of L(0) is 8.

For example, if BY(0) refers to a regular "E" index on a file -- '^DIZ(662001,"E",' -- you should set L(0)=2 -- that is, one for the subscript containing the (dynamic) value

of the field being cross-referenced, plus one for the record number.

FR(0,n)

(Optional) To select only a subset of records at a given subscript level "n", you can use FR(0,n) and/or TO(0,n). For "n" equal to any of the "n" dynamic sorting subscript levels in the global specified by BY(0), you can set FR(0,n) to the sort-from value for that subscript level.

This restricts the printed records to those whose subscript values at subscript level n sort the same or greater than the value you set into FR(0,n). If FR(0,n) is undefined for any subscript n, the sort on that subscript level begins with the first value for that subscript.

NOTE: These values must be in internal format, as they are stored in the subscript of the index or global defined by BY(0).

TO(0,n)

(Optional) This variable contains the ending value (the sort-to value) for any of the "n" dynamic sorting subscripts in the global specified by BY(0). If TO(0,n) is undefined for any subscript "n", the sort on that subscript level ends with the last value for that subscript.

NOTE: These values must be in internal format, as they are stored in the subscript of the index or global defined by BY(0).

DISPAR(0,n)

(Optional) Like the FR(0,n) and TO(0,n) variables, this variable array can be set for any of the "n" dynamic sorting subscripts in the global specified by BY(0). This array allows you to create subheaders for the sorting subscripts in the global. In order to create a sub-header, you must define a title for the subscript, as VA FileMan has no knowledge of the subscripts. Each entry in the array can have information in two ^-pieces.

The first piece contains the sort qualifiers that are normally entered interactively before a sort field (see the User Manual for more information.) Two of the sort

qualifiers can be used here: "!" to number the entries by sort value and "#" to page break when the sort values changes.

The second piece contains the sort qualifiers that are normally entered interactively after the sort field. In order to print a subheader, you must enter literal subheader "caption" (e.g., ;"Station/PO Number: "). To have no subheader text other than the subheader value, use a null caption (e.g., ;""). You can also use the sort qualifiers ;Cn ;Ln or ;Sn, (see the User *Getting Started Manual* for more information.)

The subheaders defined in DISPAR(0,n) cannot be suppressed.

DISPAR(0,n,"OUT") (Optional) If a literal title is input to DISPAR(0,n) above, then you can also enter M code to transform the value of the subscript from the global before it is printed as a subheader. It acts like an OUTPUT transform. At the time of execution, the untransformed value will be in Y. The code should put the transformed value back into Y. Any other variables used in the code should be NEWed.

Examples

Example 1

Suppose you have a simple MUMPS cross-reference that inverts dates so that the values in the cross-reference are 99999999-date. The cross-reference might look something like:

```
^DIZ(662001,"AC",97069889,2)=""
^DIZ(662001,"AC",97969898,3)=""
^DIZ(662001,"AC",97969798,1)=""
    ...etc.
```

If you wanted to sort all entries by this inverse date and to convert the date values into a readable format for the subheader, you would set up the variables for the EN^DIP call like this:

```
>S DIC="^DIZ(662001,",L=0,FLDS="your field list"
>S BY(0)="^DIZ(662001,""AC"","
```

```
>S L(0)=2
>S DISPAR(0,1)="^;""DATE"""
>S DISPAR(0,1,"OUT")="S:Y Y=99999999-Y S Y=$$FMTE^XLFDT(Y)"
```

Example 2

Suppose you have a list of record numbers in a global that looked like this:

```
^TMP($J,1)=""
^TMP($J,3)=""
^TMP($J,35)=""
^TMP($J,39)=""
    ...etc.
```

If you wanted to print those records sorted by the .01 field of the file, you would:

```
>S DIC="^DIZ(662001,",L=0,BY=.01,(FR,TO)="",FLDS="your
       field list"
>S BY(0)="^TMP($J,"
>S L(0)=1
```

Example 3

Suppose you have a MUMPS multifield-style cross-reference, with subscripts based on the values of two fields. The first field in the subscript is free-text, and the second is a number. The cross-reference might look like:

```
^DIZ(662001,"AD","ANY",4.99,5)=""
^DIZ(662001,"AD","ANYTHING",1.3,2)=""
^DIZ(662001,"AD","ANYTHING",1.45,1)=""
^DIZ(662001,"AD","SOMETHING",.4,10)=""
    ...etc.
```

You want to sort from value "A" to "AZ" on the free-text field and from 1 to 2 on the numeric field. Also, you want to print a subheader for the numeric field. You could set your variables like this:

```
>S DIC="^DIZ(662001,",L=0,FLDS="your field list"
>S BY(0)="^DIZ(662001,""AD"","
>S L(0)=3
>S FR(0,1)="A",TO(0,1)="AZ"
>S FR(0,2)=1,TO(0,2)=2
>S DISPAR(0,2)="^;""NUMBER"""
>S DISPAR(0,2,"OUT")="S Y=$J(Y,2)"
```

Details and Features

Sorting on MUMPS Cross-references

The BY(0) feature is designed to let you pre-sort your FileMan reports using MUMPS cross-references. As long as the MUMPS cross-reference has 0 to 7 dynamic (sorting) subscripts, followed by the record numbers stored in a final subscript level, you can order your reports based on that cross-reference using BY(0).

While you may have used MUMPS cross-references in the past only for sorting hard-coded reports, you may want to consider using them with FileMan-based reports as well.

Sorting a Compound Cross-reference Defined in the INDEX file

The BY(0) feature will allow you to sort using a compound cross-reference on the new INDEX file (a compound cross-reference is one that indexes more than one data field). This feature will let you use any index that has no more than 7 data valued subscripts.

Sorting Using One or More Subscript Levels

Each intervening subscript level between the static part of the open global root in BY(0) and the record number subscript level serves as one sort level, starting with the highest subscript level.

In example 3 above, the records would sort by the value of the free-text field stored in the first dynamic subscript, and within that by the value of the numeric field stored in the second dynamic subscript.

Additional Sorting with BY, FR, and TO

When using BY(0), you can still sort in the usual way (setting BY, FR, and TO) to further sort and limit the range within the list provided by BY(0). Note that if you set BY(0), BY cannot contain the name of a SORT template. If your sort is complicated, see the documentation below on "Storing BY(0) specifications in SORT Templates."

VA FileMan selects only the list of records specified by BY(0) and its associated variables. FileMan accepts as-is the sort sequence created by any dynamic subscripts in the global specified in BY(0). Then within that sort sequence, it further sorts the records by the information provided in the BY, FR, and TO variables.

You can only sort by up to 7 sort levels in EN1^DIP, so the number of subscripts you sort by using BY(0) combined with the number of fields you sort by using BY must not total more than 7.

If BY(0) has been defined without BY, FR, and TO, the user will *not* be prompted for the SORT BY or FROM/TO ranges.

Storing BY(0) Specifications in SORT Templates

You can store the BY(0) information in a SORT template, in order to design more complicated sorts. This allows you to sort using the global described in the BY(0) variable, and within those subscripts, to sort by additional fields and to save the entire sort description into a template. You need programmer access to do this.

In FileMan's sort dialog (with programmer access), at the SORT BY: prompt, you can enter the characters BY(0) as shown in the example immediately below. When you enter BY(0), you are then prompted for the BY(0), L(0) and all related values, exactly the same as if they were entered as input variables to the EN1^DIP call.

```
Select OPTION: 2  PRINT FILE ENTRIES

OUTPUT FROM WHAT FILE: ZZTAMI TEST//
SORT BY: NAME// BY(0)

BY(0): // ^DIZ(662001,"H",
L(0): //2

Edit ranges or subheaders? NO// YES

SUBSCRIPT LEVEL: 1// 1
FR(0,n): // 2690101
TO(0,n): // 2701231
DISPAR(0,n) PIECE ONE: //
DISPAR(0,n) PIECE TWO: // ;"Date of Birth: "
DISPAR(0,n,OUT): // S Y=$$FMTE^XLFDT(Y,1)

Edit ranges or subheaders? NO//

BY(0)=^DIZ(662001,"H",     L(0)=2

SUB: 1  FR(0,1): 2690101
       TO(0,1): 2701231
       DISPAR(0,1) PIECE ONE:
       DISPAR(0,1) PIECE TWO: ;"Date of Birth: "
       DISPAR(0,1,OUT): S Y=$$FMTE^XLFDT(Y,1)

OK? YES//
Enter additional sort fields? NO// YES
```

```
WITHIN BY(0), SORT BY: NAME
START WITH NAME: FIRST//
      WITHIN NAME, SORT BY:

STORE IN 'SORT' TEMPLATE: ZZTAMIBY0
```

When you enter BY(0), you are prompted for BY(0) and L(0). In addition, you're asked if you want to edit ranges or subheaders. This lets you enter the FR(0,n), TO(0,n), DISPAR(0,n) and DISPAR(0,n,"OUT") values for various subscript levels. This lets you specify all the aspects of sorting using BY(0). You can store this criteria in a SORT template. If you answer YES to "Enter additional sort fields?", you will be allowed to enter additional sort fields, exactly the same as you would when creating a SORT template without the BY(0) features.

The functionality of BY(0) interactively or in a SORT template is identical to its functionality in the EN1^DIP programmer call.

An error results if, in a call to EN1^DIP, you sort by a SORT template that contains BY(0) sort criteria, and also use BY(0) as an input variable.

NOTE: The sort ranges associated with subscripts in the BY(0) global or index can be set dynamically by setting the FR(0,n) and TO(0,n) input variables. These input variables will override any sort ranges set in the template.

The "SUBSCRIPT LEVEL" prompt refers to the position of the data value in the global or index. Thus, entering a value for FR(0,n) when the SUBSCRIPT LEVEL is 1, sets the "from" value for the first data valued subscript.

Use the documentation for the BY(0) and related input variables for additional help. Also be sure to use online ? and ?? help.

The following is an example of how to call EN1^DIP when the BY(0) information is contained in a template:

```
S DIC="^DIZ(16600,",L=0,BY="[ZZTEST]",FR(0,1)=
  70001,FLDS=.01
D EN1^DIP
```

BY(0) "Don'ts"

You should not use BY(0) if you are merely setting it to the global location of an existing regular cross-reference. You will not gain any speed, because FileMan's built-in sort optimizer already knows to sort on regular cross-references.

Also, don't specify a field's regular cross-reference as the global reference in BY(0) to sort on, and then sort on the same field using BY, FR, and TO. This actually increases the amount of work FileMan needs to do!

"On the Fly" Globals Whose Static Global Reference Ends with "B"

If you build your own list of sorted records on the fly in a temporary global (as opposed to setting BY(0) to a VA FileMan-maintained cross-reference) it's best not to let the final subscript of your static global reference be "B".

This will avoid problems that might be caused by VA FileMan's special handling of the "B" index for mnemonic cross-references.

^DIPT: Print/Display

The PRINT template file contains a computed field labeled PRINT FIELDS which displays a PRINT template exactly as it was entered. Use this entry point to make this display immediately available to a user.

Input Variable

D0 (Required) Set D0 equal to the internal number of the template in the PRINT template file. For example, to display the PRINT template whose record number is 70:

```
S D0=70 D ^DIPT
```

NOTE: Use the number 0 (zero) not the letter O in this variable name.

DIBT^DIPT: Sort/Display

The SORT template file contains a computed field labeled SORT FIELDS which displays a SORT template exactly as it was entered. Use this entry point to make this display immediately available to a user.

Input Variable

D0 (Required) Set D0 equal to the internal number of the template in the SORT template file. For example, to display the SORT template whose record number is 70:

```
S D0=70 D DIBT^DIPT
```

NOTE: Use the number 0 (zero) not the letter O in this variable.

^DIPZ: Compile

PRINT templates can be compiled into M routines just as INPUT templates can be. The purpose of this DIPZ code generation is simply to improve overall system throughput.

Only regular PRINT templates can be compiled. You cannot compile FILEGRAM, EXTRACT, Selected Fields for Export, or EXPORT templates that are also stored in the PRINT template file.

Call the ^DIPZ routine and specify the maximum routine size, the name of the PRINT template to be used, the name of the M routine to be created, and the margin width to be used for output (typically 80 or 132). If you specify the routine name XXX and if more code is generated than can fit into that one routine, overflow routines (XXX1, XXX2, etc.) will be created. Routine XXX may call XXX1, XXX2, etc.

Once DIPZ has been used to create a hard-coded output routine, that routine is usually invoked automatically by VA FileMan within the Print File Entries and Search File Entries options and when called at EN1^DIP whenever the corresponding PRINT template is used. The compiled routines are not used if a user-specified output margin width is less than the compiled margin. Also, if the template is used with ranked sorting (i.e., the ! sort qualifier is used), the compiled version is not used.

As with compiled INPUT templates, as soon as data dictionary definitions of fields used in the PRINT template are changed, the hard-core routine(s) is/(are) compiled immediately.

Invoking Compiled PRINT Templates

A DIPZ-compiled M routine may be called by any program that passes to it the variables DT, DUZ, IOSL (screen length), U (^), and D0 (the entry number to be displayed). Additionally, the variable DXS must be killed before calling the routine and after returning from it. The compiled routine writes out its report for that single entry. However, routines compiled from templates that include statistical totals cannot be called in this way.

EN^DIPZ: Print/Compile

PRINT templates can be compiled into M routines just as INPUT templates can be. The purpose of this DIPZ code generation is simply to improve overall system throughput.

Only regular PRINT templates can be compiled. You cannot compile Filegram, Extract, Selected Fields for Export, or EXPORT templates that are also stored in the PRINT template file.

This entry point recompiles a PRINT template without user intervention by setting the input variables:

Input Variables

X	The routine name.
Y	The internal number of the template to be compiled.
DMAX	The maximum size the compiled routines should reach. Consider using the $$ROUSIZE^DILF function to set this variable.

D^DIQ: Display

This entry point takes an internal date in the variable Y and converts it to its external form. This call is very similar to DD^%DT.

Input Variable

Y (Required) Contains the internal date to be converted. If this has five or six decimal places, seconds are automatically returned.

Output Variable

Y External form of the date or date/time value, e.g., JAN 01, 1998.

DT^DIQ: Display

This call converts the date in Y exactly like D^DIQ. Unlike D^DIQ, however, it also writes the date after it has been converted.

Input Variable

Y	(Required) Contains the internal date to be converted. If this has five or six decimal places, seconds are automatically returned.

Output Variable

Y	External form of the date or date/time value, e.g., JAN 01, 1998.

EN^DIQ: Display

This entry point displays a range of data elements in captioned format, to the current device. The output from this call is very similar to that of the Inquiry to File Entries option (described in the Inquire Option section of the *VA FileMan Getting Started Manual*).

Input Variables

DIC (Required) The global root of the file in the form ^GLOBAL(or ^GLOBAL(#,

If you are displaying an entry in a subfile, set DIC to the full global root leading to the subfile entry, including all intervening subscripts and the terminating comma, up to but not including the ien of the subfile entry to display.

DA (Required) If you are displaying an entry at the **top level** of a file, set DA to the internal entry number of the file entry to display.

If you are editing an entry in a **subfile**, set up DA as an array, where DA=entry number in the subfile to display, DA(1) is the entry number at the next higher file level,...DA(n) is the entry number at the file's top level.

DR (Optional) Names the global subscript or subscripts which are to be displayed by DIQ. If DR contains a colon (:), the range of subscripts is understood to be specified by what precedes and follows the colon. Otherwise, DR is understood to be the literal name of the subscript. All data fields stored within, and descendent from, the subscript(s) will be displayed, even those which normally have Read access security protection.

If DR is not defined, all fields are displayed.

DIQ(0) (Optional) You can include the following flags in this variable to change the display of the entry:

A To display **A**udit records for the entry.

C To display **C**omputed fields.

R To display the entry's **R**ecord number (IEN).

Y^DIQ: Display

This entry point converts the internal form of any data element to its external form. It works for all FileMan data types, uses output transforms, and follows pointer trails to their final resolution. The equivalent Database Server call is $$EXTERNAL^DILFD.

Input Variables

Naked Global Reference

The naked global reference must be at the zero node of the data dictionary definition which describes the data [i.e., it must be at ^DD(File#,Field#,0)].

See the description of input variable C below for an example of setting the naked reference.

C

Set C to the second piece of the zero node of the data dictionary which defines that element. Typically, the programmer would:

```
S C=$P(^DD(file#,field#,0),U,2)
```

and then:

```
D Y^DIQ
```

This set will correctly set the naked global reference as described above.

Y

Set Y to the internal form of the value being converted. This is the data that you want to convert to external form.

Output Variable

Y

The external form of the value. Basically, Y is changed from internal to external.

EN^DIQ1: Data Retrieval

This entry point retrieves data from a file for a particular entry.
NOTE: The equivalent Database Server calls are GETS^DIQ and $$GET1^DIQ.
It is your responsibility to kill the output array, ^UTILITY("DIQ1",$J), before and after using this call.

Input Variables

DIC The file number or global root.

DR A string specifying the data fields to retrieve for the given entry. The DR-string may contain:

A single number corresponding to the internal number of a field in the file.

A range of field numbers, in the form M:N, where M is the first and N the last number of the inclusive range. All fields whose numbers lie within this range will be retrieved.

A combination of the above, separated by semicolons. If field numbers .01, 1, 2, 4, 10, 11, 12, 13, 14, 15, and 101 exist for a file, and you want to retrieve the data in these fields, simply write:

```
S DR=".01;1;4;10:15;101"
```

DR(subfile_number) If you want to retrieve values from fields from a subentry in a multiple field, include the top-level field number for the multiple in DR. Then, include the multiple's subfield numbers whose values you want to retrieve in a node in DR, subscripted by the subfile number.

See also DA(subfile_number) below for how to specify which subfile entry to retrieve.

For example, if you want to retrieve data from subfields .01 and 7 for subentry 1 from field 4 which defines the multiple 16000.02, then you write:

```
S DIC=16000,DR="4",DA=777
S DR(16000.02)=".01:7",DA(16000.02)=1
D EN^DIQ1
```

DA The internal number of the entry from which data is to be extracted.

DA(subfile_number) If you want to retrieve values from fields from a subentry in a multiple, set DA to the top-level entry number. Then, include the subentry number in a node in DA, subscripted by the subfile number. See DR(subfile_number) above for how to specify which fields in the subfile entry to retrieve.

You can descend one or more subfile levels; however, you can only retrieve values for one subentry at any given subfile level. The full path from the top level of the file to the lowest-level subfile entry must be fully specified in nodes in DR and DA.

DIQ (Optional) The local array name into which the field values will be placed. ^UTILITY("DIQ1",$J, will be used if DIQ is not present. This array name should not begin with DI.

DIQ(0) (Optional) This variable is used to control which is returned: internal values, external values, or both. DIQ(0) also indicates when null values are not returned. The DIQ(0) string can contain the values that follow:

I return **I**nternal values

E return **E**xternal values

N do not return **N**ull values

Output

The format and location of the output from EN^DIQ1 depends on the status of input variables DIQ and DIQ(0) and on whether or not a word processing field is involved.

DIQ and DIQ(0) undefined

Output into:

```
^UTILITY("DIQ1",$J,file#,DA,field#)=external value
```

This is for backward compatibility. Each field requested will be defined in the utility global but the value may be null. The only exception to this would be when DA held the number of an entry which does not exist. In that case, nothing is returned. The

values returned are the external values. Printable values—pointers, sets of codes, etc.—are resolved; dates are in external format.

DIQ(0) defined, DIQ undefined

Output into:

```
^UTILITY("DIQ1",$J,file#,DA,field#,"E")=external value
^UTILITY("DIQ1",$J,file#,DA,field#,"I")=internal value
```

If DIQ(0) contains "E", the external value is returned with a final global subscript of "E".
If DIQ(0) contains "I", the internally stored value is returned with a final global subscript of "I". The internal value is the value stored in the file, for example, the record number of the entry in the pointed-to file, not the resolved value of the pointer. Since computed fields store no data, no nodes are returned for computed fields.
If DIQ(0) contains "N", no nodes are set for either internal or external values if the field is null.
If DIQ(0) contains both "I" and "E", generally two nodes are returned for each field: one with the internal value, one with the external value. However, no nodes are produced for the internal value if the field is computed and no nodes are produced at all for null-valued fields if DIC(0) contains "N". Nodes are subscripted as described above.

DIQ defined

The output is similar except that the data is stored in the specified local array. So if DIQ(0) is not defined, then the output is:

```
@(DIQ(file#,DA,field#))=external value
```

If DIQ(0) is defined, then the output is:

```
@DIQ(file#,DA,field#,"E")=external value
@DIQ(file#,DA,field#,"I")=internal value
```

Word Processing Field

Output from a word processing field will only be an external value. The status of DIQ(0) has no effect. If DIQ is not defined, it goes into the global nodes that follow:

```
^UTILITY("DIQ1",$J,file#,DA,field#,1)
^UTILITY("DIQ1",$J,file#,DA,field#,2)
     .
     .
     .
```

If DIQ is defined, it goes into:

```
@DIQ(file#,DA,field#,1)=External Value 1
@DIQ(file#,DA,field#,2)=External Value 2
@DIQ(file#,DA,field#,3)=External Value 3
@DIQ(file#,DA,field#,4)=External Value 4
    .
    .
    .
```

^DIR: Reader

DIR is a general purpose response reader that can be used to issue a prompt, read input interactively, perform syntax checking on the input, issue error messages or help text, and return input in a processed form. Its use is recommended to standardize user dialog and to eliminate repetitive coding.

DIR is reentrant: A DIR call may be made from within a DIR call. To reenter DIR, use the NEW command to save the DIR array (NEW DIR) before setting input variables and making the second call.

A. **Input and Output Variables (Summary)**
B. **Required Input Variables (Full Listing)**
C. **Optional Input Variables (Full Listing)**
D. **Output Variables (Full Listing)**
E. **Examples**

A. Input and Output Variables (Summary)

Input Variables-Required

DIR(0)	Required: First character of Piece-1 (first 3 characters for DD-type)	Read type
	Optional: Subsequent characters of Piece-1	Input modifiers
	Optional: Piece-2	Input parameters
	Optional: Piece-3	INPUT transform

Input Variables-Optional

DA	For DD-type reads, can specify entry from which to retrieve default value
DIR("A")	Programmer-supplied prompt to override default
DIR("A",#)	Array for information to be displayed before the prompt
DIR("B")	Default response
DIR("L")	For set-of-code fields: programmer-specified format to display

DIR("L",#)	codes.
DIR("S")	Screen for pointer, set-of-code, and list/range reads
DIR("T")	Time specification to be used instead of DTIME
DIR("?")	Help displayed when the user enters a single question mark
DIR("?",#)	
DIR("??")	Help displayed when the user enters a double question mark

Output Variables-Always Returned

X	Unprocessed user response
Y	Processed user response

Output Variables-Conditionally Returned

Y(0)	External form of response for set, pointer, list, and date
DTOUT	Defined if the user times out
DUOUT	Defined if the user entered an up-arrow
DIRUT	Defined if the user entered an up-arrow, pressed the Enter/Return key, or timed out
DIROUT	Defined if the user enters two up-arrows

B. Required Input Variables (Full Listing)

DIR(0 DIR(0) is the only required input variable. It is a three piece variable. The first character of the first piece must be defined (or first 3 characters for DD-type). Additional characters of the first piece and the second two pieces are all optional.

The first character of the first up-arrow piece indicates the type of the input to be read. The second piece describes parameters, delimited by colons, to be applied to the input. Examples are maximum length for free text data or decimal digits for numeric data. The third piece is executable M code that acts on the input in the same manner as an INPUT transform. The acceptable types are shown below:

DIR(0) (Summary)

DIR(0) Read Type	Piece-1		Piece-2	Piece-3
	First Charac-ter (re-quired)	**Subsequent Characters (optional)**	**Format**	**Executable M code (optional)**
Date	D	A,O	Minimum date:-Maximum date:%DT	code
End-of-Page	E	A	--	--
Free-text	F	A,O,U,r	Minimum length: Maximum length	code
List or range	L	A,O,C	Minimum: Maximum: Maximum decimals	code
Numeric	N	A,O	Minimum: Maximum: Maximum decimals	code
Pointer	P	A,O,r	Global Root or #:DIC(0)	code
Set of Codes	S	A,O,X,B	Code: Stands for;Code: stands for;	code
Yes/No	Y	A,O	--	code
DD	#,#	A,O,r	--	code

DIR(0) (Detailed Explanation)

Piece-1 of DIR(0) (Subsequent Characters are Optional):

The first up-arrow piece of DIR(0) can contain other parameters that help to specify the nature of the input or modify the behavior of the reader. These characters must appear after the character indicating type (or after the field number if it is a DD type). They are described below and examples are provided later in this section):

A Indicates that nothing should be **A**ppended to the programmer-supplied prompt DIR("A"), which is described below. If there is no DIR("A"), then no prompt is issued.

B Only applies to a set of codes; indicates that the possible choices are to be listed horizontally after the prompt.

C Only applies to list reads. The values returned in Y and the Y() array are **C**ompressed. They are not expanded to include each individual number, rather, ranges of values are returned using the hyphen syntax. This is similar to the format in which the user can enter a range of numbers.

This flag is particularly useful when a user may select many numbers, e.g., when decimals are involved. The call is much faster and the possibility of the local symbol table filling up with nodes in the Y() array is eliminated.

O Indicates that a response is **O**ptional. If this is not included, then a null response is not allowed. For DD type reads, the O is automatically included if the field in question is not a required field.

r If user does not choose to accept the default, they must type in their entire response. They will not get the "Replace-With" prompt, no matter how long the default response is.

U Only applies to free text reads. It allows the user response to contain ^ (**U**p-arrow). A leading up-arrow aborts the read and sets DUOUT and DIRUT whether or not U is in DIR(0). However, U allows ^s to be embedded in the user response.

X Only applies to set of codes. Indicates a request for an e**X**act match. No lower- to uppercase conversion is to be done.

Piece-2 of DIR(0) (Optional)

Qualifying limits on user response are as described in summary table above.

Piece-3 of DIR(0) (Optional)

The third piece of DIR(0) is executable M code that acts like the INPUT transform of a field in a data dictionary. The value that was entered by the user is contained in the variable X. The code can examine X and, if it is not appropriate, should KILL X. If X is undefined after the execution of the third piece of DIR(0), the reader knows that the input was unacceptable, issues a help message, and re-asks for input. It is unnecessary to put checks for minimum and maximum or length in the third piece. These should be specified in the second piece of DIR(0). An example of DIR(0) with all three pieces is:

```
S DIR(0)="F^3:30^K:X'?.U X"
```

which says that if the input is not all uppercase, then the data is unacceptable. The check for a length from 3 to 30 characters takes place automatically because of the second piece. The third piece is not executed if the specifications in the second piece are not met. If the user combines the DD data type with a third piece in DIR(0), for example:

```
S DIR(0)="19,.01^^K:X'?1""DI"" X"
```

then the third piece of DIR(0) is not executed until after the INPUT transform has been executed and X was not Killed by the transform.

C. Optional Input Variables (Full Listing)

DA

(Optional) For DD-type reads only, if DIR("B") is not set, you may retrieve a value from the database to display as a default. Identify the entry from which the value should come by setting the DA variable to its record number. If a subfile is involved, set up a DA() array where DA equals the record number for the lowest level subfile, DA(1) for the next higher, and so on.

NOTE: Although you can retrieve defaults from the database by using DA, the values in the database are not changed by ^DIR calls.

DIR("A")

(Optional) The reader provides a generic default prompt for each type, e.g., enter a number or enter response. To issue a

more meaningful prompt, DIR("A") can be set to a character string that more clearly indicates the nature of the data being requested. For example, setting the following:

```
S DIR("A")="PRICE PER DISPENSE UNIT: "
S DIR(0)="NA^0:5:2"
```

causes the prompt to appear as:

```
PRICE PER DISPENSE UNIT:
```

DIR("A",#)

(Optional) If you want to issue a longer message before actually reading the input, you can set the DIR("A",#) array in addition to DIR("A"). The #'s must be numeric. After the array has been displayed, DIR("A") is issued as the prompt for the read. It is necessary for DIR("A") to be set if the programmer is to use this array. For example, setting the following:

```
S DIR("A")="PRICE PER DISPENSE UNIT:"
S DIR("A",1)="Enter price data with two decimal points."
S DIR("A",2)="Cost calculations require this precision."
```

causes the following dialog to appear to the user:

```
Enter price data with two decimal points.
Cost calculations require this precision.
PRICE PER DISPENSE UNIT:
```

DIR("B")

(Optional) Set this variable to the default response for the prompt issued. It appears after the prompt and before the // (double slashes). If the user simply presses the Enter/Return key, the default response is accepted by the reader.

DIR("L")
DIR("L",#)

(Optional) Only applies to set-of-codes fields. Lets you replace the standard vertical listing of codes that the Reader displays with your own listing. It is up to you to ensure that the contents of the DIR("L") array match the codes in the second ^-piece of DIR(0).

The format of the DIR("L") array is similar to DIR("A") and DIR("?"). The #'s must be numeric starting from 1. The numeric subscripted array nodes are written first and the DIR("L") node is written last. For example, if you code:

```
S DIR(0)="SO^1:ONE;2:TWO;3:THREE;4:FOUR;5:FIVE"
S DIR("L",1)="Select one of the following:"
S DIR("L",2)=""
S DIR("L",3)="  1  ONE      4  FOUR"
S DIR("L",4)="  2  TWO      5  FIVE"
S DIR("L")="  3  THREE"
D ^DIR
```

the user sees the following:

```
Select one of the following:

1  ONE      4  FOUR
2  TWO      5  FIVE
3  THREE

Enter response:
```

DIR("S") (Optional) Use the DIR("S") variable to screen the allowable responses for pointer, set of codes, and list/range reads. This variable works as the DIC("S") variable does for ^DIC calls. Set DIR("S") equal to M code containing an IF statement. After execution, if $T is set to 1, the user response is accepted; if set to 0, it is not.

For pointer reads, when DIR("S") is executed, the M naked indicator is equal to the 0 node of the entry being screened. The variable Y equals its record number.

For set of codes reads, when the DIR("S") is executed, Y equals the internal code.

For list/range reads, if you also use the C flag in piece 1 of DIR(0), your output is still compressed. Internally during the call, however, the range must be uncompressed so that each number in the range can be screened. So using DIR("S") with the C flag during list/range reads loses the C flag's advantages in speed (but the C flag's advantage in avoiding storage overflows remains).

DIR("T") (Optional) Time-out value to be used in place of DTIME. Value is represented in seconds.

DIR("?") (Optional) This variable contains a simple help prompt, which is displayed to the user when one question mark is entered. It usually takes the place of the reader's default prompt. For example, if you code:

```
S DIR(0)="F^3:10"
S DIR("?")="Enter from three to ten characters"
S DIR("A")="NICKNAME"
D ^DIR
```

the user sees the following:

```
NICKNAME: ?

 Enter from three to ten characters.
```

NOTE: When displayed, a period (.) is added to the DIR("?") string. Periods are not appended when displaying the DIR("?",#) array, however.

When one question mark is entered in DD reads, the data dictionary's help prompt is shown before DIR("?"). For pointer reads, a list of choices from the pointed-to file is shown in addition to DIR("?").

As an alternative, you can set DIR("?") to an up-arrow followed by M code, which is executed when the user enters one question mark. An example might be:

```
S DIR("?")="^D HELP^%DTC"
```

Execution of this M code overrides the reader's default prompt. If DIR("?") is defined in this way (a non-null second piece), the DIR("?",#) array is not displayed.

DIR("?",#) (Optional) This array allows the user to display more than one line of help when the user types a single question mark. The first up-arrow piece of DIR("?") must be set for the array to be used. The second up-arrow piece of DIR("?") must be null, otherwise the DIR("?",#) array is ignored. The #'s must be numeric starting from 1. The numbered lines are written first, that is, first DIR("?",1), then DIR("?",2), etc. The last help line written is DIR("?"). These lines are the only ones written, which means that the reader's default prompt is not issued.

DIR("??") (Optional) This variable, if defined, is a two-part variable. The first up-arrow piece may contain the name of a help frame. The help processor displays this help frame if the user enters two question marks.

The second part of this variable (after the first up-arrow piece) may contain M code that is executed after the help frame is displayed.

For example:

```
S DIR("??")="DIHELPXX^D EN^XXX"
```

NOTE: In order to use this variable, you must have Kernel's help processor on your system.

D. Output Variables (Full Listing)

X This is the unprocessed response entered by the user. It is always returned. If the user accepts the default in DIR("B"), it is the default. If the user up-arrows out or just presses the Enter/Return key on an optional input, X is the up-arrow or null.

Y Y is always defined as the processed output. The values returned are:

Type	Y Returned as
Date	The date/time in VA FileMan format.
End-of-page	Y=1 for continue (user pressed the Enter/Return key). Y=0 for exit (the user pressed up-arrow). Y="" for time out (the user timed out).
Free-text	The data typed in by the user. In this case, it is the same as X.
List or range	The list of numeric values, delimited by commas and ending with a comma. If the C flag was not included in the first piece of DIR(0), an expanded list of numbers, including each individual number in a range, is returned. If the C flag was included, a compressed list that uses the hyphen syntax to indicate a range of numbers is returned. Any leading zeros or trailing zeros following the decimal point are removed; i.e., only canonic numbers are returned. If the list of returned numbers has more than 245

	characters, integer-subscripted elements of Y [Y(1), Y(2), etc.] contain the additional numbers. Y(0) is always returned equal to Y.
Numeric	The canonic value of the number entered by the user; i.e., leading zeros are deleted and trailing zeros after the decimal are deleted.
Pointer	The normal value of Y from a DIC lookup, that is, Internal Entry Number^Entry Name. If the lookup was unsuccessful, Y=-1.
Set of Codes	The internal value of the response.
Yes/No	Y=1 for yes. Y=0 for no
DD (#,#)	The first ^-piece of Y contains the result of the variable X after it has been passed through the INPUT transform of the field specified. Depending on the data type involved, subsequent ^-pieces may contain additional information.

The following list summarizes the values of Y upon timeout, up-arrows, or pressing the Enter/Return keys for all reads. Exceptions are noted.

Condition	Value of Y	Comments
Timeout	Y=""	--
Up-arrow (^)	Y=^	in all cases except end-of-page reads.
Y=0	upon end-of-page reads.	--
Double Up-arrow (^^)	Y=^^	--

Return	Y=""	for optional reads (reads allowing a null response).
	Y=-1	for pointer reads.
	Y=0	for YES/NO type when NO is default.
	Y=1	for YES/NO type when YES is default.
	Y=1	for end-of-page reads.
	Y=default	when a default is provided other than for YES/NO type questions.

Y(0) This is defined for the set of codes, list, pointer, date, and Yes/No reads. It is also returned for DD reads when the field has a set of codes, pointer, variable pointer, or date data type. It holds the external value of the response for set of codes or Yes/No, the zero node of the entry selected for a pointer, and the external date for a date and variable pointer. To have Y(0) returned for pointer-types, the DIC(0) string in the second piece of DIR(0) must contain a Z, for example:

```
DIR(0)="P^19:EMZ"
```

For list reads, it contains the same values as the Y variable. There may be additional nodes in the Y() array depending on the size of the list selected by the user.

DTOUT If the read has timed-out, then DTOUT is defined.

DUOUT If the user entered a leading up-arrow, DUOUT is defined.

DIRUT If the user enters a leading up-arrow, times out, or enters a null response, DIRUT is defined. A null response results from pressing the Enter/Return key at a prompt with no default or

entering the at-sign (@), signifying deletion. If, however, the user presses the Enter/Return key in response to an end of page read, DIRUT is not defined. If DIRUT is defined, the user can enter the following common check to quit after a reader call:

```
Q:$D(DIRUT)
```

DIROUT If the user entered two up-arrows, DIROUT is defined.

E. Examples

1. **Date**
2. **End-of-Page**
3. **Free Text**
4. **List or Range**
5. **Numeric**
6. **Pointer**
7. **Set**
8. **Yes/No**
9. **DD**

1. Date Example

```
S DIR(0)="D^2880101:2880331:EX"
```

This tells the reader that the input must be an acceptable date. To determine that, ^%DT is invoked with the %DT variable equal to EX. If the date is a legitimate date, then it is checked to see if the date falls between January 1, 1988 and March 31, 1988. In general, both minimum and maximum are optional. If they are there, they must be in VA FileMan format. The only exceptions are that NOW and DT may be used to reference the current date/time. Remember that NOW contains a time stamp. If it is used as a minimum or maximum value, an R or T should be put into the %DT variable. If DIR(0) is set up to expect a time in the response, you can help the user by including that requirement in the prompt. Otherwise, a response without a time stamp (such as TODAY) might unexpectedly fail.

2. End-of-Page Example

```
S DIR(0)="E"
```

There are no parameters. Enter/Return and up-arrow are the only acceptable responses. This DIR(0) setting causes the following prompt to be issued:

```
Press the return key to continue or '^' to exit:
```

3. Free-Text Example

```
S DIR(0)="F^3:30"
```

This tells the reader that the input must be alphanumeric or punctuation, (control characters are not allowed) and that the length of input must be no fewer than 3 and no more than 30 characters. The maximum acceptable length for a free-text field is 245 characters.
NOTE: A leading up-arrow always aborts the read and sets DIRUT or DUOUT.

With DIR(0) containing U

```
S DIR(0)="FU^3:30"
```

The user can enter any response that is from 3 to 30 characters long. The response can contain embedded up-arrows. Without U, an embedded up-arrow causes the user to receive an error message.

With DIR(0) containing A

```
S DIR(0)="FA^2:5",DIR("A")="INITIAL"
```

The prompt is set only to the word INITIAL. If the A were not included, a colon and space would be appended to the prompt and it would look like this:

```
INITIAL:
```

4. List or Range Example

```
S DIR(0)="L^1:25"
```

This tells the reader that the input may be any set of numbers between 1 and 25. The numbers may be separated by commas, dashes, or a combination of both. Two acceptable responses to the example above are:

```
1,2,20
4-8,16,22-25
```

Remember that this is a numeric range or list. It can only contain positive integers and zero (no negative numbers).

With DIR(0) containing C

```
>S DIR(0)="LC^1:100:2" D ^DIR

Enter a list or range of numbers (1-100): 5,8.01,9-40,
7.03,45.9,80-100

>ZW Y
Y=5,7.03,8.01,9-40,45.9,80-100,
Y(0)=5,7.03,8.01,9-40,45.9,80-100,
```

Here the user can enter numbers from 1 to 100 with up to two decimal places. The C flag tells the reader not to return each individual number in Y. Instead, inclusive ranges of numbers are returned. In this case, without the C flag, 137 subscripted nodes of the Y( ) array would be returned; the call would be very slow and might cause an error if the size of the Y( ) array exceeded local storage.

5. Numeric Example

```
S DIR(0)="N^20:30:3"
```

This tells the reader that the input must be a number between 20 and 30 with no more than three decimal digits.
NOTE: If no maximum is specified in the second ^-piece, the default maximum is 999999999999.

With DIR(0) containing O

```
S DIR(0)="NO^0:120",DIR("A")="AGE"
```

This allows the user to press the Enter/Return key without entering any response and leave the reader. Without the O, the following messages appear:

```
This is a required response.  Enter '^' to exit.
```

6. Pointer Example

```
S DIR(0)="P^19:EMZ"
```

This tells the reader to do a lookup on File 19, setting DIC(0)="EMZ" before making the call.

If the user enters a response that causes the lookup to fail, the user is prompted again for a lookup value.

A pointer read can be used to look up in a subfile. In that case, the global root must be used in place of the file number. For example, to look up in the menu subfile (stored descendent from subscript 10) for entry #2 in File 19:

```
S DIR(0)="P^DIC(19,2,10,:QEM"
```

Remember to set any necessary variables, e.g., DA(1).

7. Set Example

```
S DIR(0)="S^1:MARRIED;2:SINGLE"
```

This tells the reader to only accept one of the two members of the set. The response may be 1, 2, MARRIED, or SINGLE. When DIR("A") is included without the A modifier on the first piece, the prompting is done as follows:

```
S DIR(0)="S^M:MALE;F:FEMALE"
S DIR("A")="SEX" D ^DIR

Select one of the following:
       M      MALE
       F       FEMALE
SEX:
```

With DIR(0) containing A

```
S DIR(0)="SA^M:MALE;F:FEMALE"
S DIR("A")="SEX: " D ^DIR
```

Whereas, with the A, it would appear as follows:

```
SEX:
```

With DIR(0) containing B

```
S DIR(0)="SB^M:MALE;F:FEMALE"
S DIR("A")="SEX" D ^DIR
```

When this is executed, instead of getting the vertical listing as shown above, the prompt would appear as:

```
SEX: (M/F):
```

With DIR(0) containing X

```
S DIR(0)="SX^M:MALE;F:FEMALE"
S DIR("A")="SEX"
```

This would cause a lowercase M or F to be rejected. The prompting is done as follows:

```
Select one of the following:
    M   Male
    F   Female
SEX:  f  (user's response)
Enter a code from the list.
```

8. Yes/No Example

```
S DIR(0)="Y",DIR("B")="YES"
```

This tells the reader that the response can only be Yes or No. When using DIR("B") to provide a default response, spell out the entire word so that when the user presses the Enter/Return key to accept the default, echoing functions properly.

9. DD Example

```
S DIR(0)="19,1"
```

This format is different from the others in that the first number is a file number and the second is a field number in that file. The reader uses the data dictionary for

field 1 in file 19 and issues the label of that field as the prompt. The input is passed through the INPUT transform in the dictionary. Help messages are also the ones contained in the dictionary for this field.

Normally, DD reads based on a free text field do not allow embedded up-arrows. However, if the field specified is positioned on the data node using the Em,n format (instead of the ^-piece format), up-arrows embedded in the user's response are accepted. (See the Field Global Storage section of the Advanced File Definition chapter for an explanation of locating fields on the data node.) Initial up-arrows abort the read and set DIRUT and DUOUT.

It is not possible to use this format if the field defines a subfile, i.e., the second piece of the zero node of the field definition contains a subfile number. To use the reader for a field in a subfile, do the following:

```
S DIR(0)="Subfile#,field#"
```

It is the programmer's responsibility to set any variables necessary for the INPUT transform to execute correctly.

Always NEW or KILL DA before doing a DD-type DIR call, unless you wish to use the default feature. The default feature allows you to retrieve default values from the database for DD reads by setting DA (or the DA array for subfiles) equal to the record number containing the desired default value.

EN^DIS: Search File Entries

You can call the Search File Entries option of VA FileMan for a given file when you want the user to be able to specify the search criteria. This is done by invoking EN^DIS. In addition to DT and DUZ, the program needs the DIC input variable.

Input Variable

DIC (Required) The global root of the file in the form ^GLOBAL(or ^GLOBAL(#, or the number of the file.

If the search is allowed to run to completion, and if the search criteria have been stored in a template, then a list of the record numbers that meet the search criteria is stored in that same template.

NOTE: The same global array is used to store a list of record numbers saved in FileMan Inquire mode.

```
^DIBT(SORT_TEMPLATE#,1,IEN)=""
```

The 1 node indicates that the IEN list was created one of two ways:

1. The user was in FileMan INQUIRE mode, selected a number of records, and saved the list in a template.

2. The user ran the FileMan SEARCH, either through the interactive FileMan menu or through the programmer entry point EN^DIS. In this case, the IEN list is the group of record numbers that met the search criteria.

IEN is the internal entry number of a record in the file indicated by the fourth piece of the zero node of the template, ^DIBT(SORT_TEMPLATE#,0).

The list of record numbers stored in the template can be used as input to the print routine, EN1^DIP, to create further reports.

EN^DIU2: Data Dictionary Deletion

Occasionally you may need to delete a file's data dictionary and its entry in ^DIC in order to properly update a running system. Use this entry point to do it.

You usually have the option of deleting the data when you delete the data dictionary. (See the DIU(0) variable below.) However, data will always be deleted if your file is in ^DIC(File#,. Be careful using this utility when your data is in the ^DIC global.

In all cases, both DIU and DIU(0) are returned from the call. You will find that DIU is returned as the global root regardless of whether it was defined as the file number or as the global root when making the call.

NOTE: If the root of a file's data is an unsubscripted global [e.g., DIU="^MYDATA("], you must make sure that the systems on which you want to perform the deletion do not restrict the killing of the affected unsubscripted globals.

REMINDER: It is your responsibility to clean up (kill) DIU, the input variable, after any call to this routine!

Input Variables

DIU (Required) The file number or global root, e.g., ^DIZ(16000.1,. This must be a subfile number when deleting a subfile's data dictionary.

DIU(0) Input parameter string that may contain the following:

- **D** **D**elete the data as well as the data dictionary.
- **E** **E**cho back information during deletion.
- **S** **S**ubfile data dictionary is to be deleted.
- **T** **T**emplates are to be deleted.

Example

```
>S DIU="^DIZ(16000.1,",DIU(0)="" D EN^DIU2
```

Only the data dictionary will be deleted. The data and templates remain. By including either the D or T, you can also delete the data or the templates. If the E is included, then the user will be asked whether or not the global should be deleted.

Subfile Deletion

If you want to delete the dictionary for a subfile, you must include the S in DIU(0). The variable, DIU, in this case must be a subfile data dictionary number. It cannot be a global root. When deleting a subfile's dictionary, all dictionaries subordinate to that dictionary are also deleted. Data can also be deleted when deleting a subfile; this process could take some time depending on the number of entries in the whole file.

Example

```
>S DIU=16000.01,DIU(0)="S" D EN^DIU2
```

EN^DIWE: Text Editing

This routine is used to edit word processing text using VA FileMan's editors. If the user has established a Preferred Editor through Kernel, that editor is presented for use. FileMan's editors expect the text to contain only printable ASCII characters.

Input Variables

DIC The global root of where the text is located.

NOTE: VA FileMan uses ^UTILITY($J,"W") when EN^DIWE is called. Thus, DIC should not be set equal to that global location.

DWLW (Optional) This variable indicates the maximum number of characters that will be stored on a word-processing global node. When the user enters text, the input line will not be broken to DWLW-characters until after the Enter/Return key is pressed. Thus, if DWLW=40 and the user types 90 characters before pressing the Enter/Return key, the text would be stored in three lines in the global. If this variable is not set, the default value is 245. This variable is always killed by FileMan.

DWPK (Optional) This variable determines how lines that are shorter than the maximum line length (set by DWLW) are treated by FileMan. It can be set to 1 or 2. This variable is always killed by FileMan.

DWPK=1 If the user enters lines shorter than the maximum line length in variable DWLW, the lines will be stored as is; they will not be joined. If lines longer than DWLW are entered, the lines will be broken at word boundaries.

DWPK=2 If the user types lines shorter than the maximum line length in variable DWLW, the lines will be joined until they get to the maximum length; the lines are "filled" to DWLW in length. If the lines are longer than DWLW, they will be broken at word boundaries. This is the default used if DWPK is not set prior to the EN^DIWE call.

DWDISABL This variable can be used to disable specific Line Editor commands. For example, if DWDISABL contains "P", then the Print command in the Line Editor is disabled. This variable is killed by FileMan. (Optional)

DIWEPSE (Optional) If this variable is defined before entering the Preferred Editor (if the Preferred Editor is not the Line Editor), the user receives the following prompt:

```
Press RETURN to continue or '^' to exit:
```

Set this variable if you want to allow the user to read information on the screen before the display is cleared by a screen-oriented editor. This variable is always killed by FileMan.

DIWESUB (Optional) The first 30 characters of this variable are displayed within angle brackets (< and >) on the top border of the Screen Editor screen. This variable is killed by FileMan.

DIWETXT (Optional) The first IOM characters of this variable are displayed in high intensity on the first line of the Screen Editor screen. This variable is killed by FileMan.

DDWLMAR (Optional) This variable indicates the initial column position of the left margin when the Screen Editor is invoked. The user can subsequently change the location of the left margin. This variable is killed by FileMan.

DDWRMAR (Optional) This variable indicates the initial column position of the right margin when the Screen Editor is invoked. The user can subsequently change the location of the right margin. This variable is killed by FileMan.

DDWRW (Optional) This variable indicates to the Screen Editor the line in the document on which the cursor should initially rest. This variable has effect only if the user's preferred editor is the Screen Editor and applies only when the Screen Editor is first invoked. If the user switches from the Screen Editor to another editor and then back to the Screen Editor, the cursor always rests initially on line 1.

If this variable is set to "B", the cursor will initially rest at the bottom of the document and the value of DDWC described

immediately below is ignored. The default value of DDWRW is 1. This variable is killed by FileMan.

DDWC (Optional) This variable indicates to the Screen Editor the initial column position of the cursor. The same restrictions described above for DDWRW apply to DDWC.

If this variable is set to "E", the cursor will initially rest at the end of the line defined by DDWRW. The default value of DDWC is 1. This variable is killed by FileMan.

DDWFLAGS Flags to control the behavior of the Screen Editor. The possible values are:

M Indicates that the Screen Editor should initially be in NO WRAP **M**ode when invoked.

Q Indicates that if the user attempts to **Q**uit the editor with <PF1>Q, the confirmation message "Do you want to save changes?" is NOT asked.

R Indicates that the Screen Editor should initially be in **R**EPLACE mode when invoked.

This variable is killed by FileMan. (Optional)

^DIWF: Form Document Print

Form Document Print Introduction (^DIWF)

The entry points in ^DIWF are designed to use the contents of a word processing field as a target document into which data can be inserted at print time. The data may come from another VA FileMan file or be provided by the user interactively at the time the document is printed. A file containing a word processing type field is first selected and then an entry from that file. The word processing text in that entry is then used as a form with which to print output from any other file.

The word processing text used will typically include windows into which data from the target file automatically gets inserted by DIWF. The window delimiter is the vertical bar (|). Thus, if a word processing document contains |NAME| somewhere within it, DIWF will try to pick the NAME field (if there is one) out of the file being printed. Any non-multiple field label or computed expression can be used within a |-window, if:

1. an expression within the |-window cannot be evaluated, and
2. the output of DIWF is being sent to a different terminal than the one used to call up the output,

then the user will be asked to type in a value for the window, for each data entry printed. Thus, the word processing text used as a target document might include the window |SALUTATION|, where SALUTATION is not a valid field name in the source file. When DIWF encounters this window, and failing to find a SALUTATION field in the source file, it will ask the user to enter SALUTATION text which then immediately gets incorporated into the output in place of that window. Notice that we are referring to two files-the document file which contains the word processing text and the print from file which DIWF will use to try to fill-in data for the windows.

Invoking DIWF at the top (i.e., D ^DIWF) results in an interactive dialog with the user.

Example

Suppose you had a file called FORM LETTER (File #16001) and data is stored in ^DIZ(16001,. This file has a word processing type field where the text of a form letter is stored. In this file, as shown below, there are several form letter entries one of which is APPOINTMENT REMINDER:

```
Select Document File: FORM LETTER
Select DOCUMENT: APPOINTMENT REMINDER
Print from what FILE: EMPLOYEE
WANT EACH ENTRY ON A SEPARATE PAGE? YES// <RET>
SORT BY: NAME// FOLLOWUP DATE=MAY 1, 1999
DEVICE:
```

In this example, the word processing text found in the APPOINTMENT REMINDER entry of the FORM LETTER file is used to print a sheet of output for each EMPLOYEE file entry whose FOLLOWUP DATE equals May 1,1999.

If the document file contains a pointer field pointing to File #1, and if the document entry selected has a value for that pointer, then the file pointed to will be automatically used to print from and the user will not be asked "Print from what FILE:".

NOTE: The Read access is checked by DIWF for both files selected.

EN1^DIWF: Form Document Print

Form Document Print Introduction (^DIWF)

The entry points in ^DIWF are designed to use the contents of a word processing field as a target document into which data can be inserted at print time. The data may come from another VA FileMan file or be provided by the user interactively at the time the document is printed. A file containing a word processing type field is first selected and then an entry from that file. The word processing text in that entry is then used as a form with which to print output from any other file.

The word processing text used will typically include windows into which data from the target file automatically gets inserted by DIWF. The window delimiter is the vertical bar (|). Thus, if a word processing document contains |NAME| somewhere within it, DIWF will try to pick the NAME field (if there is one) out of the file being printed. Any non-multiple field label or computed expression can be used within a |-window, if:

1. an expression within the |-window cannot be evaluated, and
2. the output of DIWF is being sent to a different terminal than the one used to call up the output,

then the user will be asked to type in a value for the window, for each data entry printed. Thus, the word processing text used as a target document might include the window |SALUTATION|, where SALUTATION is not a valid field name in the source file. When DIWF encounters this window, and failing to find a SALUTATION field in the source file, it will ask the user to enter SALUTATION text which then immediately gets incorporated into the output in place of that window. Notice that we are referring to two files-the document file which contains the word processing text and the print from file which DIWF will use to try to fill-in data for the windows.

This entry point is used when the calling program knows which file (document file) contains the desired word processing text to be used as a target document.

Input Variable

DIC A file number or a global root. The file identified must contain a word processing field.

Output Variable

Y This will be -1 only if the file sent to DIWF in the variable DIC does not contain a word processing field.

Example

```
>S DIC=16001 D EN1^DIWF
```

The user will then be branched to the "Select DOCUMENT:" prompt in the dialog described above to select a particular entry in the Form Letter file.

EN2^DIWF: Form Document Print

Form Document Print Introduction (^DIWF)

The entry points in ^DIWF are designed to use the contents of a word processing field as a target document into which data can be inserted at print time. The data may come from another VA FileMan file or be provided by the user interactively at the time the document is printed. A file containing a word processing type field is first selected and then an entry from that file. The word processing text in that entry is then used as a form with which to print output from any other file.

The word processing text used will typically include windows into which data from the target file automatically gets inserted by DIWF. The window delimiter is the vertical bar (|). Thus, if a word processing document contains |NAME| somewhere within it, DIWF will try to pick the NAME field (if there is one) out of the file being printed. Any non-multiple field label or computed expression can be used within a |-window, if:

1. an expression within the |-window cannot be evaluated, and
2. the output of DIWF is being sent to a different terminal than the one used to call up the output,

then the user will be asked to type in a value for the window, for each data entry printed. Thus, the word processing text used as a target document might include the window |SALUTATION|, where SALUTATION is not a valid field name in the source file. When DIWF encounters this window, and failing to find a SALUTATION field in the source file, it will ask the user to enter SALUTATION text which then immediately gets incorporated into the output in place of that window. Notice that we are referring to two files-the document file which contains the word processing text and the print from file which DIWF will use to try to fill-in data for the windows.

This entry point is used when the calling program knows both the document file and the entry within that file which contains the desired word processing text to be used as a target document.

Input Variables

DIWF The global root at which the desired text is stored. Thus, in our example, if APPOINTMENT REMINDER is the third document in

the Form Letter file (stored in ^DIZ(16001,) and the word

processing field is stored in subscript 1, you can:

```
S DIWF="^DIZ(16001,3,1,"
```

DIWF will then automatically use this entry and the user will not be asked to select the document file and which document in that file.

DIWF(1) If the calling program wants to specify which file should be used as a source for generating output, the number of that file should appear in the variable DIWF(1). Otherwise, the user will be asked the "Print from what FILE:" question.

After this point, EN1^DIP is invoked. You can have the calling program set the usual BY, FR, and TO variables if you want to control the SORT sequence of the data file.

Output Variable

Y Y will be -1 if:

- There is no data beneath the root passed in DIWF.
- The file passed in DIWF(1) could not be found.

^DIWP: Formatter

Call ^DIWP to format and (optionally) output any group of text lines.

Before calling ^DIWP, you should kill the global ^UTILITY($J,"W").

^DIWP works in **two modes** (based on whether the DIWF input parameter contains "W" or not):

1. In ^DIWP's "**accumulate**" **mode**, repeated calls to ^DIWP accumulate and format text in ^UTILITY($J,"W"). After you have finished accumulating text, if you want to write the text to the current device, you should call ^DIWW. ^DIWW writes the accumulated text to the current device with the margins you specified in your calls to ^DIWP and then it removes the text from ^UTILITY.

2. In ^DIWP's "**write**" **mode**, if the text added to ^UTILITY($J,"W") by ^DIWP causes one or more (that is, n) line breaks, n lines are written to the current device(and the remaining partial line is stored in ^UTILITY. This leaves one line of text in ^UTILITY once all calls to ^DIWP are completed. To write the remaining line of text to the current device and remove it from ^UTILITY, call ^DIWW.

Input Variables

X The string of text to be added as input to the formatter.

The X input string may contain |-windows, as described in the Formatting Text with Word Processing Windows topic in the Advanced Edit Techniques chapter of the *VA FileMan Advanced User Manual* (e.g., |SETTAB(9,23,44)|). The expressions within the windows will be processed as long as they are not context-dependent; that is, as long as they do not refer symbolically to database field names. Thus, |TODAY| will cause today's date to be inserted into the formatted text, but |SSN| will be printed out as it stands, because it cannot be interpreted in context.

DIWL The (integer-valued) left margin for the text. Set this to a postive number, 1 or greater. Do not change the value of DIWL if you are making repeated calls to ^DIWP to format text.

DIWR The (integer-valued) right margin for the text.

DIWF A string of format control parameters. If contained in DIWF, the parameters have the following effects:

Cn The text will be formatted in a **C**olumn width of **n**, thus overriding the value of DIWR.

D The text will be in **D**ouble-spaced format.

In The text will be **In**dented n columns in from the left margin (DIWL).

N Each line will be printed as it appears in the text (**N**o-wrap). If DIWF contains N, the value of DIWR will be ignored. See the Advanced Edit Techniques chapter in the *VA FileMan Advanced User Manual* for details about word wrapping.

R The text will be in **R**ight-justified format.

W If the DIWF parameter contains "W", ^DIWP operates in "**W**rite" mode If the DIWF parameter does not contain "W", ^DIWP operates in "accumulate" mode. See above for the discussion of these two modes.

When making repeated calls to ^DIWP, don't mix modes. Use "write" or "accumulate" mode, but don't switch between them.

| Word processing windows (material within **vertical bars**) will not be evaluated. The window will print as it exists in the word processing field.

^DIWW: WP Print

Use ^DIWW to output to the current device the remaining text left in ^UTILITY($J,"W") by ^DIWP.

The ^DIWW entry point is designed to be used in conjunction with the ^DIWP entry point. Using ^DIWP, you can accumulate and format text in ^UTILITY($J,"W"), in one of **two modes**:

3. In ^DIWP's "**accumulate**" **mode**, repeated calls to ^DIWP accumulate and format text in ^UTILITY($J,"W"). When you have finished accumulating text, you should call ^DIWW to write the text to the current device. ^DIWW writes the accumulated text to the current device with the margins you specified in your calls to ^DIWP and then removes the text from ^UTILITY.

4. In ^DIWP's "**write**" **mode**, if the text added to ^UTILITY($J,"W") by ^DIWP causes one or more (that is, n) line breaks, n lines are written to the current device (and the remaining partial line is stored in ^UTILITY.) This leaves one line of text in ^UTILITY once all calls to ^DIWP are completed. To write the remaining line of text to the current device and remove it from ^UTILITY, call ^DIWW.

%DT: Introduction to Date/Time Formats

This introduction pertains to all of the %DT calls which follow. Please read this first because it is relevant to all of the %DT calls.

%DT is used to validate date/time input and convert it to VA FileMan's conventional internal format: "YYYMMDD.HHMMSS", where:

- **YYY** is number of years since 1700 (hence always 3 digits)
- **MM** is month number (00-12)
- **DD** is day number (00-31)
- **HH** is hour number (00-23)
- **MM** is minute number (01-59)
- **SS** is the seconds number (01-59)

This format allows for representation of imprecise dates like JULY '78 or 1978 (which would be equivalent to 2780700 and 2780000, respectively). Dates are always returned as a canonic number (no trailing zeroes after the decimal).

^%DT: Internal to External Date

Introduction to Date/Time Formats: %DT

%DT is used to validate date/time input and convert it to VA FileMan's conventional internal format: "YYYMMDD.HHMMSS", where:

YYY is number of years since 1700 (hence always 3 digits)
MM is month number (00-12)
DD is day number (00-31)
HH is hour number (00-23)
MM is minute number (01-59)
SS is the seconds number (01-59)

This format allows for representation of imprecise dates like JULY '78 or 1978 (which would be equivalent to 2780700 and 2780000, respectively). Dates are always returned as a canonic number (no trailing zeroes after the decimal).

This routine accepts input and validates the input as being a correct date and time.

Input Variables

%DT A string of alphabetic characters which alter how %DT responds. Briefly stated, the acceptable characters are:

A	**A**sk for date input.
E	**E**cho the answer.
F	**F**uture dates are assumed.
I	For **I**nternationalization, assume day number precedes month number in input.
M	Only **M**onth and year input is allowed.
N	Pure **N**umeric input is not allowed.
P	**P**ast dates are assumed.
R	**R**equires time input.
S	**S**econds should be returned.

T **T**ime input is allowed but not required.

X E**X**act input is required.

For an explanation of each character, see %DT Input Variables in Detail below.

X If %DT does not contain an A, then the variable X must be defined as equal to the value to be processed. See Date Fields in the Editing Specific Field Types chapter of the *VA FileMan Getting Started Manual* for acceptable values for X and for the interpretation of those values.

%DT("A") (Optional) A prompt which will be displayed prior to the reading of the input. Without this variable, the prompt "DATE:" will be issued.

%DT("B") The default answer to the "DATE:" prompt. It is your responsibility to ensure that %DT("B") contains a valid date/time. Allowable date input formats are explained in the Editing Specific Field Types chapter of the *VA FileMan Getting Started Manual*.

%DT(0) (Optional) Prevents the input date value from being accepted if it is chronologically before or after a particular date. Set %DT(0) equal to a VA FileMan-format date (e.g., %DT(0)=2690720) to allow input only of dates greater than or equal to that date. Set it negative (e.g., %DT(0)=-2831109.15) to allow only dates less than or equal to that date/time. Set it to NOW to allow dates from the current (input) time forward. Set it to -NOW to allow dates up to the current time.

NOTE: Be sure to kill this variable after returning from %DT.

Output Variables

Y %DT always returns the variable Y, which can be one of two values:

Y=-1	The date/time was invalid.
Y=YYYMMDD.HHMMSS	The value determined by %DT.

X X is always returned. It contains either what was passed to %DT (in the case where %DT did not contain an A) or what the user entered.

DTOUT This is only defined if %DT has timed-out waiting for input from the user.

%DT Input Variables in Detail

A %DT **A**sks for input from the terminal. It continues to ask until it receives correct input, a null, or an up-arrow. If %DT does not contain the character A, the input to %DT is assumed to be in the variable X.

E The **E**xternal format of the input will be echoed back to the user after it has been entered. If the input was erroneous, two question marks and a "beep" will be issued.

F If a year is not entered (example 1), or if a two-digit year is entered (example 2), a date in the **F**uture is assumed.

EXCEPTION: If a two-digit year is entered and those two digits equal the current year, the current year is assumed even if the date is in the past (example 3).

Example	Current Date	User Input	Date Returned	Returned Without F
1)	July 1, 2000	5/1	May 1, 2001	May 1, 2000
2)	July 1, 2000	5/1/90	May 1, 2090	May 1, 1990
3)	July 1, 2000	5/1/00	May 1, 2000	May 1, 2000

See Y2K Changes below for the behavior of %DT when neither the F nor P flag is used.

I For **I**nternalization, this flag makes %DT assume that in the input, the day number precedes the month number. For example, input of 05/11/2000 is assumed to be November 5, 2000 (instead of May 11, 2000). Also, with this flag, the month must be input as a number.

For example, November must be input as 11, not NOV.

M Only **M**onth and year input is allowed. Input with a specific day or time is rejected (example 1). If only a month and two digits are entered, the two digits are interpreted as a year instead of a day (example 2).

If the M flag is used with the X flag, a month must be specified; otherwise, the input can be just a year (example 3).

M Flag

Example	Date Input	Date Returned	Returned Without M
1)	7-05-2005	invalid	July 5, 2005
2)	7-05	July 2005	July 5, 2000*

*Assuming the current year is 2000 and the F and P flags aren't used.

M Flag (with X Flag)

Example	Date Input	Date Returned	Returned Without X
3)	05 or 2005	invalid	2005

N Ordinarily, a user can enter a date in a purely **N**umeric form, i.e., MMDDYY. However, if %DT contains an N, then this type of input is not allowed.

P If a year is not entered (example 1), or if a two-digit year is entered (example 2), a date in the **P**ast is assumed.

EXCEPTION: If a two-digit year is entered and those two digits equal the current year, the current year is assumed even if the date is in the future (example 3).

Ex.	Current Date	User Input	Date Returned	Returned Without P
1)	March 1, 1995	6/1	June 1, 1994	June 1, 1995
2)	March 1, 1995	6/1/98	June 1, 1898	June 1, 1998
3)	March 1, 1995	6/1/95	June 1, 1995	June 1, 1995

See Y2K Changes below for the behavior of %DT when neither the F nor P flag is used.

R Time is **R**equired. It must be input.

S **S**econds are to be returned.

T **T**ime is allowed in the input, but it is not necessary. See Date Fields in the Editing Specific Field Types chapter of the *VA FileMan Getting Started Manual* for details of how user-input times are interpreted.

X E**X**act input is required. If X is used without M, date input must include a day and month. Without X, the input can be just month-year or only a year.

If X is used with M, date input must include a month. If M is used without X, then the input can be just a year.

Y2K Changes:

If no year is entered, the current year is assumed (example 1).

If a two-digit year is entered, a year less than 20 years in the future and no more than 80 years in the past is assumed. For example, in the year 2000, two-digit years are assumed to be between 1920 through 2019.

NOTE: Only the year, not the current month and day, is taken into account in this calculation (examples 2 through 5).

Example	Current Date	User Input	Date Returned
1)	Sep 15, 2000	3/15	Mar 15, 2000
2)	Sep 15, 2000	1/1/20	Jan 01, 1920
3)	Sep 15, 2000	12/31/20	Dec 31, 1920
4)	Sep 15, 2000	1/1/19	Jan 01, 2019
5)	Sep 15, 2000	12/31/19	Dec 31, 2019

DD^%DT: Internal to External Date

Introduction to Date/Time Formats: %DT

%DT is used to validate date/time input and convert it to VA FileMan's conventional internal format: "YYYMMDD.HHMMSS", where:

YYY is number of years since 1700 (hence always 3 digits)
MM is month number (00-12)
DD is day number (00-31)
HH is hour number (00-23)
MM is minute number (01-59)
SS is the seconds number (01-59)

This format allows for representation of imprecise dates like JULY '78 or 1978 (which would be equivalent to 2780700 and 2780000, respectively). Dates are always returned as a canonic number (no trailing zeroes after the decimal).

There are two ways to convert a date from internal to external format—this call and X ^DD("DD"). (This is the reverse of what %DT does.) This entry point takes an internal date in the variable Y and converts it to its external representation.

Example 1

```
>S Y=2690720.163 D DD^%DT W Y
JUL 20, 1969@1630
```

This results in Y being equal to JUL 20, 1969@16:30. (Single space before the 4-digit year.)

Input Variables

Y	(Required) This contains the internal date to be converted. If this has five or six decimal places, seconds will automatically be returned.
%DT	(Optional) This forces seconds to be returned even if Y does not have that resolution. %DT must contain S for this to happen.

Output Variable

Y Y is returned as the external form of the date.

See also DT^DIO2, which takes an internal date in the variable Y and *writes out* its external form.

^%DTC: Date/Time Utility

^%DTC returns the number of days between two dates.

Input Variables

X1	(Required) One date in VA FileMan format. This is not returned.
X2	(Required) The other date in VA FileMan format. This is not returned.

Output Variables

X	The number of days between the two dates. X2 is subtracted from X1.
%Y	If %Y is equal to 1, the dates have both month and day values. If %Y is equal to 0, the dates were imprecise and therefore not workable.

C^%DTC: Date/Time Utility

C^%DTC takes a date and adds or subtracts a number of days, returning a VA FileMan date and a $H format date. If time is included with the input, it will also be included with the output.

Input Variables

X1	(Required) The date in VA FileMan format to which days are going to be added or from which days are going to be subtracted. This is not returned.
X2	(Required) If positive, the number of days to add. If negative, the number of days to subtract. This is not returned.

Output Variables

X	The resulting date, in VA FileMan format, after the operation has been performed.
%H	The $H form of the date.

COMMA^%DTC: Date/Time Utility

Formats a number to a string that will separate billions, millions, and thousands with commas.

Input Variables

X (Required) The number you want to format. X may be positive or negative.

X2 (Optional) The number of decimal digits you want the output to have. If X2 is not defined, two decimal digits are returned. If X2 is a number followed by the dollar sign (e.g., 3$) then a dollar sign will be prefixed to X before it is output.

X3 (Optional) The length of the desired output. If X3 is less than the formatted X, X3 will be ignored. If X3 is not defined, then a length of twelve is used.

Output Variable

X The initial value of X, formatted with commas, rounded to the number of decimal digits specified in X2. If X2 contained a dollar sign, then the dollar sign will be next to the leftmost digit. If X was negative, then the returned value of X will be in parentheses. If X was positive, a trailing space will be appended. If necessary, X will be padded with leading spaces so that the length of X will equal the value of the X3 input variable.

Examples

Example 1

```
>S X=12345.678 D COMMA^%DTC
```

The result is:

```
X="  12,345.68 "
```

Example 2

```
>S X=9876.54,X2="0$" D COMMA^%DTC
```

The result is:

```
X="     $9,877 "
```

Example 3

```
>S X=-3,X2="2$" D COMMA^%DTC
```

The result is:

```
X="     ($3.00)"
```

Example 4

```
>S X=12345.678,X3=10 D COMMA^%DTC
```

The result is:

```
X="12,345.68 "
```

DW^%DTC: Date/Time Utility

This entry point produces results similar to H^%DTC. The difference is that X is reset to the name of the day of the week—Sunday, Monday, and so on. If the date is imprecise, then X is returned equal to null.

H^%DTC: Date/Time Utility

H^%DTC converts a VA FileMan date/time to a $H format date/time.

Input Variable

X	(Required) The date/time in VA FileMan format. This is not returned.

Output Variables

%H	The same date in $H format. If the date is imprecise, then the first of the month or year is returned.
%T	The time in $H format, i.e., the number of seconds since midnight. If there is no time, then %T equals zero.
%Y	The day-of-week as a numeric from 0 to 6, where 0 is Sunday and 6 is Saturday. If the date is imprecise, then %Y is equal to -1.

HELP^%DTC: Date/Time Utility

This entry point displays a help prompt based on %DT and %DT(0).

Input Variables

%DT The format of %DT is described in the %DT section of this chapter. The help prompt will display different messages depending on the parameters in the variable.

%DT(0) (Optional) The format of %DT(0) is described in the %DT section of this chapter. This input variable causes HELP to display the upper or lower bound that is acceptable for this particular call.

NOW^%DTC: Date/Time Utility

NOW^%DTC returns the current date/time in VA FileMan and $H formats.

Output Variables

%	VA FileMan date/time down to the second.
%H	$H date/time.
%I(1)	The numeric value of the month.
%I(2)	The numeric value of the day.
%I(3)	The numeric value of the year.
X	VA FileMan date only.

S^%DTC: Date/Time Utility

This entry takes the number of seconds from midnight and turns it into hours, minutes, and seconds as a decimal part of a VA FileMan date.

Input Variable

% A number indicating the number of seconds from midnight, e.g., $P($H,",",2).

Output Variable

% The decimal part of a VA FileMan date.

Example

```
>SET %=44504 D S^%DTC W %
.122144
```

YMD^%DTC: Date/Time Utility

Converts a $H format date to a VA FileMan date.

Input Variable

%H	(Required) A $H format date/time. This is not returned.

Output Variables

%	Time down to the second in VA FileMan format, that is, as a decimal. If %H does not have time, then % equals zero.
X	The date in VA FileMan format.

YX^%DTC: Date/Time Utility

This entry point takes a $H date and passes back a printable date and time. It also passes back the VA FileMan form of the date and time.

Input Variable

%H	(Required) This contains the date and time in $H format which is to be converted. Time is optional. This is not returned.

Output Variables

Y	The date and time (if time has been sent) in external format. Seconds will be included if the input contained seconds.
X	The date in VA FileMan format.
%	The time as a decimal value in VA FileMan format. If time was not sent, then % will be returned as zero.

%XY^%RCR: Array Moving

This entry point can be used to move arrays from one location to another. The location can be local or global.

After the call has completed, both arrays are defined. They are identically subscripted if the %Y array did not previously exist. If the array identified in %Y had existing elements, those elements will still exist after the call to %XY^%RCR. However, their values may have to be examined because an identically subscripted element in the %X array will replace the one in the %Y array, but an element which existed in the %Y array (but not in the %X array) will remain as it was.

Input Variables

%X	The global or array root of an existing array. The descendents of %X will be moved.
%Y	The global or array root of the target array. It is best if this array does not exist before the call.

Example

To move the local array X(to ^TMP($J, you would write:

```
>S %X="X(" S %Y="^TMP($J," D %XY^%RCR
```

2. Database Server (DBS) API

INTRODUCTION

The VA FileMan Database Server (DBS) is an Application Programmer Interface (API) for accessing data attributes and data in VA FileMan files. The principal function of this API is to separate database access from user presentation. In Classic FileMan's roll and scroll mode, the interaction with the end user was closely tied to the code that actually changed the database. Whenever FileMan needed information from the user, a Read was done; whenever FileMan needed to present information to the user, a Write was done.

However, with FileMan's DBS calls, no Writes to the current device are done. Interaction with the user is managed by the client application. Package developers can manage user interaction from within their own code and can call FileMan whenever interaction with the database is needed. The DBS calls are used to update the database in a non-interactive mode. Information needed by the FileMan routines is passed through parameters rather than through interactive dialog with the user. Any information that needs to be displayed to the end user is passed by FileMan back to the calling routine in arrays.

This separation of data access from user I/O makes possible the construction of alternative front-ends to the VA FileMan database (for example, a windowed Graphical User Interface [GUI]). In addition, this API can be the basis for data access by applications running outside M.

The first section in this chapter (How to Use) describes the conventions used in the DBS API. The next section (How the DBS Communicates) offers a detailed description of the way DBS calls return information to the client application in arrays. Finally, the individual calls are described, including input parameters, output, and examples of their use.

HOW TO USE THE DBS CALLS

Format and Conventions of the Calls

All of the DBS calls use parameter passing instead of relying on variables set prior to the call that are passed through the symbol table. However, FileMan's key variables (e.g., DUZ and DT) are not passed in the parameter list. When needed, FileMan continues to expect them to be defined in the local symbol table.

Except where noted, the order of the parameters in the argument list follows a consistent pattern as follows:

```
TAG^ROUTINE(FILE,IENS,FIELD,FLAGS,OTHER_REQUIRED_PARAMS,
    OTHER_OPTIONAL_PARAMS)
```

If a particular call does not use one or more of the first four parameters, that parameter is omitted from the list of arguments. Generally, when a file is needed, the file number (not global root) must be passed. This allows for consistency when referring either to a top level file or to a subfile. Similarly, a field is identified by its field number.

When it is necessary to pass the root of a local or global array, the complete closed reference of the array for use with subscript indirection is needed, not the traditional open VA FileMan root. Examples are illustrated below:

Acceptable Roots	**Unacceptable Roots**
^TMP("NMSP",$J)	^TMP("NMSP",$J,
LOCALVAR	LOCALVAR(

Since the array identified by this root is accessed by indirection, the contents of the array may be changed by the VA FileMan call. The description of the individual calls indicates whether you can rely on the arrays not being changed. In addition, to assure that an input array is not inadvertently changed during the DBS call, namespace the array.

IENS: To Identify Entries and Subentries

The way to represent internal entry numbers for entries in the database is by a structure called an Internal Entry Number String (IENS). It is FileMan's way of representing the internal entry numbers for an entry in all of the DBS calls.

An IENS is a comma-delimited list of internal entry numbers beginning with the lowest level subentry and ending with the top-level entry number. Regardless of how many levels exist, a "," is appended to the end. For example, to specify subentry 2 in a multiple for entry 250, IENS would equal "2,250,". The corresponding values for the DA() array would be DA=2 and DA(1)=250 (or D0=250 and D1=2). If you were referencing the top level of the file, the IENS would be "250,"; DA=250 or D0=250. There are calls that can be used to construct an IENS from a DA() array and a DA() array from an IENS-see descriptions of DA^DILF and $$IENS^DILF.

In the simplest case, each comma-piece of the IENS is a number that directly and uniquely identifies an entry in a file or subfile. However, sometimes the client application does not know the entry number. For example, often the entry number is unknown when a call to the Updater is being made. In other situations, the client application wants the DBS to find a record and then file data in it; the entry number is unimportant to the client. In order to accommodate these circumstances, certain placeholders can be used in the IENS if the particular DBS call supports their use. The extended IENSs (those including a placeholder) are not accepted for all DBS calls. The calls that accept the extended IENSs are identified in the call's documentation.

The placeholder consists of a one- or two-character code identifying how you want the entry number derived, followed by a positive integer. The integer uniquely identifies the record involved in different nodes of the VA FileMan Data Array (FDA), as described below. The codes are:

Placeholder Code	Description
+	Add a new entry or subentry.
?	Find an entry or subentry and use it for filing.
?+	Find an entry or subentry; if one does not exist, add it (LAYGO).

Thus, if you wanted to find an entry and then to add a new subentry into that entry, your IENS might look like: "+2,?1,". Every time you referenced that top-level entry

in your FDA, you would use "?1"; every time you referenced that particular subentry, you would use "+2". A second new subentry might be "+3", and so on. See the descriptions of the Updater and Finder calls for more information about using the entry number placeholders.

FDA: Format of Data Passed to and from VA FileMan

Data is passed to and from the DBS as values in the FileMan Data Array (FDA). The FDA contains the file, internal entry numbers, and field information in its subscripting scheme.

The format of the FDA is:

```
FDA_ROOT(FILE#,"IENS",FIELD#)="VALUE"
```

FILE# The number of the file or subfile to which the data belongs.

IENS As explained above, a comma-delimited string of entry and subentry numbers. The IENS always ends with a comma.

FIELD# The number of the field being accessed.

VALUE The internal (and verified) or external (and unverified) value of the field. The specific call that you are making along with the way certain flags are set determines if the internal or external value is appropriate.

The values for word processing fields are stored in the FDA differently. Instead of setting the node equal to the actual value, set it equal to the root of an array (local or global) that holds the data. The word processing data must be stored at nodes with positive numbers in the designated array or at the 0-node descendent from those nodes. The subscripts need not be integers. For example, if the value of an FDA node were "^TMP($J,"WP")", the location of the word processing data could be:

```
^TMP($J,"WP",1,0)=Line 1
^TMP($J,"WP",2,0)=Line 2
...etc.
```

OR:

```
^TMP($J,"WP",1)=Line 1
^TMP($J,"WP",2)=Line 2
...etc.
```

For word processing data, the file and field numbers should reflect the file (or subfile) and field of the word processing field, not the subfile number of the pseudo-multiple where the word processing data is actually stored.

Nodes in the FDA can be set in several ways. The Validator call (VAL^DIE) optionally creates nodes in an FDA for valid user input. If the Validator is not being used, programmers can use a call (FDA^DILF) that creates an element in the FDA. Finally, the application developer can set the nodes manually in the client application's code.

Documentation Conventions

If a parameter must be passed by reference, that parameter is preceded by a period ("**.**") when the format for the call is shown. In the example below, the ARGUMENT array must be passed by reference:

```
CALL^DIFM(.ARGUMENT)
```

If a parameter can be passed either by reference **or** by value, it is preceded by a period enclosed in brackets ("**[.]**"). In the example below, the ARGUMENT parameter can be passed either by reference or by value.

```
CALL^DIFM([.]ARGUMENT)
```

It is very important that arrays be passed as specified in the descriptions of the calls-that is, by value or reference as indicated.

HOW THE DATABASE SERVER (DBS) COMMUNICATES

Overview

A distinguishing feature of the DBS calls is that they don't "talk"—nothing is Written to a device. The DBS communicates with the client application by passing data in arrays instead of communicating directly with the user by Writing to the screen. It is the client application's responsibility to determine if, when, and how to inform the user of the information originating from the DBS.

The way that the DBS passes *primary* information, like the value of a field when doing a Data Retriever call or a record's internal entry number when doing a Finder call, is documented for each call. *Secondary* information consists of error messages, help text, and information currently Written from nodes in the Data Dictionary by Classic FileMan calls. The way secondary information is passed to the client application is described in this section.

How Information Is Returned

Information is passed back to the client application in arrays. By default the arrays are:

```
^TMP("DIHELP",$J)   for help
^TMP("DIMSG",$J)    for other user messages
^TMP("DIERR",$J)    for error messages
```

NOTE: In traditional VA FileMan Classic calls, the first two of these types of messages are written directly to the screen; the last one did not exist or consisted solely of "<BEEP>??".

In addition, there is an output variable associated with each of these arrays. DIHELP and DIMSG equal the number of nodes of text associated with their respective arrays. DIERR has the following two pieces: number_of_errors^number_of_nodes_of_text.

If the client application wants the data returned in another array (local or global), the array's closed root should be passed as a parameter in the DBS call. The major DBS calls have a parameter to accept this root as the last parameter. Thus, if the call looks like:

```
D CALL^FM("OTHER_PARAMETERS","MYMSGS")
```

information is returned in:

```
MYMSGS("DIHELP")
MYMSGS("DIMSG")
MYMSGS("DIERR")
```

Also, the values stored in the corresponding local variables are put into the top level nodes of these arrays. When the application specifies an array for output, nothing is returned in the ^TMP arrays.

Contents of Arrays

DIHELP Array

Text in the DIHELP array has several sources. Some help text is stored in the new DIALOG file; an example of this sort of help is the text returned by %DT when you enter a "?" at a prompt requiring a date. Other help comes directly from text in the Data Dictionary. Xecutable Help relies on calls to the Loader (EN^DDIOL, see below) embedded in the executable code. The Loader call takes the place of Writes.

NOTE: In other contexts, the Loader puts text under the DIMSG subscript. However, when executing Xecutable Help, the Loader puts the text under the DIHELP subscript instead.

The following DBS call returns help for a particular field:

```
D HELP^DIE(FILE,IENS,FIELD,TYPE_OF_HELP,MSG_ROOT)
```

TYPE_OF_HELP is a set of flags that allows the client application to specify which help text (Help Prompt, Description, list of Set of Codes, Xecutable Help, etc.) to return. Alternatively, a single or double question mark returns the same information that is currently returned in scrolling mode. (See the documentation for the Helper call for details.)

If MSG_ROOT is not specified as a target, the help is returned in ^TMP("DIHELP", $J) as described above. The local variable DIHELP equals the total number of nodes returned.

Text in the array that contains help is subscripted with integers. If more than one kind of help is being returned, a null node is put between them.

If a flag is set by the client application when the CHK^DIE or VAL^DIE calls are made, help is returned when a value is found to be invalid. The help is returned in the standard way described above.

DIMSG Array

A main source of the DIMSG array is output from the Loader: EN^DDIOL. Writes that are currently embedded in the database must be changed to calls to EN^DDIOL if the DBS is to be used. When running applications in scrolling mode, the Loader simply Writes the text to the screen. However, if the node containing the EN^DDIOL call is executed from within one of the DBS calls, the DBS returns text in an array, usually subscripted by DIMSG. (For more detailed information about EN^DDIOL, see its description in the Classic FileMan API section of this manual.)

When the user is not in scrolling mode, the Loader will most frequently place the text into the DIMSG array with the local variable DIMSG set equal to the total number of lines in the array. There are certain situations, however, where the output is put into another array. As mentioned above, when the DBS HELP^DIE call is used to get help, the output of an EN^DDIOL call embedded in Xecutable Help is placed into the DIHELP array.

Like DIHELP, the DIMSG array is simply a list of lines of text.

Suppose an INPUT transform currently contains:

```
N Y S Y=$L(X) K:Y>30!(Y<3) X I '$D(X) W !,"Your input was "_Y_
" characters long.",!,"This is the wrong length."
```

It can be changed to:

```
N Y S Y=$L(X) K:Y>30!(Y<3) X I '$D(X) S Y(1)="Your input was "_Y_
" characters long.",Y(2)="This is the wrong length." D EN^DDIOL(.Y)
```

This change would have no effect if the user were in scrolling mode; the same message is written to the screen. However, if the second INPUT transform were executed from a silent call, nothing is Written and the "DIMSG" array returned to the client application might look like this:

```
^TMP("DIMSG",$J,1)="Your input was 2 characters long."
^TMP("DIMSG",$J,2)="This is the wrong length."
```

DIERR Array

When an error condition is encountered during a DBS call, an error message and other information is placed in the DIERR array. In addition, the DIERR variable is returned with the following two pieces of information: the number of errors generated during the call in the first piece and the total number of lines of the error

messages in the second. Thus, a $D check on the variable DIERR after the completion of the call allows the client application to determine if an error occurred. Both syntactical (e.g., the root of an array is not in the proper format for subscript indirection) and substantive substantive (e.g., a specified field does not exist in the specified file) errors are returned.

The information contained in the DIERR array is designed to give the client application specific information about the kind of error that occurred to allow for intelligent error handling and to provide readable error messages. Here is an example of error reporting following a Filer call:

```
>W $G(DIERR)
2^2
>D ^%G

Global ^TMP("DIERR",$J
        TMP("DIERR",$J
^TMP("DIERR",731990208,1) = 305
^TMP("DIERR",731990208,1,"PARAM",0) = 1
^TMP("DIERR",731990208,1,"PARAM",1) = ^TMP("MYWPDATA",$J)
^TMP("DIERR",731990208,1,"TEXT",1) = The array with a root of
   '^TMP("MYWPDATA",$J)' has no data associated with it.
^TMP("DIERR",731990208,2) = 501
^TMP("DIERR",731990208,2,"PARAM",0) = 3
^TMP("DIERR",731990208,2,"PARAM",1) = 89
^TMP("DIERR",731990208,2,"PARAM","FIELD") = 89
^TMP("DIERR",731990208,2,"PARAM","FILE") = 16200
^TMP("DIERR",731990208,2,"TEXT",1) = File #16200 does not contain
   a field 89.
^TMP("DIERR",731990208,"E",305,1) =
^TMP("DIERR",731990208,"E",501,2) =
```

The DIERR variable acts like a flag. It reports that two errors occurred and that they have a total of two lines of text. The ^TMP("DIERR",$J) global contains information about the errors. The next subscript is a sequence number identifying the error. The top node for each error is set equal to an error number that corresponds to an entry in the DIALOG file.

The error messages are documented in the Error Codes section. Here, the first error is 305 and the second is 501. The "PARAM" 0-node contains the number of parameters returned with the error message and the parameters themselves are identified.

The meaning of the parameters is also documented. The actual text of the error is stored in the nodes descended from "TEXT". Finally, a cross-reference exists ("E") for error message numbers. It allows you to determine quickly if a particular error occurred. For example, if you wanted to know if error number 305 happened in

order to do special error processing, you could check $D(^TMP("DIERR",$J,"E",305)).

The DIERR array is more complicated than the other arrays discussed, thereby making more information available to the client application for error handling.

Obtaining Formatted Text From The Arrays

If you want the text from any of the three arrays, the following call extracts it from the structures described above and either writes it to the screen or puts it into a local array for further use:

```
D MSG^DIALOG(FLAGS,.OUTPUT_ARRAY,TEXT_WIDTH,LEFT_MARGIN,
  INPUT_ROOT)
```

The flags for this call control whether the text is Written to the current device or moved into the output_array specified in the second parameter. The flags also direct whether the source arrays are saved or deleted and which kinds of dialog (errors, help, or other messages) are processed. Some formatting of text is also supported. See the description of MSG^DIALOG in this DBS section for details of its use.

Cleaning Up the Output Arrays

When you make a DBS call and use the default arrays in the ^TMP global for output of help, user, and error messages, the DBS call kills off these arrays and their related variables at the start of the call. Therefore, you know that any data that exists after the call was generated by that call.

If you don't use the default arrays for output, however, and instead specify your own arrays for this information to be returned in, your arrays are not automatically killed at the start of a DBS call. So if there is any chance that these arrays might already exist, you should kill them yourself before making the DBS call.

After making a DBS call, if you used the default arrays in ^TMP for output of help, user, and error messages, you should delete these arrays before your application Quits. To do this, use the following call:

```
D CLEAN^DILF
```

See the description of CLEAN^DILF later in this DBS section for details of its use.

If you are using your own arrays for output, however, you need to clean up your arrays yourself. You should still call CLEAN^DILF to kill off the variables related to these arrays, however.

Example of Call to VA FileMan DBS

One of the DBS calls validates data. If the data is valid, the internal representation of that data is returned. If the data is invalid, an up-arrow (^) is returned along with various messages, optionally including the relevant help text. The validate call looks like this (see the Validator documentation for details):

```
VAL^DIE(FILE,IENS,FIELD,FLAGS,VALUE,.RESULT,FDA_ROOT,MSG_ROOT)
```

Your call might look like this:

```
D VAL^DIE(999000,"223,",4,"H","AB",.MYANSWER,"","MYMSGS(""WIN3"")")
```

If MYANSWER equaled "^" after the call, your MYMSGS("WIN3") array might look like :

```
MYMSGS("WIN3","DIERR")=1^1
MYMSGS("WIN3","DIERR",1)=701
MYMSGS("WIN3","DIERR",1,"PARAM",0)=4
MYMSGS("WIN3","DIERR",1,"PARAM",3)="AB"
MYMSGS("WIN3","DIERR",1,"PARAM","FIELD")=4
MYMSGS("WIN3","DIERR",1,"PARAM","FILE")=999000
MYMSGS("WIN3","DIERR",1,"PARAM","IENS")="223"
MYMSGS("WIN3","DIERR",1,"TEXT",1)="The value 'AB' for field ALPHA
  DATA in file TEST1 is not valid."
MYMSGS("WIN3","DIERR","E",701,1)=""
MYMSGS("WIN3","DIHELP")=1
MYMSGS("WIN3","DIHELP,1)="Answer must be 3-30 characters in length."
MYMSGS("WIN3","DIMSG")=1
MYMSGS("WIN3","DIMSG",1)="Your input was 2 characters long."
MYMSGS("WIN3","DIMSG",2)="This is the wrong length."
```

The DIERR portion of this array indicates that error number 701 is being reported. Documentation makes clear that this means that an input value was invalid. The PARAM nodes (also documented) give the client application the relevant file#, field#, IENS, and value. This information might be used by the application in its error handling. The TEXT node contains the error message; note that it is customized to include specifics of the current error. The DIHELP node contains single-question-mark help for the field. The DIMSG nodes contain a message generated by the INPUT transform via an EN^DDIOL call. (The sample INPUT transform discussed in the DIMSG section above produced this message.)

Now, the client application decides what (if anything) to show the user. In a GUI environment, you might decide to put the error message along with any text from the INPUT transform into a document gadget. A HELP button that could be used by the user to display the help information might be added to the box. FileMan's DBS has provided text; the client application is in complete control regarding the use of this text.

DATABASE SERVER CALLS CROSS-REFERENCED BY CATEGORY

Data Dictionary

FIELD^DID
FIELDLST^DID
FILE^DID
FILELST^DID
$$GET1^DID
$$FLDNUM^DILFD
PRD^DILFD
$$ROOT^DILFD
$$VFIELD^DILFD
$$VFILE^DILFD

Data Dictionary Modification

DELIX^DDMOD
DELIXN^DDMOD

Data Editing

CHK^DIE
FILE^DIE
HELP^DIE
$$KEYVAL^DIE
UPDATE^DIE
VAL^DIE
VALS^DIE
WP^DIE
RECALL^DILFD

Data Retrieval

$$GET1^DIQ
GETS^DIQ

Lookup

FIND^DIC
$$FIND1^DIC
LIST^DIC

User Dialog

BLD^DIALOG
$$EZBLD^DIALOG
MSG^DIALOG

Utilities

CLEAN^DILF
$$CREF^DILF
DA^DILF
DT^DILF
FDA^DILF
$$HTML^DILF
$$IENS^DILF
$$OREF^DILF
$$VALUE1^DILF
VALUES^DILF
$$EXTERNAL^DILFD

DATABASE SERVER (DBS) CALLS PRESENTED IN ALPHABETICAL ORDER)

This section lists and describes the VA FileMan Database Server (DBS) calls in alphabetical order. However, the table above cross-references the DBS calls by category:

DELIX^DDMOD: Traditional Cross-reference Deleter

This procedure deletes a traditional cross-reference definition from the data dictionary of a file. Optionally, it deletes the data in the index or executes the kill logic for all entries in the file. Compiled input templates that contain the field on which the cross-reference is defined are recompiled. If cross-references on the file are compiled, they are recompiled.

DELIX^DDMOD can be used is the pre-install or post-install routine of a KIDS (Kernel Installation and Distribution System) Build, for example, to delete a traditional cross-reference from the installing site.

See DELIXN^DDMOD for information on the call to delete a new-style index definition.

Format

```
DELIX^DDMOD(FILE,FIELD,CROSS_REF,FLAGS,OUTPUT_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number.

FIELD (Required) Field number.

CROSS_REF (Required) Cross-reference number. Traditional cross-references are defined in the data dictionary under^DD(file#,field#,1,cross reference number)

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

K For Regular, KWIC, Mnemonic, and Soundex-type cross-references, delete the data in the index. For MUMPS and Trigger-type cross-references, execute the **K**ill logic of the cross-reference for all entries in the file. For Bulletin-type cross-references, the "K" flag is ignored; the kill logic for Bulletin-type cross-references is never executed by this procedure.

W **W**rite messages to the current device as the index is deleted and cross-references and input templates are recompiled.

OUTPUT_ROOT (Optional) The name of the array that should receive information about input templates and cross-references that may have been recompiled and a flag to indicate that the deletion was audited in the DD Audit file (#.6). See Output below. This must be a closed root, either local or global.

MSG_ROOT (Optional) The name of the array that should receive any error messages. This must be a closed root, either local or global. If not passed, errors are returned descendent from ^TMP("DIERR",$J).

Output

OUTPUT_ROOT See OUTPUT_ROOT under Input Parameters.

If the field on which the deleted cross-reference was defined is used in any compiled input templates, those input templates are recompiled. Information about the recompiled input templates is stored descendant from OUTPUT_ROOT("DIEZ"):

OUTPUT_ROOT("DIEZ",input template #) =
input template name ^ file # ^compiled routine name

If cross-references for the file are compiled, they are recompiled, and the compiled routine name is stored in OUTPUT_ROOT("DIKZ"):

OUTPUT_ROOT("DIKZ") = compiled routine name

If the data dictionary for the file is audited, an entry is made in the DD Audit file (#.6) and OUTPUT_ROOT("DDAUD") is set to 1:

OUTPUT_ROOT("DDAUD") = 1

Examples

Example 1

In this example, regular cross-reference #4 (the "C" index), defined on field #12 in file #16200, is deleted. The "K" flag indicates that the entire ^DIZ(16200,"C") index should be removed from the file.

```
>D DELIX^DDMOD(16200,12,4,"K","MYOUT")

>ZW MYOUT

MYOUT("DDAUD")=1
MYOUT("DIEZ",100)=ZZTEST EDIT^16200^ZZIT
MYOUT("DIKZ")=ZZCR
```

The MYOUT output array indicates that the deletion was recorded in the DD Audit file (#.6). The input template ZZTEST EDIT (#100) was recompiled into the ZZIT namespaced routines, because field #12 is used in that template. Cross-references on file #16200 are recompiled under the ZZCR namespace.

Example 2

In this example, the whole-file regular cross-reference #7 (the "N" index), defined on field #15 within subfile #16200.075, is deleted. The "K" flag indicates that the entire ^DIZ(16200,"N") index should be removed, and the "W" flag indicates that messages should be printed to the current device.

```
>D DELIX(16200.075,15,7,"KW")

Removing index ...
Deleting cross-reference definition ...


Compiling ZZ TEST CR Input Template of File 16200...
'ZZIT1' ROUTINE FILED..
'ZZIT' ROUTINE FILED....
'ZZIT2' ROUTINE FILED.

Compiling Cross-Reference(s) 16200 of File 16200.

...SORRY, HOLD ON...

'ZZCR1' ROUTINE FILED.
'ZZCR2' ROUTINE FILED.
'ZZCR3' ROUTINE FILED.
'ZZCR4' ROUTINE FILED.
'ZZCR5' ROUTINE FILED.
'ZZCR' ROUTINE FILED.
```

Error Codes Returned

202 The specified parameter is missing or invalid.

301 The passed flags are incorrect.

401 The file does not exist.

406 The file has no .01 definition.

407 A word-processing field is not a file.

501 The file does not contain the specified field.

DELIXN^DDMOD: New-Style Index Deleter

This procedure deletes a new-style index definition from the Index file. Optionally, it deletes the data in the index or executes the kill logic for all entries in the file. Compiled input templates that contain one or more of the fields defined in the index are recompiled. If cross-references on the file are compiled, they are recompiled.

DELIXN^DDMOD can be used is the pre-install or post-install routine of a KIDS (Kernel Installation and Distribution System) Build, for example, to delete a new-style index from the installing site.

See DELIX^DDMOD for information on the call to delete a traditional cross-reference definition.

Format

```
DELIXN^DDMOD(FILE,INDEX,FLAGS,OUTPUT_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number. For whole-file indexes, this is the number of the file at the upper level where the data in the index resides.

INDEX (Required) Index name.

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

- **K** For Regular indexes, delete the data in the index. For MUMPS indexes, execute the **K**ill logic for all entries in the file.
- **W** **W**rite messages to the current device as the index is deleted and cross-references and input templates are recompiled.

OUTPUT_ROOT (Optional) The name of the array that should receive information about input templates and cross-references that may have been recompiled. See Output below. This must be a closed root, either local or global.

MSG_ROOT (Optional) The name of the array that should receive any error messages. This must be a closed root, either local or global. If not passed, errors are returned descendent from ^TMP("DIERR",$J).

Output

OUTPUT_ROOT See OUTPUT_ROOT under Input Parameters.

If a field used in the index is used in any compiled input templates, those input templates are recompiled. Information about the recompiled input templates is stored descendant from OUTPUT_ROOT("DIEZ"):

OUTPUT_ROOT("DIEZ",input template #) =

input template name ^ file # ^ compiled routine name

If cross-references for the file are compiled, they are recompiled, and the compiled routine name is stored in OUTPUT_ROOT("DIKZ"):

OUTPUT_ROOT("DIKZ") = compiled routine name

Examples

Example 1

In this example, the new-style "G" index defined on file #16200 is deleted. The "K" flag indicates that the entire ^DIZ(16200,"G") index should be removed from the file.

```
>D DELIXN^DDMOD(16200,"G","K","MYOUT")

>ZW MYOUT
MYOUT("DIEZ",94)=ZZ TEST^16200^ZZIT
MYOUT("DIEZ",100)=ZZ TEST A^16200^ZZITA
MYOUT("DIKZ")=ZZCR
```

The MYOUT output array indicates that a field or fields used in the deleted index are also used in the compiled input templates ZZ TEST (#94) and ZZ TEST 2 (#100).

Those two input templates were recompiled. Cross-references on file #16200 were also recompiled under the ZZCR namespace.

Example 2

In this example, the whole-file regular index (the "J" index) is deleted. The fields in the index come from fields in a multiple, subfile #16200.075, but the whole-file index resides at the top-level file #16200. The "K" flag indicates that the entire ^DIZ(16200,"J") index should be removed, and the "W" flag indicates that messages should be printed to the current device.

```
>D DELIXN^DDMOD(16200,"J","KW","MYOUT")

Removing index ...
Deleting index definition ...


Compiling ZZ TEST Input Template of File 16200....
'ZZIT' ROUTINE FILED....
'ZZIT1' ROUTINE FILED.


Compiling ZZ TEST A Input Template of File 16200....
'ZZITA' ROUTINE FILED....
'ZZITA' ROUTINE FILED.

Compiling Cross-Reference(s) 16200 of File 16200.

...SORRY, JUST A MOMENT PLEASE...


'ZZCR1' ROUTINE FILED.
'ZZCR2' ROUTINE FILED.
'ZZCR3' ROUTINE FILED.
'ZZCR4' ROUTINE FILED.
'ZZCR5' ROUTINE FILED.
'ZZCR6' ROUTINE FILED.
'ZZCR7' ROUTINE FILED.
'ZZCR8' ROUTINE FILED.
'ZZCR9' ROUTINE FILED.
'ZZCR10' ROUTINE FILED.
'ZZCR' ROUTINE FILED.
```

Error Codes Returned

202 The specified parameter is missing or invalid.

301 The passed flags are incorrect.

BLD^DIALOG( ): DIALOG Extractor

This entry point performs the following functions:

1. Extracts a dialog from a FileMan DIALOG file entry
2. Substitutes dialog parameters into the text if requested
3. Returns the text in an array

If the DIALOG entry has POST MESSAGE ACTION code, this code is executed after the message has been built, but before quitting.

Format

```
BLD^DIALOG(DIALOG#,[.]TEXT_PARAM,[.]OUTPUT_PARAM, OUTPUT_ARRAY,FLAGS)
```

Input Parameters

DIALOG#

(Required) Record number from the DIALOG file for the text to be returned.

[.]TEXT_PARAM

(Optional) Local array containing the dialog parameters to substitute into the resulting text. Set the subscript of each node in this array to a dialog parameter that's in a |window| in the referenced DIALOG entry's text. The value of each node should be in external, printable format and will be substituted in the Dialog text for that dialog parameter.

If there is only one parameter in the list, you can pass its value in a local variable or as a literal, otherwise, pass by reference.

[.]OUTPUT_PARAM

(Optional) This is useful mainly if you are returning error messages as part of an API for other programmers to use. Use it to pass dialog parameters back to the user of your API, such that they can be accessed individually instead of just being embedded in the error text.

Use only with DIALOG file entries of type Error. Pass this local array by reference. Subscript each node by the parameter name and set the node to the

corresponding parameter value. The parameter values can be in any format (external or internal).

For example, if you pass DIPAROUT by reference and want to pass back standalone values for the '1' and 'FILE' parameters in the output array along with dialog text, set DIPAROUT to:

```
DIPAROUT(1)=TEST FILE
DIPAROUT("FILE")=662001
```

Dialog text will be returned as expected but, in addition, dialog parameter values will be returned in:

```
^TMP("DIERR",$J,msg#,"PARAM",1)
^TMP("DIERR",$J,msg#,"PARAM","FILE")
```

NOTE: If you only want to return one parameter, you can pass its value in a local variable or as a literal rather than in an array by reference. BUT the subscript for such a parameter in the output array is always 1.

OUTPUT_ARRAY

(Optional) If provided, the text will be output in the local or global array named by this parameter. If this parameter is null, output is returned in the ^TMP global, under the "DIERR", "DIHELP", or "DIMSG" subscripts as documented in the DBS Contents of Arrays section.

If you specify DIR("A") or DIR("?") as the output array, special handling is provided for populating the output array for use in a call to the Reader, ^DIR. Text is output in the format needed for input to the Reader.

NOTE: You are responsible for cleaning up the output array or global before calling BLD^DIALOG. If the array already exists, BLD^DIALOG simply appends its output to the current contents of the output array, under a new message subscript.

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

S **S**uppress the blank line that is normally inserted between discrete blocks of text that are built by separate calls to this routine.

F **F**ormat the local array similar to the default output format of the ^TMP global, so that MSG^DIALOG can be called to either Write the array to the current device or to a simple local array.

Output

If the OUTPUT_ARRAY input parameter is not passed, Dialog text is returned in ^TMP under the "DIERR", "DIHELP", or "DIMSG" subscripts as documented in the DBS Contents of Arrays section. If the DIALOG text is returned in a local array instead, the name of the array and leading subscript(s) are defined by the name of the array passed to this routine.

In addition to the DIALOG text, a local variable is returned. The local variable is one of the following:

Variable Name	Returned if Dialog Type Is:	Variable Value
DIERR	Error	Piece 1: # of discrete error messages returned Piece 2: Total # of lines of text returned
DIHELP	Help	Total # of lines of text returned
DIMSG	General Message	Total # of lines of text returned

NOTE:
(1) If the variable to be used (DIHELP, DIERR, or DIMSG) already exists before calling BLD^DIALOG, the number or numbers already stored in the variable are incremented (not overwritten) to reflect the cumulative total over repetitive calls to BLD^DIALOG.

The local variable (DIHELP, DIERR, or DIMSG) is not set if you ask for text to be built in the special variables DIR("A") and DIR("?"), used as input to ^DIR.

(2) If you wish to add entries to the DIALOG file, you must use a number-space assigned by the Data-Base Administrator. Please see Developer Tools, Dialog File section in this Programmer Manual for more information.

Examples

The DIALOG entry numbers shown in the examples below are for demonstration purposes and are not distributed as part of the VA FileMan package.

Example 1

In the case of errors, the output looks like the following example. ^TMP("DIERR",$J,error_number) is set equal to the IEN from the DIALOG file. The actual error text is contained descendent from the "TEXT" subscript. If output parameters were passed to the routine, they are returned descendent from the "PARAM" subscript, where "PARAM",0) contains the total number of output parameters. Finally, there is an entry descendent from "E", where the next subscript is the IEN from the DIALOG file, and the final subscript refers to the error number in this output array. This serves as a sort of cross-reference by error code. When errors are generated by a routine called from developers' code, this cross-reference can be used by the developer to quickly check whether a specified error had been generated:

```
DIPAROUT(1)=TEST FILE
DIPAROUT("FILE")=662001

>D BLD^DIALOG(10999,"Myfile",.DIPAROUT)
```

The output looks like:

```
DIERR=1^1

^TMP("DIERR",591465626,1) = 10999
^TMP("DIERR",591465626,1,"PARAM",0) = 2
^TMP("DIERR",591465626,1,"PARAM",1) = TEST FILE
^TMP("DIERR",591465626,1,"PARAM","FILE") = 662001
^TMP("DIERR",591465626,1,"TEXT",1) = Entries in file Myfile cannot
   be edited.
^TMP("DIERR",591465626,"E",10999,1) =
```

Example 2

Here we generate a second error to show how it is appended to the previous error in the ^TMP global:

```
DIPARIN(1)='B'
DIPARIN("FILE")=662001
DIPAROUT(1)='B'
DIPAROUT("FILE")=662001

>D BLD^DIALOG(10202,.DIPARIN,.DIPAROUT)
```

Now the output looks like this:

```
DIERR=2^2

^TMP("DIERR",591465626,1) = 10999
^TMP("DIERR",591465626,1,"PARAM",0) = 2
^TMP("DIERR",591465626,1,"PARAM",1) = TEST FILE
^TMP("DIERR",591465626,1,"PARAM","FILE") = 662001
^TMP("DIERR",591465626,1,"TEXT",1) = Entries in file Myfile cannot
    be edited.
^TMP("DIERR",591465626,2) = 10202
^TMP("DIERR",591465626,2,"PARAM",0) = 2
^TMP("DIERR",591465626,2,"PARAM",1) = 'B'
^TMP("DIERR",591465626,2,"PARAM","FILE") = 662001
^TMP("DIERR",591465626,2,"TEXT",1) = There is no 'B' index for File
    #662001.
^TMP("DIERR",591465626,"E",10999,1) =
^TMP("DIERR",591465626,"E",10202,2) =
```

Example 3

In this example, we build the same error message as in Example 1, but this time we put the output into a local array. Notice that we do not send a flag in the FLAGS parameter for this call, so only the error text is returned. This would ordinarily be done when the developer planned to process the output from their own routine.

```
>D BLD^DIALOG(10999,"Myfile",.DIPAROUT,"MYARRAY")
```

The output looks like:

```
DIERR=1^1

MYARRAY(1)=Entries in file Myfile cannot be edited.
```

Example 4

In this example, we build the same error message as in Example 3, again sending the output to a local array. This time, however, we will pass the F flag in the FLAGS parameter so that all of the error information is returned in a format similar to that of the ^TMP global, but without the $J subscript. In this format, the developer could then call MSG^DIALOG to either write the array to the current device or to copy the text into a simple array. This might, for example, be done when the developer wanted to examine the error messages returned and kill some of them before having FileMan write the remaining messages.

```
>D BLD^DIALOG(10999,"Myfile",.DIPAROUT,"MYARRAY","F")
```

The output looks like:

```
DIERR=1^1

MYARRAY("DIERR",1)=10999
MYARRAY("DIERR",1,"PARAM",0)=2
MYARRAY("DIERR",1,"PARAM",1)=TEST FILE
MYARRAY("DIERR",1,"PARAM","FILE")=662001
MYARRAY("DIERR",1,"TEXT",1)=Entries in file Myfile cannot be     edited.
MYARRAY("DIERR","E",10999,1)=
```

Example 5

In this example, we build a help message with a single input parameter. Notice that the only output is the DIHELP variable and the text. Similarly, other types of messages only return the DIMSG variable and the text.

```
>D BLD^DIALOG(10335,"PRINT")
```

The output looks like:

```
DIHELP=4

^TMP("DIHELP",591469242,1) = This number will be used to determine
    how large to make the generated
^TMP("DIHELP",591469242,2) = compiled PRINT routines.  The size
    must be a number greater
^TMP("DIHELP",591469242,3) = than 2400, the larger the better, up
    to the maximum routine size for
^TMP("DIHELP",591469242,4) = your operating system.
```

Example 6

Now we build the same help message as Example 5 but put it into a local array.

```
>D BLD^DIALOG(10335,"PRINT","","MYARRAY")
```

Now the output looks like:

```
DIHELP=4

MYARRAY(1)=This number will be used to determine how large to make
    the generated
MYARRAY(2)=compiled PRINT routines.  The size must be a number
    greater
MYARRAY(3)=than 2400, the larger the better, up to the maximum
    routine size for
MYARRAY(4)=your operating system.
```

Example 7

In this final example, we build the same help message as in Example 6 but put it into the special array DIR("?"). Note that for the special local variables used for calls to the FileMan Reader, ^DIR, this call puts the text into the format that the Reader expects. It does not set the DIMSG, DIHELP, or DIERR variables.

```
>D BLD^DIALOG(10335,"PRINT","","DIR(""?"")")
```

The output looks like:

```
DIR("?")=your operating system.
DIR("?",1)=This number will be used to determine how large to make
   the generated
DIR("?",2)=compiled PRINT routines.  The size must be a number greater
DIR("?",3)=than 2400, the larger the better, up to the maximum routine
   size for
```

Error Codes Returned

None

$$EZBLD^DIALOG( ): DIALOG Extractor (Single Line)

This extrinsic function returns the first line of text from an entry in the DIALOG File. It can be used when the text entry is only one line and when the output does not need to be put into an array. For example, use it to extract a single word or short phrase to use as a text parameter to embed into another DIALOG file entry. If the DIALOG entry has POST MESSAGE ACTION code, this code is executed after the message has been built but before quitting.

Format

```
$$EZBLD^DIALOG(DIALOG#,[.]TEXT_PARAM)
```

Input Parameters

DIALOG#	(Required) Record number from the DIALOG File for the text to be returned.
[.]TEXT_PARAM	(Optional) Name of local array containing the parameter list for those parameters that are to be incorporated into the resulting text. These parameters should be in external, printable format. If there is only one parameter in the list, it can be passed in a local variable or as a literal.

Output

This extrinsic function returns the first line of text from a DIALOG file entry. No output variables are returned.

NOTE: If you wish to add entries to the DIALOG file, you must use a number-space assigned by the Data-Base Administrator. Please see Developer Tools, Dialog File section in this Programmer Manual for more information.

Examples

Example 1

To write a single line of text with no parameters, do the following:

```
>W $$EZBLD^DIALOG(110)
The record is currently locked.
```

Example 2

To write a single line of text with a single parameter passed as a literal, do the following:

```
>W $$EZBLD^DIALOG(201,"PARAM")
The input variable PARAM is missing or invalid.
```

Example 3

To write a single line of text with parameters in an input array, do the following:

```
>S TESTPAR(1)="PAR2"
>W $$EZBLD^DIALOG(201,.TESTPAR)
The input variable PAR2 is missing or invalid.
```

Error Codes Returned

None

MSG^DIALOG( ): Output Generator

This procedure takes text from one of the FileMan dialog arrays (for errors, help text, or other text) or from a similarly structured local array, writes it and/or moves it into a simple local array.

The subscripting of these arrays will tell the developer whether the dialog is a "Help" message, an "Error" message, or other dialog, such as a prompt. Different combinations of these messages may be returned from the DBS calls. In addition, error messages will be returned whenever an error occurs, either in the way the call was made or in attempting to interact with the database.

With the DBS calls, it becomes the job of the developer to display dialog to the end user as needed, perhaps in a GUI box or in the bottom portion of a screen-oriented form. The developer can also save error messages in a file.

MSG^DIALOG is designed to make it easier for the developer to use the dialog arrays. The developer can use MSG^DIALOG to do simple formatting of the dialog and to either write dialog to the current device or to move the dialog to a simple local array for further processing.

Format

```
MSG^DIALOG(FLAGS,.OUTPUT_ARRAY,TEXT_WIDTH,LEFT_MARGIN,INPUT_ROOT)
```

Input Parameters

FLAGS (Optional) Flags to control processing. If none of the text-type flags (E, H or M) is entered, the routine behaves as if E were entered. If no flags are entered, it behaves as if FLAGS contained WE. The possible values are:

- **A** Local **A**rray specified by the second parameter receives the text.
- **W** **W**rites the text to the current device.
- **S** **S**aves the ^TMP or other designated input array (does not kill the array).
- **E** **E**rror array text is processed.

H **H**elp array text is processed.

M **M**essage array text (other text) is processed.

B **B**lank lines are suppressed between error messages.

T Return **T**otal number of lines in the top level node of the local array specified by the second parameter.

.OUTPUT_ARRAY (Optional) This parameter contains the name of the local array to which the text is to be written. If FLAGS contains an A, this parameter must be sent. Otherwise, the parameter is ignored. Note that the output array is killed before the text is added, not appended to what is already there.

TEXT_WIDTH (Optional) Maximum line length for formatting text. If specified, the text is broken into lines of this length when writing to the current device or when moving the text to the OUTPUT_ARRAY. Lines are not "joined" to fill out to this width.

If you don't specify TEXT_WIDTH:

Text that is displayed on the current device is formatted to a line length of IOM-5 if IOM is defined, or to 75 characters otherwise.

Text written to an OUTPUT_ARRAY is not reformatted.

LEFT_MARGIN (Optional) Left margin for writing text. If sent, the text is lined up in a column starting at this column number. Otherwise, the text is lined up with the left margin (column 0). This parameter has no effect on text sent to an array (A flag).

INPUT_ROOT (Optional) Closed root of local input array in which text resides. If the text resides in a local array, this parameter must be sent. The last non-variable subscript of the local array must describe the type of text it contains, as the ^TMP global normally does ("DIERR" for

errors, "DIHELP" for help text, or "DIMSG" for other text).

Output

If W is passed in the FLAGS parameter, the text is written to the current device. If A is passed in the FLAGS parameter, the text is written to the local array whose name is specified in the second parameter. The format of that array is:

ARRAY Total number of lines (only returned if the T flag is passed in the FLAGS parameter).

ARRAY(n) A line of formatted text (n=sequential integer starting with 1).

If FLAGS does NOT contain S, then the input array and associated local variables (DIMSG, DIHELP, DIERR) are killed.

NOTE: If you wish to add entries to the DIALOG file, you must use a number-space assigned by the Data-Base Administrator. Please see Developer Tools, Dialog File section in this Programmer Manual for more information.

Examples

Example 1

In this first example, we want to write the error text to the current device and kill the input array. Notice that because no flags are sent to the call, the default flags for Write Error message (WE) are assumed. Thus, the call will write the single error message "The record is currently locked," from the "DIERR" portion of the ^TMP global. It will also kill ^TMP("DIERR",$J) and the local variable DIERR as follows:

```
^TMP("DIERR",698526778,1) = 110
^TMP("DIERR",698526778,1,"TEXT",1) = The record is currently
    locked.
^TMP("DIERR",698526778,"E",110,1) =

^TMP("DIHELP",698526778,1) = This number will be used to determine
    how large to make the generated
^TMP("DIHELP",698526778,2) = compiled PRINT TEMPLATE routines.  The
    size must be a number greater
^TMP("DIHELP",698526778,3) = than 2400, the larger the better, up
    to the maximum routine size for
^TMP("DIHELP",698526778,4) = your operating system.

^TMP("DIMSG",698526778,1) = Records from list on ZZMYARRAY SEARCH
    template.
```

Then, write the error text to the current device and kill the input array:

```
>D MSG^DIALOG()
The record is currently locked.
```

Example 2

In this example, we want to write the help text from the "DIHELP" subscripted portion of the ^TMP global, both to the current device and to the local 'MYARRAY' array. In addition, we want to format each line to 50 as follows:

```
>D MSG^DIALOG("HAW",.MYARRAY,50,5)
```

This number will be used to determine how large to make the generated compiled PRINT template routines. The size must be a number greater than 2400, the larger the better, up to the maximum routine size for your operating system.

```
>ZW MYARRAY
MYARRAY(1)=This number will be used to determine how large to
MYARRAY(2)=make the generated
MYARRAY(3)=compiled PRINT TEMPLATE routines.  The size must
MYARRAY(4)=be a number greater
MYARRAY(5)=than 2400, the larger the better, up to the
MYARRAY(6)=maximum routine size for
MYARRAY(7)=your operating system.
```

Example 3

In the third example, help text was returned from a DBS call in a local array. This was done because the developer specified to the DBS call that dialog was to be returned in its own local array rather than in the ^TMP global. Suppose our local array looks like this:

```
MYHELP("DIHELP",1)=This number will be used to determine how large
    to make the generated
MYHELP("DIHELP",2)=compiled PRINT TEMPLATE routines.  The size must
    be a number greater
MYHELP("DIHELP",3)=than 2400, the larger the better, up to the
    maximum routine size for
MYHELP("DIHELP",4)=your operating system.
```

If the developer wishes to write the text to the current device and to preserve the 'MYHELP' local array, the call and the results will look like this:

```
>D MSG^DIALOG("WSH","","","","MYHELP")
```

This number will be used to determine how large to make the generated compiled PRINT template routines. The size must be a number greater than 2400, the larger the better, up to the maximum routine size for your operating system.

Error Codes Returned

None

FIND^DIC( ): Finder

This procedure finds records in a file based on input value(s). The caller must specify a file number and the input values to be used for the lookup. The caller can also specify the index(es) to be used in the search, the data to output, and a number of records to retrieve. The caller can also pass screening logic. By default, the Finder returns the IEN and the .01 field of the entries along with all identifiers. The developer can override the default output and return other information for the entries.

This call was designed as a non-interactive lookup, to find entries that are at least a partial match to the lookup values input to the call. This procedure cannot file data or add new records.

NOTE: The Finder does NOT honor the Special Lookup or Post-Lookup Action nodes defined in the data dictionary for a file.

Format

```
FIND^DIC(FILE,IENS,FIELDS,FLAGS,[.]VALUE,NUMBER,[.]INDEXES,[.]SCREEN,
IDENTIFIER,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE
(Required) The number of the file or subfile to search. If this parameter is a subfile, it must be accompanied by the IENS parameter.

IENS
(Optional) The IENS that identifies the subfile, if FILE is a subfile number. To identify a subfile, rather than a subfile entry, leave the first comma-piece empty. For example, a value of ",67," indicates that the subfile within entry #67 should be used. If FILE is a file number, this parameter should be empty. Defaults to no subfile.

FIELDS
(Optional) The fields to return with each entry found. This parameter can be set equal to any of the specifications listed below. The individual specifications should be separated by semicolons (";").

NOTE: In most cases, a developer will want to include the "@" specifier (described below) to suppress the default output values normally returned by the Finder and then

specify the fields and other elements to return here in the FIELDS parameters. This gives the developer full control over exactly what will be returned in the output list and makes the call more self-documenting in the developer's code.

- **Field Number:** This specifier causes the Finder to return the value of the field for each record found. For example, specifying .01 returns the value of the .01 field. You can specify computed fields. You cannot specify word processing or multiple fields. By default, fields will be returned in external format. The "I" suffix (described below) can be appended to the field number to get the internal format of the field.

- **IX:** This returns for each record, the value(s) from the index on which the lookup match was made. The number of index values returned will depend on the number of data value subscripts in the starting lookup index. If a subscript in the index is derived from a field, the external format of that field will be returned by default. Otherwise, the value will be returned directly as it appears in the index. The "I" suffix (described below) can be appended to IX to get the internal index values. The index values are returned in the "ID" nodes as described in the Output section below.

  NOTE: For records located on a mnemonic index entry, the value from the index entry will always be returned, rather than its corresponding external field value.

- **FID:** This returns the fields display identifiers (i.e., field identifiers). By default, the field values are returned in external format. The "I" suffix (described below) can be appended to FID to get the internal format of the field identifiers.

- **WID:** This returns the fields WRITE (display only) identifiers. The Finder executes each WRITE identifier's M code and copies contents of ^TMP("DIMSG",$J) to the output. You must ensure that the WRITE identifier code issues no direct I/O, but instead calls EN^DDIOL.
  NOTE: The "I" suffix, described below, cannot be used with "WID" and will generate an error.

- **E suffix:** You can append an "E" to a field number, the specifier "IX", or the specifier "FID" to force the fields to be returned in external format. You can use both the "E" and "I" suffix together (ex., .01EI) to return both the internal and external value of the field.

- **I suffix:** You can append an "I" to a field number, the specifier "IX", or the specifier "FID" to force the fields to be returned in internal format. You can use both the "E" and "I" suffix together (ex., .01IE) to return both the internal and external value of the field.

- **- prefix:** A minus sign (-) prefixing one of the other field specifiers tells the Finder to exclude it from the returned list. This could be used, for example, in combination with the "FID" specifier to exclude one of the identifier fields. For example, if field 2 was one of the field identifiers for a file, "FID;-2" would output all of the field identifiers except for field 2.

- **@:** This suppresses all the default values normally returned by the Finder, except for the IEN and any fields and values specified in the FIELDS parameter. It is recommended that developers ALWAYS use the "@" specifier in Finder calls. Use of the "@" specifier allows the developer to control exactly what will be returned in the output. See below for the default values normally returned by the Finder.

Default Values

If you do not pass anything in the FIELDS parameter, the Finder returns:

1. The IEN
2. The .01 field in internal format
3. Any field display identifiers
4. Any WRITE (display-only) identifiers
5. The results of executing the Finder's IDENTIFIER parameter

If you do pass a FIELDS parameter, the Finder returns (unless you use the @ field specifier):

1. The IEN
2. The .01 field in internal format
3. The fields and values specified by the FIELDS parameter
4. Any WRITE (display-only) identifiers
5. The results of executing the Finder's IDENTIFIER parameter

FLAGS

(Optional) Flags to control processing. This parameter lets the caller adjust the Finder's algorithm. The possible values are:

A **A**llow pure numeric input to always be tried as an IEN. Normally, the Finder will only try pure numbers as IENs if: 1) the file has a .001 field, or 2) its .01 field is not numeric and the file has no lookup index.

When this flag is used, records that match other numeric interpretations of the input will be found in addition to a record with a matching IEN. For example, a lookup value of "2" would match a record with a lookup field of "2JOHN" as well as a record with an IEN of 2. If more than one match is found, all matching records are returned.

NOTE: If the numeric lookup value is preceded by an accent grave character (`), lookup interprets the input as an IEN, and only attempts to match by IEN. The A flag is not required in this case.

B **B** index used on lookups to pointed-to files. Without the B flag, if there are cross-referenced pointer fields in the list of indexes to use for lookup then: (1.) for each cross-referenced pointer field, FileMan checks ALL lookup indexes in each pointed-to file for a match to X (time-consuming), and (2.) if X matches

any value in any lookup index (not just on the .01 field) in a pointed-to file and the IEN of the matched entry is in the home file's pointer field cross-reference, FileMan considers this a match (perhaps not the lookup behavior desired).

The B flag prevents this behavior by looking for a match to X only in the "B" index (.01 field) of files pointed to by cross-referenced pointer fields. This makes lookups quicker and avoids the risk of FileMan matching an entry in the pointed-to file based on something other than the .01 field.

See the Details and Features section for an explanation of the "Lookup Index" and the Examples section for more information on use of the B flag.

C Use the **C**lassic way of performing lookups on names, i.e., like the classic FileMan lookup routine ^DIC. If C is passed in the FLAGS parameter and, for example, the user enters a lookup value of "Smi,J", the Finder will find "Smith,John" but also "Smiley,Bob J." The Finder takes the first comma piece of the lookup value "Smi", and looks for partial matches to that. It then takes the second comma piece of the lookup value "J" and looks for partial matches to "J" on the second or any other piece of the value on the entry being examined. It uses any punctuation or space for a delimiter.

The default, without passing C in the FLAGS parameter, will look for partial matches ONLY on the second piece, thus in our example, finding "Smith,John" but not "Smiley,Bob J.". The old style of comma-piece processing can be quite slow, especially with common names like "Smith".

K Primary **K**ey used for starting index. If no index is specified in the INDEXES parameter, this flag causes the Finder to use the Uniqueness index for the Primary Key as the starting index for the search. Without the K flag, or if there is no Primary Key for this file (in the KEY file), the Finder defaults to the "B" index.

M **M**ultiple index lookup allowed. If more than one index is passed in the INDEXES parameter, all indexes in the list are searched. Otherwise, the M flag causes the Finder to search the starting index and all indexes that alphabetically follow it. This includes both indexes from the traditional location in the data dictionary, as well as lookup indexes defined on the INDEX file that have an "L" (for LOOKUP) in the new "Use" field.

The starting index is taken from the INDEXES parameter. If that is null, the search begins with the default starting Index (see K flag description above).

NOTE: If the first index passed in the INDEXES parameter is a compound index, the M flag is removed and only that one index is searched. See "Lookup Index" in the Details and Features section for more information.

O **O**nly find exact matches if possible. The Finder first searches for exact matches on the requested Index(es); if any are found, it returns all exact matches to the lookup value. Only if it finds none in the file does it search for partial matches, returning every partial match. For example, if the lookup value is "EINSTEIN" and the file contains entries "EINSTEIN" and "EINSTEIN,ALBERT", only the first record is returned. If the first record did not exist, the Finder would return "EINSTEIN,ALBERT" as a match. If FLAGS does not contain an O, the Finder returns all matches, partial and exact.

If the lookup is done on a compound index, exact matches must be made for every data value subscript in the index in order to consider the entry to be an exact match.

P **P**ack output. This flag changes the Finder's output format to pack the information returned for each record onto a single node per record. A MAP node is introduced to make it easier to locate different data elements in the output. See the information below in

the Output, the Details and Features, and the Examples sections for more information.

Q **Q**uick lookup. If this flag is passed, the Finder assumes the passed value is in internal format. The Finder performs NO transforms of the input value, but only tries to find the value in the specified lookup indexes. Therefore, when the Q flag is passed, the lookup is much more efficient. If the FLAGS parameter does not contain a Q, the Finder assumes the lookup value is an external or user-entered value and performs all normal transforms as documented below.

U **U**nscreened lookup. This flag makes the Finder ignore any whole file screen (stored at ^DD(file#,0,"SCR")) on the file specified in the FILE parameter.

NOTE: Passing this flag does not make the Finder ignore the SCREEN parameter.

X E**X**act matches only. The Finder returns every exact match to the lookup value on the requested Index(es). Any partial matches present in the file are ignored, and transforms, such as changing the lookup value to uppercase, are not performed. For example, in the scenarios described under the O flag, the Finder behaves identically in the first situation, but under the second it returns no matches, since "EINSTEIN,ALBERT" is not an exact match to "EINSTEIN". If both the O and X flags are passed, the O flag is ignored. If the lookup is done on a compound index, exact matches must be made for every data value subscript in the index.

[.]VALUE (Required) The lookup value(s). These should be in external format as they would be entered by an end-user, unless the Q flag is used. Except for special lookup values listed below, the lookup value is matched to entries on the lookup INDEXES specified in the call. If the lookup index is compound, then lookup values can be provided for each of the data value subscripts in the index. In that case, VALUE is passed by reference as an array where VALUE(n) represents the lookup value to be matched to the **n**th subscript in the index. If only one lookup value is passed in VALUE, it is assumed to apply to the first data value subscript in the index.

In addition, certain values generate special behavior by the Finder as follows:

1. **Control characters.** This value always results in no matches. Control characters are not permitted in the database.
2. **^ (Up-arrow [shift-6]).** This value always results in no matches. This single character value signifies to VA FileMan that the current activity should be stopped.
3. **"" (The empty string).** On single field indexes, this value always results in no matches. The empty string, used by VA FileMan to designate fields that have no value, cannot be found in FileMan indexes. However, if the lookup uses a compound index, VALUE(n) can be null for any of the lookup values as long as at least one of them is non-null. If VALUE(1) is null, it may make the lookup slower. If VALUE(n) is null, all non-null values for that subscript position will be returned.
4. **" " (The space character).** This value indicates that the Finder should return the current user's previous selection from this file. This corresponds to the "space-bar-recall" feature of FileMan's user interface. If VA FileMan has no such previous selection for this user, or if this selection is now prohibited from selection somehow (see discussion of SCREEN, below), then the Finder returns no matches. The Finder itself never preserves its found values for this recall; applications wishing to preserve found values should call RECALL^DILFD. The special lookup characters should appear either in VALUE or in VALUE(1).
5. **"`"-Number (accent-grave followed by a number).** This indicates that the Finder should select the entry whose internal entry number equals the number following the accent-grave character. This corresponds to an equivalent feature of FileMan's user interface. If this entry is prohibited from selection, the Finder returns no match. The use of '-number input does not require passing A in the FLAGS parameter. The special lookup characters should appear either in VALUE or in VALUE(1).

6. **Numbers.** The Finder tries strictly numeric input as an IEN under any of the following four conditions: 1) The caller passes A in the FLAGS parameter, 2) the file has a .001 field, 3) the file's .01 field is not numeric and the file has no lookup index, or 4) The INDEXES parameter contains "#" as one of its index names. In all cases, the numeric lookup value is expected to be in either VALUE or VALUE(1). In condition 4, if the "#" is the only INDEX, and if the lookup value does not match an IEN, the lookup fails, otherwise, the Finder continues the search using the other indexes.

   In conditions 1, 2 and 3, strictly numeric input differs from `-numeric input in that whether or not a record corresponding to this IEN exists or is selectable, the Finder proceeds with a regular lookup, using the numeric value to find matches in the file's indexes. Even used this way, however, numeric input has the following special restriction: it is not used as a lookup value in any indexed pointer or variable pointer field (unless Q is passed in the FLAGS parameter).

   For example, suppose an application performs a Finder call on the EMPLOYEE file, passing a lookup value of 12; that the EMPLOYEE file points to the State file, in which Washington is record number 12; and that the EMPLOYEE file's pointer to the State file is indexed. The application would not be able to use the input value of 12 to find every employee who lives in Washington state.

NUMBER

(Optional) The maximum number of entries to find. If the Finder actually matches the input to this many entries, it breaks out of its search and returns what it has found so far. In such a situation, there is no way for the Finder to resume its search later where it left off. A value of "*" designates all entries.

Defaults to "*".

[.]INDEXES

(Optional) The indexes the Finder should search for matches. This parameter should be set to a list of index names separated by ^ characters. This parameter specifies both which indexes to check and the order in which to check

them. The caller does not need to pass the M flag for the INDEXES parameter to work properly. For example, a value of "B^C^ZZALBERT^D" specifies four indexes to check in the order shown.

If the first index passed is a compound index, only that one index can be in the list. Attempting to put more than one index in the list when the first one is compound will generate an error. If the first index in the list is a single subscript index, however, compound indexes can follow that one in the list. In that case, the lookup expects only one lookup value and only the first subscript of any compound index is checked for matches.

If no index name, or only one index name, is passed in the INDEXES parameter, and if the FLAGS parameter contains an M, then the Finder will do the search using the starting index, as well as all indexes that follow the starting one alphabetically (unless the starting index is compound—see paragraph above). See also the documentation on the M flag.

If the index is not specified, the default starting index will be "B" unless the FLAGS parameter contains a K, in which case the default will be the Uniqueness Index defined for the Primary Key on the file.

Mnemonic cross-references folded into the specified index are included in the output.

When the first subscript of one of the indexes on the file you are searching indexes a pointer or variable pointer, then the Finder searches the pointed-to file for matches to the lookup value. Array entries can be passed in the INDEXES parameter to control this search on the pointed-to file. Suppose the name of the array is NMSPIX. Then you can set NMSPIX("PTRIX",from_file#,pointer_field#,to_file#)= "^"_delimited_index_list. This array entry allows the user to pass a list of indexes that will be used when doing the search on the pointed-to file.

For example, if your file (662001) has a pointer field (5) to file 200 (NEW PERSON), and you wanted the lookup on field 5 to find entries in the NEW PERSON file only by name ("B" index), or by the first letter of the last name

concatenated with the last 4 digits of the social security number ("BS5" index), set NMSPIX("PTRIX",662001,5,200)="B^BS5".

[.]SCREEN

(Optional) **Entry Screen.** The screen to apply to each potential entry in the returned list to decide whether or not to include it. This may be set to any valid M code that sets $TEST to 1 if the entry should be included, to 0 if not. This is exactly equivalent to the DIC("S") input variable for the Classic FileMan lookup ^DIC. The Finder will execute this screen in addition to any SCR node (whole-file screen) defined on the data dictionary for the file. Optionally, the screen can be defined in an array entry subscripted by "S" (for example, SCR("S")), allowing additional screen entries to be defined for variable pointer fields as described below.

The entry screen code can rely upon the following:

Naked indicator	Zero-node of entry's record.
D	Index being traversed.
DIC	Open global reference of file being traversed.
DIC(0)	Flags passed to the Finder.
Y	Record number of entry under consideration.
Y() array	For subfiles, descendents give record numbers for all upper levels. Structure resembles the DA array as used in a call to the classic FileMan edit routine ^DIE.
Y1	IENS equivalent to Y array.

The code can also safely change any of these values.

For example, "I Y<100" ensures that only records with an internal entry number less than 100 are accepted as matches. See Details and Features in this section for an explanation of the other conditions and screens involved in finding an entry. Defaults to adding no extra conditions to those listed in that section.

Variable Pointer Screen. If one of the fields indexed by the cross-reference passed in the INDEXES parameter is a variable pointer, then additional screens equivalent to the DIC("V") input variable to Classic FileMan lookup ^DIC can also be passed. Suppose the screens are being passed in the SCR array. Then for a simple index with just one data value field, the code can be passed in SCR("V"). For simple or compound indexes, screens can be passed for any indexed fields that are variable pointers in the format SCR("V",n) where "n" represents the subscript location of the variable pointer field on the index.

The Variable Pointer screen restricts the users ability to see entries on one or more of the files pointed-to by the variable pointer. The screen logic is set equal to a line of M code that will return a truth value when executed. If it evaluates TRUE, then entries that point to the file can be included in the output; if FALSE, any entry pointing to the file is excluded. At the time the code is executed, the variable Y(0) is set equal to the information for that file from the data dictionary definition of the variable pointer field. You can use Y(0) in the code set into the variable pointer screen parameter. Y(0) contains:

^-Piece	**Contents**
Piece 1	File number of the pointed-to file.
Piece 2	Message defined for the pointed-to file.
Piece 3	Order defined for the pointed-to file.
Piece 4	Prefix defined for the pointed-to file.
Piece 5	y/n indicating if a screen is set up for the pointed-to file.
Piece 6	y/n indicating if the user can add new entries to the pointed-to file.

All of this information was defined when that file was entered as one of the possibilities for the variable pointer field.

For example, suppose your .01 field is a variable pointer pointing to files 1000, 2000, and 3000. If you only want the user to be able to enter values from files 1000 or 3000, you could set up SCR("V") like this:

```
S SCR("V")="I +Y(0)=1000!(+Y(0)=3000)"
```

IDENTIFIER (Optional) The text to accompany each found entry to help identify it to the end user. This should be set to M code that calls the EN^DDIOL utility to load identification text. The identification text generated by this parameter is listed AFTER that generated by any WRITE identifiers on the file itself. The code should not issue WRITE commands.

For example, a value of "D 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.

This code relies upon all of the same input as the SCREEN parameter described above and can safely change the same things. Defaults to no code.

TARGET_ROOT (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 the TARGET_ROOT is not passed, the list is returned descendent from ^TMP("DILIST",$J).

MSG_ROOT (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

TARGET_ROOT The examples in this section assume that the output from the Finder was returned in the default location descendent from ^TMP("DILIST",$J), but it could just as well be in an

array specified by the caller in the TARGET_ROOT parameter described above.

There are two different formats possible for the output, (1) Standard output format, and (2) Packed output (format returned when the P flag is included in the FLAGS parameter).

1. Standard Output Format

The format of the Output List is:

- **Header Node**

  Unless the Finder has run into an error condition, it will always return a header node for its output list, even if the list is empty, because no matches were found. The header node, on the zero node of the output array, has this format:

  ```
  ^TMP("DILIST",$J,0) = # of entries found ^ maximum
  requested ^any more? ^ results flags
  ```

  1. The # of entries found will be equal to or less than the maximum requested.

  2. The maximum requested should equal the NUMBER parameter, or, if NUMBER was not passed, "*".

  3. The any more? value is 1 if there are more matching entries in the file than were returned in this list, or 0 if not.

  4. The results flag at present is usually empty. If the output was packed and some of the data contained embedded "^" characters, the results flag contains the H flag. In the future the Finder may return other flags as well in this piece, so check whether it contains H, not whether it equals it. For more information see Details and Features.

- **Record Data**

  Standard output for the Finder returns its output with each field of each matching record on a separate node. Records are subscripted in this array by arbitrary

sequence number that reflects the order in which the record was found.

- .01 Field

  Unless suppressed with the "@" in the FIELDS parameter (the suggested practice), the .01 field of each record is returned under the 1 subtree of the array, in internal format.

  ```
  ^TMP("DILIST",$J,1,seq#) = .01_field_value_in_
  internal_format
  ```

  NOTE: This is different from the Lister, which returns the indexed field values in the 1 subtree.

- IEN

  Each record's IEN is returned under the 2 subtree:

  ```
  ^TMP("DILIST",$J,2,seq#) = IEN
  ```

The other values returned for each record are grouped together under the "ID" subtree, then by record.

- Field Values or Field Identifiers

  The output format is the same whether the field value is one of the Field Identifiers from the data dictionary for the file or the field was requested in the FIELDS parameter.

  Field identifiers and field values are subscripted by their field numbers. Each node shows up as:

  ```
  ^TMP("DILIST",$J,"ID",seq#,field #) = field_value
  ```

  If both the "I" and "E" suffix are specified, an additional subscript level with the values of "E" and "I" is used to distinguish the external and internal values of the field. If a field is only returned in one format, the extra subscript is never included. Values output with the extra format specifier look like:

  ```
  ^TMP("DILIST",$J,"ID",seq#,field#,"E" or "I")
  = field_value
  ```

- Output for field specifier "IX" in FIELDS

  A field specifier of "IX" in the FIELDS parameter retrieves the value of the indexed field(s). In the output, the values of these fields are returned as follows, where the final subscript is a sequential number indicating the subscript location in the index.

  ```
  ^TMP("DILIST",$J,"ID",seq#,0,1) = first_
  subscript_index_value

  ^TMP("DILIST",$J,"ID",seq#,0,2) = second_
  subscript_index_value
  ```

  If both the "I" and "E" suffix are specified, an additional subscript level with the values of "E" and "I" is used to distinguish the external and internal values from the index. If the subscript on the index is not derived from a field, i.e. if it's a computed subscript, then the internal and external value both will be the same, the value directly from the index.

- WRITE Identifiers

  WRITE (display-only) identifiers are grouped under the "WRITE" subtree of the "ID" tree, then by record number. It is the caller's responsibility to ensure that none of the WRITE identifiers issue direct READ or WRITE commands and that they issue any output through EN^DDIOL so it can be collected by the Finder. The output from all the WRITE identifiers for a single record is listed as individual lines of text:

  ```
  ^TMP("DILIST",$J,"ID","WRITE",seq#,line #) = text
                          generated by WRITE IDs
  ```

- IDENTIFIER parameter

  Any text generated by the caller's IDENTIFIER parameter is returned in the last lines of the WRITE identifier text.

- **Map Node for Unpacked Format**

  In order to facilitate finding information in the output,

a Map Node is built for unpacked format. This node is returned in ^TMP("DILIST",$J,0,"MAP").

The Map node for unpacked format describes Field Identifier data in the "ID" output data nodes. It contains "^"delimited pieces described below. The position of the piece in the map node corresponds to the order in which it can be found in the "ID" output nodes. If the data is returned in internal format, the piece will be followed by "I" (ex., "2I" means that the internal value of field 2 was returned in the output).

- **#:** Individually requested field number, where # is the field number, for each field requested in the FIELDS parameter
- **FID(#):** Field Identifier, where # is the field number.

2. Packed Output Format

If the P flag is used to request packed output, the Finder packs all the return values into one output node per record. You must ensure that all requested data will fit onto a single node. Overflow causes error 206. Return values containing embedded "^" characters make the Finder encode the output data using HTML encoding (described in Details and Features).

- **Header Node—Identical to Standard Output Format**
- **Record Data**

  Values in the output are delimited by "^" characters. Piece 1 is always the IEN. The values of other pieces depend on the value of the FIELDS parameter. If the FIELDS parameter is not passed, each record's packed node will follow this format:

  ```
  ^TMP("DILIST",$J,seq#,0)=IEN^Internal_.01_field_
  value^field_Identifiers^Write_Identifiers^Output_
  from_Identifier_parameter
  ```

  Field Identifiers are sequenced by field number. Output values specified by the FIELDS parameter are packed in the order in which they occur in the FIELDS

parameter. WRITE identifiers are packed in the same order as their subscripts occur in the ID subtree of the file's data dictionary.

To parse the output of the packed nodes, use the MAP node described below.

- **Map Node for Packed Format**

  Because the packed format is not self-documenting and because individual field specifiers such as FID can correspond to a variable number of field values, the Finder always includes a map node when returning output in Packed format. This node is returned in ^TMP("DILIST",$J,0,"MAP").

  Its value resembles a data node's value in that it has the same number of ^-pieces, but the value of each piece identifies the field or value used to populate the equivalent location in the data nodes. The possible values for each piece in the map node are:

  - **IEN:** the IEN
  - **01:** the .01 field
  - **FID(#):** Field identifier, where # is the field number of the identifier
  - **WID(string):** WRITE identifier, where string is the value of the subscript in the ^DD where the identifier is stored (such as "WRITE")
  - **IDP:** Identifier parameter
  - **IX(n):** Indexed field values, where "n" refers to the subscript position in the index.
  - #: Individually requested field, by field number

NOTE: For any piece except IEN, WID or IDP, if the internal value is to be returned, the piece will be followed by "I". Thus instead of IX(1), you would see IX(1)I, indicating that the internal index value was being returned.

For example, the map node for a Finder call on the Option

file, if FIELDS => "3.6I;3.6;4", might look like this:

```
^TMP("DILIST",$J,0,"MAP") = "IEN^.01^3.6I^3.6^4"
```

Examples

Example 1

First we 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).

```
>D FIND^DIC(19,"","","","STAT","","C","","","OUT")

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

Example 2

Here we look on the OPTION file for entries that are at least partial matches to "DIS". We use the "B" index and, since we don't include the M flag to search multiple indexes, we look ONLY on the "B" index. We use the "@" in the FIELDS parameter to suppress the default values and specify that we want the .01 field NAME, field 1 DESCRIPTION, and the index values in the output.

```
>D FIND^DIC(19,"","@;.01;1;IX","","DIS",5,"B","","","OUT")

OUT("DILIST",0)=2^5^0^
OUT("DILIST",0,"MAP")=IX(1)^.01^1
OUT("DILIST",2,1)=11
OUT("DILIST",2,2)=15
OUT("DILIST","ID",1,0,1)=DISEARCH
OUT("DILIST","ID",1,.01)=DISEARCH
OUT("DILIST","ID",1,1)=Search File Entries
OUT("DILIST","ID",2,0,1)=DISTATISTICS
OUT("DILIST","ID",2,.01)=DISTATISTICS
OUT("DILIST","ID",2,1)=Statistics
```

Example 3

Next, we do a call almost identical to Example 2, but this time we use the M flag to indicate that we want to search all the lookup indexes starting from "B". This time we get more records back and looking at the index values in the entries OUT("DILIST","ID",seq#,0,subscript_location), we see that the new entries were found on an index other than the "B" index (since the values don't match the .01 field). In fact, they were found on the index for the field UPPER CASE MENU TEXT (index "C" on the file).

```
>D FIND^DIC(19,"","@;.01;1;IX","M","DIS",5,"B","","","OUT")

OUT("DILIST",0)=5^5^1^
OUT("DILIST",0,"MAP")=IX(1)^.01^1
OUT("DILIST",2,1)=11
OUT("DILIST",2,2)=15
OUT("DILIST",2,3)=468
OUT("DILIST",2,4)=470
OUT("DILIST",2,5)=469
OUT("DILIST","ID",1,0,1)=DISEARCH
OUT("DILIST","ID",1,.01)=DISEARCH
OUT("DILIST","ID",1,1)=Search File Entries
OUT("DILIST","ID",2,0,1)=DISTATISTICS
OUT("DILIST","ID",2,.01)=DISTATISTICS
OUT("DILIST","ID",2,1)=Statistics
OUT("DILIST","ID",3,0,1)=DISK DRIVE RAW DATA STATISTICS
OUT("DILIST","ID",3,.01)=XUCM DISK
OUT("DILIST","ID",3,1)=Disk Drive Raw Data Statistics
OUT("DILIST","ID",4,0,1)=DISK DRIVE REQUEST QUEUE LENGT
OUT("DILIST","ID",4,.01)=XUCM DSK QUE
OUT("DILIST","ID",4,1)=Disk Drive Request Queue Length
OUT("DILIST","ID",5,0,1)=DISK I/O OPERATION RATE
OUT("DILIST","ID",5,.01)=XUCM DSK IO
OUT("DILIST","ID",5,1)=Disk I/O Operation Rate
```

Example 4

In this example, we'll use the K flag to do a lookup on a file with a Primary Key made up of the .01 field (NAME) and field 1 (DATE OF BIRTH). We'll suppress all of the output with "@" and then ask only for both the internal and external index values. Notice that the P flag causes the output to be returned in Packed format. The MAP node tells us what is in each "^" piece of the output.

```
>K VAL S VAL(1)="ADD",VAL(2)="01/01/69"
>D FIND^DIC(662001,"","@;IXIE","PK",.VAL,"","","","","OUT")

OUT("DILIST",0)=1^*^0^
OUT("DILIST",0,"MAP")=IEN^IX(1)I^IX(2)I^IX(1)^IX(2)
OUT("DILIST",1,0)=15^ADDFIFTEEN^2690101^ADDFIFTEEN^JAN 01, 1969
```

Example 5

Here we'll demonstrate how the B flag works. We have a file whose .01 field points to the NEW PERSON file. When we do a lookup without the B flag, we find several entries, but if you look at the .01 field, you see that not all of them begin with our lookup value "M". The entry "CALIFORNIA,MR" was found because his initials "MC" begin with "M" and "WYNNETTE,TAMMY" was found because her nickname "MILLY" begins with "M".

```
>D FIND^DIC(662002,"","@;.01","P","M","","B","","","OUT")

OUT("DILIST",0)=5^*^0^
OUT("DILIST",0,"MAP")=IEN^.01
OUT("DILIST",1,0)=7^CALIFORNIA,MR
OUT("DILIST",2,0)=3^MANN,MANFRED
OUT("DILIST",3,0)=4^MITTY,WALTER
OUT("DILIST",5,0)=1^WYNETTE,TAMMY
OUT("DILIST",6,0)=13^WYNETTE,TAMMY
```

When we use the B flag, the FINDER looks ONLY at the "B" index of the NEW PERSON file.

```
>D FIND^DIC(662002,"","@;.01","PB","M","","B","","","OUT")

>ZW OUT
OUT("DILIST",0)=2^*^0^
OUT("DILIST",0,"MAP")=IEN^.01
OUT("DILIST",1,0)=3^MANN,MANFRED
OUT("DILIST",2,0)=4^MITTY,WALTER
```

Example 6

First we make a call without the new parameter, using a lookup value of "T". There are indexes on both the NICKNAME and the INITIALS field. Because we didn't specify which indexes to use, FileMan uses all lookup indexes during the lookup on the pointed-to file. In this call, we pick up several entries. The NICKNAME for Roger Marsh happens to be "TOAD", and the INITIALS field for Tammy Wynette is "TW".

```
S INDEX="B^C^E"

D FIND^DIC(662002,,".01;IXIE;@","PM","T",,INDEX,,,"TKW")

>ZW TKW
TKW("DILIST",0)=4^*^0^
TKW("DILIST",0,"MAP")=IEN^.01^IX(1)I^IX(1)
TKW("DILIST",1,0)=4^MARSH,ROGER^9^MARSH,ROGER
TKW("DILIST",2,0)=12^TIMOTHY,GREG^12^TIMOTHY,GREG
TKW("DILIST",3,0)=1^WYNETTE,TAMMY^4^WYNETTE,TAMMY
```

```
TKW("DILIST",4,0)=13^WYNETTE,TAMMY^4^WYNETTE,TAMMY
```

This time, we set the new parameter so that we only look at the "B" and BS5 indexes on the pointed-to file. This time we do not find any entries whose INITIALS or NICKNAME field start with "T". We just pick up the person whose last name starts with "T".

```
>S INDEX("PTRIX",662002,.01,200)="B^BS5"

>D FIND^DIC(662002,,".01;IXIE;@","PM","T",,.INDEX,,,"TKW")

>ZW TKW
TKW("DILIST",0)=1^*^0^
TKW("DILIST",0,"MAP")=IEN^.01^IX(1)I^IX(1)
TKW("DILIST",1,0)=12^TIMOTHY, GREG ^12^TIMOTHY,GREG
```

Error Codes Returned

120 Error occurred during execution of a FileMan hook.

202 An input parameter is missing or not valid.

204 The input value contains control characters.

205 The File and IENS represent different subfile levels.

206 The data requested for the record is too long to pack together.

207 The value is too long to encode into HTML.

301 The passed flags are unknown or inconsistent.

304 The IENS lacks a final comma.

306 The first comma-piece of the IENS should be empty.

401 The file does not exist.

402 The global root is missing or not valid.

406 The file has no .01 field definition.

407 A word-processing field is not a file.

420 The index is missing.

501 The file does not contain that field.

520 That kind of field cannot be processed by this utility.

8090 Pre-lookup transform (7.5 node).

8095 First lookup index is compound, so "M"ultiple index lookups not allowed.

The Finder may also return any error returned by $$EXTERNAL^DILFD.

Details and Features

Lookup Index

If the "Use" flag for an index entry in the new INDEX file is set to "L" for Lookup, the index name must be "B" or must alphabetically follow "B". Also, traditional indexes whose names follow "B" are considered to be Lookup type indexes.

What does this mean? For a Finder call (FIND^DIC or $$FIND1^DIC), it means that if M is passed in the FLAGS parameter and a list of indexes is not specified in the INDEXES parameter, then FileMan will automatically use any lookup type index it finds by ordering through the index name alphabetically, starting with the beginning index ("B", unless a different one is specified in the input parameters). Any index, however, can be used for lookup if it is specified in the INDEXES parameter. The developer should be careful to make sure the MUMPS-type indexes are formatted similar to VA FileMan regular indexes, with the data subscripts followed by the IEN at the level of the file/subfile passed in the FILE input parameter.

Screens Applied

Valid Entry Conditions. To be considered for selection, an entry must have a properly formatted index to get the Finder's attention and a defined zero-node with a non-null first piece.

File Pre-Lookup Action (7.5 Node). Prior to performing any search of the database whatsoever, the Finder executes the 7.5 Node for the file. This code may alter the variable X, the lookup value, to alter the value used by the Finder in its search.

NOTE: The 7.5 node only works on a simple index, not a compound one. It assumes just one lookup value X.

Call Pre-Selection Action. The SCREEN parameter is executed once a potential match has been identified (as described under the Input Parameters section).

File Pre-Selection Action. If the file has a pre-selection action defined (the SCR node), then after passing the pre-selection action for the call, the entry must also pass the action for the whole file.

Partial Matches

For most values on most indexes, an input value partially matches an entry if the index value begins with the input value (e.g., index value of "EINSTEIN,ALBERT" partially matches input value of "EINSTEIN"). The exception is numeric input. On a numeric field's index, a numeric input must match exactly.

If the lookup value is numeric but the cross-referenced field is free-text, the Finder will find all partial matches to the numeric lookup value. For example, lookup value 1 matches to 1, 199, 1000.23 and 1ABC.

Space Bar Recall

Although the Finder honors the space bar recall feature whenever passed the input value " ", selections made through the Finder are not stored for later use by space bar recall because the Finder has no way of knowing whether the selection results from interaction with the user. Only deliberate user selections should affect the space bar recall value. As a result, to support this feature, applications should call RECALL^DILFD when managing the user interface whenever the user makes a selection.

Lookup Value Transforms List

The original lookup value(s) passed to the Finder are not the only values used during the lookup. Certain transforms are done on the original lookup value and matches are made for these transformed values along with the original ones. The Q flag suppresses all of these transforms and looks on the index(es) only for the original lookup value. See "Upper Case", "Long Input", "Comma-Piecing" and "Data Type Transforms" immediately below.

Upper Case

The first basic transform ensures that lookups succeed when users leave their Caps Lock keys off. If the VALUE parameter contains any lower case characters, the Finder will also look for an all-upper-case version of the value.

Long Input

The second basic transform ensures that lookups work properly when lookup and field values are longer than the maximum length of a data-values subscript in the index. (This is 30 characters for traditional indexes, but is set by the developer for indexes defined in the new INDEX file).

Comma-piecing

The third and final basic transform provides a special feature of VA FileMan's lookup. This feature, known as comma-piecing, helps the user enter fewer characters to distinguish between similar entries. FileMan uses lookup values that contain embedded commas to build a pattern match based on all the comma-pieces. For example, distinguishing between "KENNEDY,ROBERT FRANCIS" and "KENNEDY,JOHN FITZGERALD" would normally take nine keystrokes-"KENNEDY,J"-but comma-piecing lets the user do it in three: "K,J".

Although commas are used to trigger the comma-piecing feature, the characters used to break up the entry in the file can be any kind of punctuation, not only commas. For example, "T,R" matches "THE ROAD LESS TRAVELED".

If the new C flag is used in the FLAGS parameter, then the second comma piece of the lookup value can be a match to any of the pieces in the file entry following the first one. So, for example, "B,S" distinguishes "BACH,JOHANN SEBASTIAN" from his sons "BACH,JOHANN CHRISTIAN" and "BACH,JOHANN CHRISTOPH FRIEDRICH".

Data Type Transforms

Indexes store the internal format of fields values, but users typically enter the external format as lookup values. Therefore, the Finder attempts to do conversions of the lookup values when it searches an index on a Date, Set of Codes, Pointer or Variable Pointer field.

For example, a lookup value of "t" would also be evaluated as today's date in internal FileMan format, if the Finder is searching the index on a date type field, since VA FileMan normally recognizes a user entry of "T" at a date prompt as meaning "TODAY".

If a Q flag is passed in the FLAGS parameter, no data type transforms are attempted.

NOTE: The data type transform for indexes on pointer and

variable pointer fields involves a complete lookup on the pointed-to file. For example, if an application calls the Finder with the input value "W" on a file with an indexed pointer to the State file, the Finder locates every state starting with W (Washington, West Virginia, Wisconsin and Wyoming). It will return every record in the pointing file that points to one of those states.

Also, if the pointed-to file has indexed pointers or variable pointers, the search continues to these pointed-to files.

Therefore, to make more efficient searches, and to find just the entries desired, applications should make use of all available features of the Finder to narrow down the search. For example, use the INDEXES parameter when appropriate to limit the list of indexes searched, and the B flag when appropriate to make sure that only the "B" index is searched on any pointed-to file.

HTML Encoding

Since the Finder uses the "^" character as its delimiter for Packed output, it cannot let any of the data contain that character. If any does, it will encode all of the data using an HTML encoding scheme.

In this scheme, all "&" characters are replaced with the substring "&" and all "^" characters with the string "&#94". This keeps the data properly parsable and decodable. The data for all records found, not just the ones with embedded "^"s, will be encoded if embedded "^"s are found in the data of any of the records.

If the Finder has encoded the output, it will include an H flag in ^-piece four of the output header node.

Data can be decoded using the VA FileMan library function call $$HTML^DILF(encoded string,-1). It can properly decode individual fields or complete packed data nodes.

WRITE ID Nodes

The Finder executes each individual WRITE ID node from the data dictionary. If an individual node results in creating multiple lines in the output from the EN^DDIOL call(s) it contains, then in Standard Output Format the results will

appear on multiple lines in the output array. Thus, there is not a direct correlation between the number of WRITE ID nodes and the number of nodes that will be returned in the output array of a Finder call for each record. In Packed output format, each WRITE ID node appears in a separate "^" piece, and line feeds are designated with a tilde "~" character.

Repeating a Field in FIELDS parameter

If a field is listed multiple times in the FIELDS parameter, it is returned multiple times in Packed output, but only once in unpacked output. This is because the field number is one of the subscripts of unpacked output. The exception is when the occurrences are for different formats, internal and external.

$$FIND1^DIC( ): Finder (Single Record)

This extrinsic function finds a single record in a file based on input value(s). If more than one match is found, the function returns an error. The caller must specify a file number and the input value(s) to be used for the lookup. The caller can also specify the index(es) to be used in the search, and can also pass screening logic.

NOTE: $$FIND1 does NOT honor the Special Lookup or Post-Lookup Action nodes defined in the data dictionary for a file.

Format

```
$$FIND1^DIC(FILE,IENS,FLAGS,[.]VALUE,[.]INDEXES,[.]SCREEN,MSG_ROOT)
```

Input Parameters

FILE (Required) The number of the file or subfile to search. If this parameter is a subfile, it must be accompanied by the IENS parameter.

IENS (Optional) The IENS that identifies the subfile, if FILE is a subfile number. To identify a subfile, rather than a subfile entry, leave the first comma-piece empty. For example, a value of ",67," indicates that the subfile within entry #67 should be used. If FILE is a file number, this parameter should be empty. Defaults to no subfile.

FLAGS (Optional) Flags to control processing. This parameter lets the caller adjust the Finder's algorithm. The possible values are:

A **A**llow pure numeric input to always be tried as an IEN. Normally, the Finder will only try pure numbers as IENs if: 1) The file has a .001 field, or 2) its .01 field is not numeric and the file has no lookup index.

When this flag is used, records that match other numeric interpretations of the input will be found in addition to a record with a matching IEN. For example, a lookup value of "2" would

match a record with a lookup field of "2JOHN" as well as a record with an IEN of 2.

NOTE: If the numeric lookup value is preceded by an accent grave character (`), lookup interprets the input as an IEN, and only attempts to match by IEN. The A flag is not required in this case.

B **B** index used on lookups to pointed-to files. Without the B flag, if there are cross-referenced pointer fields in the list of indexes to use for lookup then: (1.) for each cross-referenced pointer field, FileMan checks ALL lookup indexes in each pointed-to file for a match to X (time-consuming), and (2.) if X matches any value in any lookup index (not just on the .01 field) in a pointed-to file and the IEN of the matched entry is in the home file's pointer field cross-reference, FileMan considers this a match (perhaps not the lookup behavior desired).

The B flag prevents this behavior by looking for a match to X only in the "B" index (.01 field) of files pointed to by cross-referenced pointer fields. This makes lookups quicker and avoids the risk of FileMan matching an entry in the pointed-to file based on something other than the .01 field.

See the Details and Features section for an explanation of the "Lookup Index" and the Examples section for more information on use of the B flag.

C Use the **C**lassic way of performing lookups on names, i.e., like the classic FileMan lookup routine ^DIC. If C is passed in the FLAGS parameter and, for example, the user enters a lookup value of "Smi,J", the Finder will find"Smith,John" but also "Smiley,Bob J." The Finder takes the first comma piece of the lookup value "Smi", and looks for partial matches to that. It then takes the second

comma piece of the lookup value "J" and looks for partial matches to "J" on the second or any other piece of the value on the entry being examined. It uses any punctuation or space for a delimiter.

The default, without passing C in the FLAGS parameter, will look for partial matches ONLY on the second piece, thus in our example, finding "Smith,John" but not "Smiley,Bob J.". The old style of comma-piece processing can be quite slow, especially with common names like "Smith".

K Primary **K**ey used for starting index. If no index is specified in the INDEXES parameter, this flag causes the Finder to use the Uniqueness index for the Primary Key as the starting index for the search. Without the K flag, or if there is no Primary Key for this file (in the KEY file), the Finder defaults to the "B" index.

M **M**ultiple index lookup allowed. If more than one index is passed in the INDEXES parameter, all indexes in the list are searched. Otherwise, the M flag causes the Finder to search the starting index and all indexes that alphabetically follow it. This includes both indexes from the traditional location in the data dictionary, as well as lookup indexes defined on the INDEX file that have an "L" (for LOOKUP) in the new "Use" field.

The starting index is taken from the INDEXES parameter. If that is null, the search begins with the default starting Index (see K flag description above).

NOTE: If the first index passed in the INDEXES parameter is a compound index, the M flag is removed and only that one index is searched. See "Lookup Index" in the Details and Features section for more information.

O **O**nly find an exact match if possible. The Finder first searches for an exact match on the requested Index(es); if one is found, it is returned. Only if it does not find one in the file does it search for a partial match. For example, if the lookup value is "EINSTEIN" and the file contains entries "EINSTEIN" and "EINSTEIN,ALBERT", only the first record is returned. If the first record did not exist, the Finder would return "EINSTEIN,ALBERT" as a match.

NOTE: The presence of a partial match does not constitute an error condition, because a single exact match is present. If the FLAGS parameter does not contain O (or an X, see below), the presence of both partial and exact matches is treated as an error condition.

If the lookup is done on a compound index, exact matches must be made for every data value subscript in the index in order to consider the entry to be an exact match.

Q **Q**uick lookup. If this flag is passed, the Finder assumes the passed value is in internal format. The Finder performs NO transforms of the input value, but only tries to find the value in the specified lookup indexes. Therefore, when the Q flag is passed, the lookup is much more efficient. If the FLAGS parameter does not contain a Q, the Finder assumes the lookup value is an external or user-entered value and performs all normal transforms as documented below.

U **U**nscreened lookup. This flag makes the Finder ignore any whole file screen (stored at ^DD(file#,0,"SCR")) on the file specified in the FILE parameter. **NOTE:** Passing this flag does not make the Finder ignore the SCREEN parameter.

X E**X**act match only. The Finder returns only an exact match to the lookup value on the requested Index(es). Any partial matches present in the file are ignored, and transforms, such as changing the lookup value to uppercase, are not performed. For example, in the scenarios described under the O flag, the Finder behaves identically in the first situation, but under the second it returns no match, since "EINSTEIN, ALBERT" is not an exact match to "EINSTEIN". If both the O and X flags are passed, the O flag is ignored. If the lookup is done on a compound index, exact matches must be made for every data value subscript in the index.

[.]VALUE (Required) The lookup value(s). These should be in external format as they would be entered by an end-user, unless the Q flag is used. Except for special lookup values listed below, the lookup value is matched to entries on the lookup INDEXES specified in the call. If the lookup index is compound, then lookup values can be provided for each of the data value subscripts in the index. In that case, VALUE is passed by reference as an array where VALUE(n) represents the lookup value to be matched to the nth subscript in the index. If only one lookup value is passed in VALUE, it is assumed to apply to the first data value subscript in the index.

In addition, certain values generate special behavior by the Finder as follows:

1. **Control characters.** This value always results in no matches. Control characters are not permitted in the database.
2. **^ (Up-arrow [shift-6]).** This value always results in no matches. This single character value signifies to VA FileMan that the current activity should be stopped.
3. **"" (The empty string).** On single field indexes, this value always results in no matches. The empty string, used by VA FileMan to designate fields that have no value, cannot be found in FileMan indexes. However, if the lookup uses a compound index, VALUE(n) can be null for any of the lookup values as long as at least one of them is non-null. If VALUE(1) is null, it may make the lookup slower. If VALUE(n) is null, all non-null values for that subscript position will be returned.

4. " " **(The space character).** This value indicates that the Finder should return the current user's previous selection from this file. This corresponds to the "space-bar-recall" feature of VA FileMan's user interface. If FileMan has no such previous selection for this user, or if this selection is now prohibited from selection somehow (see discussions of SCREEN, below), then the Finder returns no matches. The Finder itself never preserves its found values for this recall; applications wishing to preserve found values should call RECALL^DILFD. The special lookup characters should appear either in VALUE or in VALUE(1).

5. "`"**-Number (accent-grave followed by a number).** This indicates that the Finder should select the entry whose internal entry number equals the number following the accent-grave character. This corresponds to an equivalent feature of FileMan's user interface. If this entry is prohibited from selection, the Finder returns no match. The use of '-number input does not require passing A in the FLAGS parameter. The special lookup characters should appear either in VALUE or in VALUE(1).

6. **Numbers.** The Finder tries strictly numeric input as an IEN under any of the following four conditions: 1) The caller passes A in the FLAGS parameter, 2) the file has a .001 field, 3) the file's .01 field is not numeric and the file has no lookup index, or 4) the INDEXES parameter contains "#" as one of its index names. In all cases, the lookup value is expected to be in either VALUE or VALUE(1). In condition 4, if the "#" is the only INDEX, and if the lookup value does not match an IEN, the lookup fails, otherwise, the Finder continues the search using the other indexes.

In conditions 1, 2 and 3, strictly numeric input differs from `-numeric input in that whether or not a record corresponding to this IEN exists or is selectable, the Finder proceeds with a regular lookup, using the numeric value to find matches in the file's indexes. Even used this way, however, numeric input has the following special restriction: it is not used as a lookup value in any indexed pointer or variable pointer field (unless Q is passed in the FLAGS parameter).

For example, suppose an application performs a Finder call on the EMPLOYEE file, passing a lookup value of 12; that

the EMPLOYEE file points to the State file, in which Washington is record number 12; and that the EMPLOYEE file's pointer to the State file is indexed. The application would not be able to use the input value of 12 to find every employee who lives in Washington state.

[.]INDEXES (Optional) The indexes the Finder should search for a match. This parameter should be set to a list of index names separated by ^ characters. This parameter specifies both which indexes to check and the order in which to check them. The caller does not need to pass the M flag for the INDEXES parameter to work properly. For example, a value of "B^C^ZZALBERT^D" specifies four indexes to check in the given order.

If the first index passed is a compound index, only that one index can be in the list. Attempting to put more than one index in the list when the first one is compound will generate an error. If the first index in the list is a single subscript index, however, compound indexes can follow that one in the list. In that case, the lookup expects only one lookup value and only the first subscript of any compound index is checked for matches.

If no index name, or only one index name, is passed in the INDEXES parameter, and if the FLAGS parameter contains an M, then the Finder will do the search using the starting index, as well as all indexes that follow the starting one alphabetically (unless the starting index is compound—see paragraph above). See also the documentation on the M flag.

If the index is not specified, the default starting index will be "B" unless the FLAGS parameter contains a K, in which case the default will be the Uniqueness Index defined for the Primary Key on the file.

Mnemonic cross-references folded into the specified index are included in the output.

When the first subscript of one of the indexes on the file you are searching indexes a pointer or variable pointer, then the Finder searches the pointed-to file for matches to the lookup value. Array entries can be passed in the INDEXES parameter to control this search on the pointed-to file. Suppose the name of the array is NMSPIX. Then you can set NMSPIX("PTRIX",from_file#,pointer_field#,to_file#)="^"_delimited_index_list. This array entry allows the user to pass a list of

indexes that will be used when doing the search on the pointed-to file.

For example, if your file (662001) has a pointer field (5) to file 200 (NEW PERSON), and you wanted the lookup on field 5 to find entries in the NEW PERSON file only by name ("B" index), or by the first letter of the last name concatenated with the last 4 digits of the social security number ("BS5" index), set NMSPIX("PTRIX",662001,5,200)="B^BS5".

[.]SCREEN (Optional) Entry Screen. The screen to apply to each potential entry in the returned list to decide whether or not to include it. This may be set to any valid M code that sets $TEST to 1 if the entry should be included, to 0 if not. This is exactly equivalent to the DIC("S") input variable for the Classic FileMan lookup ^DIC. The Finder will execute this screen in addition to any SCR node (whole-file screen) defined on the data dictionary for the file. Optionally, the screen can be defined in an array entry subscripted by "S" (for example, SCR("S")), allowing additional screen entries to be defined for variable pointer fields as described below.

The entry screen code can rely upon the following:

Naked indicator	Zero-node of entry's record.
D	Index being traversed.
DIC	Open global reference of file being traversed.
DIC(0)	Flags passed to the Finder.
Y	Record number of entry under consideration.
Y() array	For subfiles, descendents give record numbers for all upper levels. Structure resembles the DA array as used in a call to the classic FileMan edit routine ^DIE.
Y1	IENS equivalent to Y array.

The code can also safely change any of these values.

For example, "I Y>99" ensures that only a record numbered 100 or higher can be accepted as a match. See Details and Features

in this section for an explanation of the other conditions and screens involved in finding an entry. If duplicate entries exist, but only one passes the screens, then that one is returned and no error is generated. Defaults to adding no extra conditions to those listed in that section.

Variable Pointer Screen. If one of the fields indexed by the cross-reference passed in the INDEXES parameter is a variable pointer, then additional screens equivalent to the DIC("V") input variable for Classic FileMan lookup ^DIC can also be passed. Suppose the screens are being passed in the SCR array, then for a simple index with just one data value field, the code can be passed in SCR("V"). For simple or compound indexes, screens can be passed for any indexed fields that are variable pointers in the format SCR("V",n) where "n" represents the subscript location of the variable pointer field on the index.

The Variable Pointer screen restricts the user's ability to see entries on one or more of the files pointed to by the variable pointer. The screen logic is set equal to a line of M code that will return a truth value when executed. If it evaluates TRUE, then entries that point to the file can be included in the output; if FALSE, any entry pointing to the file is excluded. At the time the code is executed, the variable Y(0) is set equal to the information for that file from the data dictionary definition of the variable pointer field. You can use Y(0) in the code set into the variable pointer screen parameter. Y(0) contains:

^-Piece	**Contents**
Piece 1	File number of the pointed-to file.
Piece 2	Message defined for the pointed-to file.
Piece 3	Order defined for the pointed-to file.
Piece 4	Prefix defined for the pointed-to file.
Piece 5	y/n indicating if a screen is set up for the pointed-to file.
Piece 6	y/n indicating if the user can add new entries to the pointed-to file.

All of this information was defined when that file was entered as one of the possibilities for the variable pointer field.

For example, suppose your .01 field is a variable pointer pointing to files 1000, 2000, and 3000. If you only want the user to be able to enter values from files 1000 or 3000, you could set up SCR("V") like this:

```
S SCR("V")="I +Y(0)=1000!(+Y(0)=3000)"
```

MSG_ROOT (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

The function evaluates to an internal entry number (IEN) if a single match is found, 0 if no matches are found, or "" if an error occurred.

Examples

Example 1

Here we look for an option DIFG on the OPTION file. We use the M flag to search all indexes and the X flag to specify that we want exact matches only. It returns the IEN of the entry found.

```
>W $$FIND1^DIC(19,"","MX","DIFG","","","ERR")
327
```

Example 2

This time we look for an option that is not on the OPTION file. We set up the call exactly the same as Example 1. This time it returns 0 because no matching entry was found.

```
>W $$FIND1^DIC(19,"","MX","DIFG ZZZZ","","","ERR")
0
```

Example 3

Now we'll do the exact same call as in Example 1, but this time we won't include the X flag, so it will find not only "DIFG", but also any partial matches to "DIFG". Since there are several, it can't find just one match, so the call fails. The return is null and an error message is returned as well.

```
>W $$FIND1^DIC(19,"","M","DIFG","","","ERR")
DIERR=1^1

ERR("DIERR")=1^1
ERR("DIERR",1)=299
ERR("DIERR",1,"PARAM",0)=2
ERR("DIERR",1,"PARAM",1)=DIFG
ERR("DIERR",1,"PARAM","FILE")=19
ERR("DIERR",1,"TEXT",1)=More than one entry matches the value(s) 'DIFG'.
ERR("DIERR","E",299,1)=
```

Example 4

Now we'll do two different calls to find an entry on a test file. There are two entries whose .01 field equals "ADDFIFTEEN". In the first call, we'll do the lookup on the "B" index and the call fails because there are two entries that match the lookup value.

```
>W $$FIND1^DIC(662001,"","","ADDFIF","B","","ERR")

>ZW ERR
ERR("DIERR")=1^1
ERR("DIERR",1)=299
ERR("DIERR",1,"PARAM",0)=2
ERR("DIERR",1,"PARAM",1)=ADDFIF
ERR("DIERR",1,"PARAM","FILE")=662001
ERR("DIERR",1,"TEXT",1)=More than one entry matches the value(s) 'ADDFIF'.
ERR("DIERR","E",299,1)=
```

But if we try the call again and this time use the "BB" index for the file, which indexes the .01 field NAME and also field 1, DATE OF BIRTH, we can pass lookup values for both the fields, and the call is successful because we now have a single match. The two entries with the same .01 field have different values in their DATE OF BIRTH field.

```
>K VAL S VAL(1)="ADDFIF",VAL(2)="1/1/69"

>W $$FIND1^DIC(662001,"","",.VAL,"BB","","ERR")
15
```

Error Codes Returned

120 Error occurred during execution of a FileMan hook.

202 An input parameter is missing or not valid.

204 The input value contains control characters.

205 The File and IENS represent different subfile levels.

299 More than one entry matches that value.

301 The passed flags are unknown or inconsistent.

304 The IENS lacks a final comma.

306 The first comma-piece of the IENS should be empty.

401 The file does not exist.

402 The global root is missing or not valid.

406 The file has no .01 field definition.

407 A word-processing field is not a file.

420 The index is missing.

501 The file does not contain that field.

520 That kind of field cannot be processed by this utility.

8090 Pre-lookup transform (7.5 node).

8095 First lookup index is compound, so "M"ultiple index lookups not allowed.

The Finder may also return any error returned by $$EXTERNAL^DILFD.

Details and Features

The details and features of $$FIND1^DIC and FIND^DIC are the same ***except*** that FIND^DIC has three features ("HTML Encoding," "WRITE ID nodes," and "Repeating a field in FIELDS parameter") that $$FIND1^DIC does not have. The table below describes the details and features of $$FIND1^DIC.

Lookup Index

If the "Use" flag for an index entry in the new INDEX file is set to "L" for Lookup, the index name must be "B" or must alphabetically follow "B". Also, traditional indexes whose names follow "B" are considered to be Lookup type indexes.

What does this mean? For a Finder call (FIND^DIC or $$FIND1^DIC), it means that if M is passed in the FLAGS parameter and a list of indexes is not specified in the INDEXES parameter, then FileMan will automatically use any lookup type index it finds by ordering through the index name alphabetically, starting with the beginning index ("B", unless a different one is specified in the input parameters). Any index, however, can be used for lookup if it is specified in the INDEXES parameter. The developer should be careful to make sure the MUMPS-type indexes are formatted similar to VA FileMan regular indexes, with the data subscripts followed by the IEN at the level of the file/subfile passed in the FILE input parameter.

Screens Applied

Valid Entry Conditions. To be considered for selection, an entry must have a properly formatted index to get the Finder's attention and a defined zero-node with a non-null first piece.

File Pre-Lookup Action (7.5 Node). Prior to performing any search of the database whatsoever, the Finder executes the 7.5 Node for the file. This code may alter the variable X, the lookup value, to alter the value used by the Finder in its search.

NOTE: The 7.5 node only works on a simple index, not a compound one. It assumes just one lookup value X.

Call Pre-Selection Action. The SCREEN parameter is executed once a potential match has been identified (as described under the Input Parameters section).

File Pre-Selection Action. If the file has a pre-selection action defined (the SCR node), then after passing the pre-selection action for the call, the entry must also pass the action for the whole file.

Partial Matches

For most values on most indexes, an input value partially matches an entry if the index value begins with the input value (e.g., index value of "EINSTEIN,ALBERT" partially matches input value of "EINSTEIN"). The exception is numeric input. On a numeric field's index, a numeric input must match exactly.

If the lookup value is numeric but the cross-referenced field is free-text, the Finder will find all partial matches to the numeric lookup value. For example, lookup value 1 matches to 1, 199, 1000.23 and 1ABC.

Space Bar Recall

Although the Finder honors the space bar recall feature whenever passed the input value " ", selections made through the Finder are not stored for later use by space bar recall because the Finder has no way of knowing whether the selection results from interaction with the user. Only deliberate user selections should affect the space bar recall value. As a result, to support this feature, applications should call RECALL^DILFD when managing the user interface whenever the user makes a selection.

Lookup Value Transforms List

The original lookup value(s) passed to the Finder are not the only values used during the lookup. Certain transforms are done on the original lookup value and matches are made for these transformed values along with the original ones. The Q flag suppresses all of these transforms and looks on the index(es) only for the original lookup value. See "Upper Case", "Long Input", "Comma-Piecing" and "Data Type Transforms" immediately below.

Upper Case

The first basic transform ensures that lookups succeed when users leave their Caps Lock keys off. If the VALUE parameter contains any lower case characters, the Finder will also look for an all-upper-case version of the value.

Long Input

The second basic transform ensures that lookups work properly when lookup and field values are longer than the maximum length of a data-values subscript in the index. (This is 30 characters for traditional indexes, but is set by the developer for indexes defined in the new INDEX file).

Comma-piecing

The third and final basic transform provides a special feature of VA FileMan's lookup. This feature, known as comma-piecing, helps the user enter fewer characters to distinguish between similar entries. VA FileMan uses lookup values that contain embedded commas to build a pattern match based on all the comma-pieces. For example, distinguishing between "KENNEDY,ROBERT FRANCIS" and "KENNEDY,JOHN FITZGERALD" would normally take nine keystrokes-

"KENNEDY,J"-but comma-piecing lets the user do it in three: "K,J".

Although commas are used to trigger the comma-piecing feature, the characters used to break up the entry in the file can be any kind of punctuation, not only commas. For example, "T,R" matches "THE ROAD LESS TRAVELED".

If the new C flag is used in the FLAGS parameter, then the second comma piece of the lookup value can be a match to any of the pieces in the file entry following the first one. So, for example, "B,S" distinguishes "BACH,JOHANN SEBASTIAN" from his sons "BACH,JOHANN CHRISTIAN" and "BACH,JOHANN CHRISTOPH FRIEDRICH".

Data Type Transforms

Indexes store the internal format of fields values, but users typically enter the external format as lookup values. Therefore, the Finder attempts to do conversions of the lookup values when it searches an index on a Date, Set of Codes, Pointer or Variable Pointer field.

For example, a lookup value of "t" would also be evaluated as today's date in internal FileMan format, if the Finder is searching the index on a date type field, since VA FileMan normally recognizes a user entry of "T" at a date prompt as meaning "TODAY".

If a Q flag is passed in the FLAGS parameter, no data type transforms are attempted.

NOTE: The data type transform for indexes on pointer and variable pointer fields involves a complete lookup on the pointed-to file. For example, if an application calls the Finder with the input value "W" on a file with an indexed pointer to the State file, the Finder locates every state starting with W (Washington, West Virginia, Wisconsin and Wyoming). It will return every record in the pointing file that points to one of those states.

Also, if the pointed-to file has indexed pointers or variable pointers, the search continues to these pointed-to files.

Therefore, to make more efficient searches, and to find just the entries desired, applications should make use of all available features of the Finder to narrow down the search. For example,

use the INDEXES parameter when appropriate to limit the list of indexes searched, and the B flag when appropriate to make sure that only the "B" index is searched on any pointed-to file.

LIST^DIC( ): Lister

This procedure returns a sorted list of entries from a file. Callers must specify a file number. Callers can also specify the index to be used in sorting the output, a starting location, a number of records to retrieve and/or a partial match value. They can also pass screening logic. By default, the Lister returns the .01 field of the entries, along with the index value(s) used to retrieve them, and all indentifiers for the entries. The developer can override the default output and return other information for the entries.

This call is designed to populate a GUI Listbox gadget. It merely returns a list of entries from an index. Starting values must be in the same format as the index, unlike a lookup which allows search values to be in external format. The caller can make an initial call to the Lister to return a number of records "n" from the file and follow that by subsequent calls to return the next "n" records.

Format

```
LIST^DIC(FILE,IENS,FIELDS,FLAGS,NUMBER,[.]FROM,[.]PART,INDEX,[.]SCREEN,
IDENTIFIER,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE

(Required) The file whose entries are to be listed. This should equal the file or subfile number, depending on what the caller wishes to list.

IENS

(Optional) If the FILE parameter equals a file number, the Lister will ignore the IENS parameter. If the FILE parameter equals a subfile number, the Lister needs the IENS parameter to help identify which subfile to list. In other words, files can be specified with the FILE parameter alone, but subfiles require both the FILE and IENS parameters.

When the IENS parameter is used, it must equal an IENS that identifies the parent record of the exact subfile to list. Since this parameter identifies the subfile under that record, and not the subrecord itself, the first comma-piece of the parameter should be empty. (For more information on the IENS, see the discussion in the DBS Introduction.)

For example, to specify the Menu Item subfile under option

number 67, you must pass FILE = 19.01 (the subfile number for the Menu subfile) and IENS = ",67," (showing that record number 67 holds the Menu subfile you want to list).

Defaults to empty string.

FIELDS

(Optional) The fields to return with each entry found. This parameter can be set equal to any of the specifications listed below. The individual specifications should be separated by semicolons (";").

NOTE: In most cases, a developer will want to include the "@" specifier (described below) to suppress the default output values normally returned by the Lister and then specify the fields and other elements to return here in the FIELDS parameters. This gives the developer full control over exactly what will be returned in the output list and makes the call more self-documenting in the developer's code.

- **Field Number:** This specifier makes the Lister return the value of the field for each record found. For example, specifying .01 returns the value of the .01 field. You can specify computed fields. You cannot specify word processing or multiple fields. By default, fields will be returned in external format. The "I" suffix (described below) can be appended to the field number to get the internal format of the field.

  If a field is listed multiple times in the FIELDS parameter, it is returned multiple times in packed output, but only once in unpacked output, since the field number is one of the subscripts of the unpacked output.

- **IX:** This returns, for each record, the value(s) from the index used in the call. If a subscript in the index is derived from a field, the external format of that field will be returned by default. Otherwise, the value will be returned directly as it appears in the index. The "I" suffix (described below) can be appended to IX to get the internal index value(s). The index values are returned in the "ID" nodes as

described in the Output section below.

NOTE: For records located on a mnemonic index entry, the value from the index entry will always be returned, rather than its corresponding external field value.

- **FID:** This returns the fields display identifiers (i.e., field identifiers). By default, the field values are returned in external format. The "I" suffix (described below) can be appended to FID to get the internal format of the field identifiers.

- **WID:** This returns the fields WRITE (display only) identifiers. The Lister executes each WRITE identifier's M code and copies contents of ^TMP("DIMSG",$J) to the output. You must ensure that the WRITE identifier code issues no direct I/O, but instead calls EN^DDIOL.

  NOTE: The "I" suffix, described below, cannot be used with "WID" and will generate an error.

- **.E suffix:** You can append an "E" to a field number, the specifier "IX", or the specifier "FID" to force the fields to be returned in external format. You can use both the "E" and "I" suffix together (ex., .01EI) to return both the internal and external values of the field.

- **.I suffix:** You can append an "I" to a field number, the specifier "IX", or the specifier "FID" to force the fields to be returned in internal format. You can use both the "E" and "I" suffix together (ex., .01IE) to return both the internal and external value of the field.

- **- prefix:** A minus sign (-) prefixing one of the other field specifiers tells the Lister to exclude it from the returned list. This could be used, for example, in combination with the "FID" specifier to exclude one of the identifier fields. For example, if field 2 was one of the field identifiers for a file, "FID;-2" would output all of the field identifiers except for field 2.

- **@:** This suppresses all the default values normally returned by the Lister, except for the IEN and any fields and values specified in the FIELDS parameter. It is recommended that developers ALWAYS use the "@" specifier in their Lister calls. Use of the "@" specifier allows the developer to control exactly what will be returned in the output. See the default values below to see what is normally returned by the Lister.

Default Values

If you DO NOT pass a FIELDS parameter, the Lister returns:

1. The IEN
2. The indexed field value, in external format (note that for mnemonic cross-referenced entries, this would be the mnemonic subscript, not a field value)
3. The .01 field, in external format, if the indexed field value is not .01
4. Any field display identifiers
5. Any WRITE (display-only) identifiers
6. The results of executing the Lister's IDENTIFIER parameter

If you DO pass a FIELDS parameter but it does not contain the @ specifier, the Lister returns:

1. The IEN
2. The indexed field value, in external format (note that for mnemonic cross-referenced entries, this would be the mnemonic subscript, not a field value)
3. The .01 field, in external format, if the indexed field value is not .01
4. The fields and values specified by the FIELDS parameter

5. Any WRITE (display-only) identifiers
6. The results of executing the Lister's IDENTIFIER parameter

FLAGS (Optional) Flags to control processing:

B **B**ackwards. Traverses the index in the opposite direction of normal traversal.

I **I**nternal format is returned. All output values are returned in internal format (the default is external). Because the new "I" suffix can be used in the FIELDS parameter to return information in internal format, using I in the FLAGS parameter is virtually obsolete. It greatly simplifies the call to use the "@" specifier in the FIELDS parameter to suppress return of default values and to specify in the FIELDS parameter exactly what other data elements are to be returned. You can use the "I" suffix if you wish to have them returned in internal format.

K Primary **K**ey used for default index.

M **M**nemonic suppression. Tells the Lister to ignore any mnemonic cross-reference entries it finds in the index.

P **P**ack output. This flag changes the Lister's output format to pack the information returned for each record onto a single node per record. See the information below in the Output, the Details and Features, and the Examples sections for more details.

Q **Q**uick List. If this flag is passed, the Lister will use the order of the index to return the output, rather than sorting the information into a more user-friendly order. This will make a difference when doing Lister calls where the index value is a pointer or variable

pointer. The call will be more efficient but the output may not be in an intuitive order.

When the Q flag is used, both the FROM and PART parameters must be in the same format as the subscripts found in the index whose name is passed in the INDEX parameter. In the case of a pointer, for example, the FROM and PART parameters would be an internal pointer value. See the description of the FROM, PART and INDEX parameters.

U **U**nscreened lookup. This flag makes the Lister ignore any whole file screen (stored at ^DD(file#,0,"SCR")) on the file specified in the FILE parameter. **NOTE:** Passing this flag does NOT make the Lister ignore any code passed in the SCREEN parameter.

NUMBER (Optional) The number of entries to return. If the Lister reaches the end of its list, the number of entries output may be fewer than this parameter. A value of "*" or no value in this parameter designates all entries. The developer has the option to make multiple calls to the Lister, in order to control the number of records returned. In that case, the FROM value (described below) must be passed by reference, and should not be altered between calls. The Lister will return—in the FROM parameter—the values needed to find the next record on a subsequent call.

Defaults to "*".

[.]FROM (Optional) The index entry from which to begin the list (e.g., a FROM value of "XQ" would list entries following XQ). The FROM values must be passed as they appear in the index, not in external value. The index entry for the FROM value itself is not included in the returned list.

If the INDEX parameter specifies a compound index (i.e., one with more than one data-valued subscript), then the FROM parameter should be passed by reference as an array where FROM(n) represents the "**n**th" subscript on the compound index. This array helps VA FileMan find a single entry in the index. Generally, the developer can set

the FROM array to establish a starting point from which the Lister should traverse the index. However, the FROM array is especially useful when making multiple calls to the Lister to return records in discrete chunks. The Lister sets the FROM array to information about the last record returned, so the developer can simply pass this array unchanged from one Lister call to the next to return the next set of records.

This parameter can contain an array node FROM("IEN"). This subscript can be set equal to a record number that identifies the specific entry from which to begin the list. This can alternately be passed as FROM(m) where "m" is equal to the number of data value subscripts in the index plus 1. This array entry would be passed only when there is more than one entry in the index with the same values in all of the data value subscripts. For example, using a regular single-field index on a NAME field, if there were two "SMITH,JOHN" entries in the file with IENs of 30 and 43, then passing FROM(1)="SMITH,JOHN" and either FROM(2) or FROM("IEN")=30 would return a list of entries starting with name of SMITH,JOHN and IEN of 43. If the list is built using the upright file (INDEX parameter="#"), then FROM, FROM(1) and FROM("IEN") would all be the same and would represent the starting internal entry number for the list.

When listing an index on a Pointer or Variable Pointer field, the FROM value should equal a value from the "B" index at the end of the pointer chain, NOT a pointer value. However, the FROM("IEN") should still equal the number of a record in the pointing file as it does for other Lister calls. For example, suppose you have listed entries from a simple index that points to the STATE file and the previous call finished with entry 12 which points to Utah (record 49 in the STATE file). Then FROM(1) would be set to "UTAH" and FROM("IEN") or FROM(2) would be set to 12. Again, you would only want to set FROM(2) if there were other entries in your file that pointed to Utah, with IENs that followed 12.

This parameter lets the caller make multiple calls to the Lister to return a limited number of records with each call, rather than one large one. If the FROM parameter values are passed by reference, then the Lister will return—in the

FROM array—information that will tell it which record to start with on subsequent Lister calls.

To start a new list, pass FROM undefined or equal to the empty string. This will start the list with the first entry in the index unless you're traversing the index backwards, in which case, it will start the list with the last entry in the index.

See Details and Features and the Examples sections for more help on how to use this parameter.

[.]PART

(Optional) The partial match restriction. For example, a PART value of "DI" would restrict the list to those entries starting with the letters "DI". Again, this value must be a partial match to an index value, not the external value of a field. This can be passed by reference and subscripted the same as the FROM parameter so that PART values can be specified for any subscript in a compound index.

PART is often a partial match to FROM. For example, FROM(1)="ZTMMGR", and PART(1)="ZTM" would return only entries that began with "ZTM" and came after "ZTMMGR". It would not include "ZTZERO", even though it comes after "ZTMMGR". (If traversing the index backwards, it would find only entries that came before ZTMMGR).

If FROM is passed and PART is not a partial match to FROM, then the Lister will return all the partial matches to PART that come after FROM. Thus if FROM(1)="DI" and PART(1)="ZTM", then the Lister returns all partial matches to "ZTM". If in this example we were traversing the index backwards, then the lister would return nothing, because there would be nothing that came before "DI" and started with "ZTM".

For indexes on pointers or variable pointers, PART should refer to values on the "B" index of the pointed-to file at the end of the pointer chain. For example if the index was on a field pointing to the STATE file, PART(1) could be set to "A" to find all states whose name begins with "A".

INDEX

(Optional) The name of the index from which to build the list. For example, setting this to "C" could refer to the Upper Case Menu Text index on the Option file. Whether

the specified index is simple (single data-value subscript like the "B" index on most files) or compound (more than one data-value subscript) affects the FROM and PART parameters as previously described.

If the index is not specified, the default will be "B" unless the FLAGS parameter contains a K, in which case, the default will be the Uniqueness Index defined for the Primary Key on the file.

If there is no "B" index and either "B" is passed in the INDEX parameter or is the default index, then a temporary index is built on the file (which could take some time). The index is removed after the Lister call.

If "#" is passed in the INDEX parameter, then the list will be built from the upright file (i.e., in order by internal entry number) rather than from an index. In that case, if a FROM value is passed, it should be an IEN and could be passed either as a literal or in FROM(1) or FROM("IEN"), all of which are equivalent (see FROM parameter above).

Unless the M flag is used to suppress them, mnemonic cross-references folded into the specified index are included in the output.

[.]SCREEN

(Optional) **Entry Screen.** The screen to apply to each potential entry in the returned list to decide whether or not to include it. This may be set to any valid M code that sets $TEST to 1 if the entry should be included, to 0 if not. This is exactly equivalent to the DIC("S") input variable to Classic FileMan lookup ^DIC. The Lister will execute this screen in addition to any SCR node (whole-file screen) defined for the file. Optionally, the screen can be defined in an array entry subscripted by "S" (for example, SCR("S")), allowing additional screen entries to be defined for variable pointer fields as described below.

The Entry Screen code can rely upon the following:

Naked indicator	Zero-node of entry's record.
D	Index being traversed.
DIC	Open global reference of file being traversed.

DIC(0) Flags passed to the Lister.

Y Record number of entry under consideration.

Y() array For subfiles, descendants give record numbers for all upper levels. Structure resembles the DA array as used in a call to the classic FileMan edit routine ^DIE.

Y1 IENS equivalent to Y array.

The SCREEN parameter can safely change any of these values. For example, suppose there is a set of codes field defined as the 5th piece of the 0 node on the file and you only want to find entries that have the value "Y" in that field. Then the code might look like "I $P(^(0),U,5)=""Y""". All other variables used, however, must be carefully namespaced.

Defaults to no extra screening.

Variable Pointer Screen. If one of the fields indexed by the cross-reference passed in the INDEX parameter is a variable pointer, then additional screens equivalent to the DIC("V") input variable to Classic FileMan lookup ^DIC can also be passed. Suppose the screens are being passed in the SCR array. Then for a simple index with just one data value field, the code can be passed in SCR("V"). For simple or compound indexes, screens can be passed for any indexed fields that are variable pointers in the format SCR("V",n) where "**n**" represents the subscript location of the variable pointer field on the index from the INDEX parameter.

The Variable Pointer screen restricts the user's ability to see entries on one or more of the files pointed to by the variable pointer. The screen logic is set equal to a line of M code that will return a truth value when executed. If it evaluates TRUE, then entries that point to the file can be included in the output; if FALSE, then any entry pointing to the file is excluded. At the time the code is executed, the variable Y(0) is set equal to the information for that file from the data dictionary definition of the variable pointer field. You can use Y(0) in the code set into the DIC("V") variable.

Y(0) contains:

^-Piece	Contents
Piece 1	File number of the pointed-to file.
Piece 2	Message defined for the pointed-to file.
Piece 3	Order defined for the pointed-to file.
Piece 4	Prefix defined for the pointed-to file.
Piece 5	y/n indicating if a screen is set up for the pointed-to file.
Piece 6	y/n indicating if the user can add new entries to the pointed to file.

All of this information was defined when that file was entered as one of the possibilities for the variable pointer field.

For example, suppose your .01 field is a variable pointer pointing to files 1000, 2000, and 3000. If you only want the user to be able to enter values from files 1000 or 3000, you could set up DIC("V") like this:

```
S DIC("V")="I +Y(0)=1000!(+Y(0)=3000)"
```

IDENTIFIER

(Optional) The text to accompany each potential entry in the returned list to help identify it to the end user. This may be set to any valid M code that calls the EN^DDIOL utility to load identification text. The Lister will list this text AFTER that generated by any M identifiers on the file itself. This parameter takes and can change the same input as the SCREEN parameter.

For example, a value of "D 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.

This parameter should issue no READ or WRITE commands itself nor should it call utilities that issue READs or WRITEs (except for EN^DDIOL itself).

Defaults to no extra identification text.

See the description of EN^DDIOL for more information.

TARGET_ROOT (Optional) The array that should receive the output list. This must be a closed array reference and can be either local or global. For example, if TARGET_ROOT equals OROUT(42), the output list appears in OROUT(42,"DILIST").

If the TARGET_ROOT is not passed, the list is returned descendent from ^TMP("DILIST",$J).

MSG_ROOT (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

FROM See FROM under Input Parameters. If the FROM parameter is passed by reference and if there are more entries to return in the list, then the FROM array will be set to information about the last entry returned in the current Lister call. Subsequent Lister calls will use this information to know where to start the next list.

Other than FROM(1), none of the other FROM values from the index will contain data unless the next entry to return has the same index value as the last entry returned by the current Lister call. For example, if the index is on NAME and DATE_OF_BIRTH: if the last entry returned was for "Smith,John" and there is only one "Smith,John" in the file, then FROM(1)="Smith,John", FROM(2)="", FROM(3)="". However, if there is another "Smith,John", with a different DOB, then you might have FROM(1)="Smith,John", FROM(2)=2690101. If there are two "Smith,John" entries with the same DOB, then FROM(1)="Smith,John", FROM(2)=2690101, FROM(3)=the IEN of the last entry output.

TARGET_ROOT

The examples in this section assume that the output from the Lister was returned in the default location descendent from ^TMP("DILIST",$J), but it could just as well be in an array specified by the caller in the TARGET_ROOT parameter described above.

There are two different formats possible for the output: (1) Standard output format and (2) Packed output (format returned when the P flag is included in the FLAGS parameter).

1. Standard Output Format

The format of the Output List is:

- **Header Node**

  Unless the Lister has run into an error condition, it will always return a header node for its output list, even if the list is empty because no matches were found. The header node on the zero node of the output array, has this format:

  ```
  ^TMP("DILIST",$J,0) = # of entries found ^
  maximum requested ^ any more? ^ results flags
  ```

  1. The # of entries found will be equal to or less than the maximum requested.

  2. The maximum requested should equal the NUMBER parameter, or, if NUMBER was not passed, "*".

  3. The any more? value is 1 if there are more matching entries in the file than were returned in this list, or 0 if not.

  4. The results flags at present is usually empty. If the output was packed, and some of the data contained embedded "^" characters, the results flag contains the flag H. Check for the the results containing H rather than results equal to H. For more information see Details and Features.

- **Record Data**

Standard output for the Lister returns each field of each matching record on a separate node. Records are subscripted in this array by arbitrary sequence number that reflects the order in which the record was found.

- Indexed Field (Simple Index)

  Unless suppressed with the "@" in the FIELDS parameter (the suggested practice), the indexed values are returned descendent from the 1 nodes in external format.

  ```
  ^TMP("DILIST",$J,1,seq#) = index_value
  ```

  NOTE: This is different from the Finder, which returns the .01 field value in the 1 subtree.

- Indexed Field (Compound Index)

  If the Lister call used a compound index, an additional sequential integer reflects the subscript position at which the value was found.

  ```
  ^TMP("DILIST",$J,1,seq#,1) =
  first_subscript_index_value

  ^TMP("DILIST",$J,1,seq#,2) =
  second_subscript_index_value
  ```

- IEN

  Each record's IEN is returned under the 2 subtree:

  ```
  ^TMP("DILIST",$J,2,seq#) = IEN
  ```

  The other values returned for each record are grouped together under the "ID" subtree, then by record.

- Field Values or Field Identifiers.

  The output format is the same whether the field value is one of the Field Identifiers from the data dictionary for the file, or the field was

requested in the FIELDS parameter. In addition, if the .01 field is not one of the indexed fields and is not suppressed by use of "@" in the FIELDS parameter, then it is also returned along with the other Field values. By default, field values are returned in external format.

Field identifiers and field values are subscripted by their field numbers. Each node shows up as:

```
^TMP("DILIST",$J,"ID",seq#,field#) =
field_value
```

Fields default to external format unless I is passed in the FLAGS parameter (obsolete) or the I suffix is specified in the FIELDS parameter (recommended way to get internal field values).

If both the "I" and "E" suffix are specified, an additional subscript level with the values of "E" and "I" is used to distinguish the external and internal values of the field. If a field is only returned in one format, the extra subscript is never included. Values output with the extra format specifier look like:

```
^TMP("DILIST",$J,"ID",seq#,field#,"E" or "I")
= field_value
```

- Output for field specifier "IX" in FIELDS

  A field specifier of "IX" in the FIELDS parameter retrieves the value of the indexed field(s). In the output, the values of these fields are returned as follows, where the final subscript is a sequential number indicating the subscript location in the index.

```
^TMP("DILIST",$J,"ID",seq#,0,1) =
first_subscript_index_value

^TMP("DILIST",$J,"ID",seq#,0,2) =
second_subscript_index_value
```

  If both the "I" and "E" suffix are specified, an

additional subscript level with the values of "E" and "I" is used to distinguish the external and internal values from the index. If the subscript on the index is not derived from a field, i.e. if it's a computed subscript, then the internal and external value will both be the same, the value directly from the index.

- WRITE Identifiers

  WRITE (display-only) identifiers are grouped under the "WRITE" subtree of the "ID" tree, then by record number. It is the caller's responsibility to ensure that none of the WRITE identifiers issue direct READ or WRITE commands and that they issue any output through EN^DDIOL so it can be collected by the Lister. The output from all the WRITE identifiers for a single record is listed as individual lines of text:

  ```
  ^TMP("DILIST",$J,"ID","WRITE",seq#,line #) =
  text generated by WRITE IDs
  ```

- IDENTIFIER parameter

  Any text generated by the caller's IDENTIFIER parameter is returned in the last lines of the WRITE identifier text.

- **Map Node for Unpacked Format**

  In order to facilitate finding information in the output, a Map Node is built for unpacked format. This node is returned in ^TMP("DILIST",$J,0,"MAP").

  The Map node for unpacked format describes what Field Identifier data can be found in the "ID" output data nodes. It contains ^-delimited pieces described below. The position of the piece in the map node corresponds to the order in which it can be found in the "ID" output nodes. If the data is returned in internal format, the piece will be followed by "I" (ex., "2I" means that the internal value of field 2 was returned in the

output).

- **#:** Individually requested field number, where # is the field number, for each field requested in the FIELDS parameter
- **FID(#):** Field Identifier, where # is the field number.

2. Packed Output Format

If the P flag is used to request packed output, the Lister packs all the return values into one output node per record. You must ensure that all requested data will fit onto a single node. Overflow causes error 206. Return values containing embedded "^" characters make the Lister encode the output data using HTML encoding (see Details and Features)

- **Header Node**

  Identical to Standard Output Format

- **Record Data**

  Values in the output are delimited by "^" characters. Piece 1 is always the IEN. The values of other pieces depend on the value of the FIELDS parameter. If the FIELDS parameter is not passed, each record's packed node will follow this format:

  ```
  ^TMP("DILIST",$J,seq#,0)=IEN^Indexed_field_
  values^field_Identifiers^Write_Identifiers^
  Output_from_Identifier_parameter
  ```

  Field Identifiers are sequenced by field number. Output values specified by the FIELDS parameter are packed in the order in which they occur in the FIELDS parameter. WRITE identifiers are packed in the same order as their subscripts occur in the ID subtree of the file's data dictionary.

  To parse the output of the packed nodes, use the MAP node described below.

- **Map Node for Packed Format**

Because the packed format is not self-documenting and because individual field specifiers such as FID can correspond to a variable number of field values, the Lister always includes a map node when returning output in Packed format. This node is returned in ^TMP("DILIST",$J,0,"MAP").

Its value resembles a data node's value in that it has the same number of ^-pieces, but the value of each piece identifies the field or value used to populate the equivalent location in the data nodes. The possible values for each piece in the map node are:

- **IEN:** (the IEN)
- **.01:** (the .01 field)
- **FID(#):** (Field identifier, where # is the field number of the identifier)
- **WID(string):** (Write identifier, where string is the value of the subscript in the ^DD where the identifier is stored (such as "WRITE"))
- **IDP:** (Identifier parameter)
- **IX(n):** Indexed field values, where "n" refers to the subscript position in the index.
- #: Individually requested field, by field number

NOTE: For any piece except IEN, the WID, or the IDP, if the internal value is to be returned, the piece will be followed by "I". Thus instead of "IX(1)", you would have "IX(1)I", indicating that the internal index value was being returned.

For example, the map node for a Lister call on the OPTION file, if FIELDS => "3.6I;3.6;4", might look like this:

```
^TMP("DILIST",$J,0,"MAP") = "IEN^.01^3.6I^3.6^4"
```

Examples

Example 1

This is an example of a forward traversal of the "B" index on the Option file, limited to five entries that all begin with the characters "DIFG", but skipping any first entry that might equal "DIFG" (the FROM value is always skipped):

```
>D LIST^DIC(19,"","","",5,"DIFG","DIFG","","","","OUT")

OUT("DILIST",0)=5^5^1^
OUT("DILIST",0,"MAP")=FID(1)
OUT("DILIST",1,1)=DIFG CREATE
OUT("DILIST",1,2)=DIFG DISPLAY
OUT("DILIST",1,3)=DIFG GENERATE
OUT("DILIST",1,4)=DIFG INSTALL
OUT("DILIST",1,5)=DIFG SPECIFIERS
OUT("DILIST",2,1)=321
OUT("DILIST",2,2)=322
OUT("DILIST",2,3)=323
OUT("DILIST",2,4)=326
OUT("DILIST",2,5)=325
OUT("DILIST","ID",1,1)=Create/Edit Filegram Template
OUT("DILIST","ID",2,1)=Display Filegram Template
OUT("DILIST","ID",3,1)=Generate Filegram
OUT("DILIST","ID",4,1)=Install/Verify Filegram
OUT("DILIST","ID",5,1)=Specifiers
```

Example 2

This related example reveals that there is a DIFG option. When we traverse backward, starting with the first entry from the previous example, DIFG is the only option that meets both the FROM and PART parameter criteria. The sequence number is 5. When we traverse an index backward to get a set number of records, the sequence number counts backward from that number in order to make the output come out in the same order as when we traverse forward. This type of Lister call is normally used in a GUI ListBox when the user is backing up on a list.

```
>D LIST^DIC(19,"","","B",5,"DIFG
CREATE","DIFG","","","","OUT")

OUT("DILIST",0)=1^5^0^
OUT("DILIST",0,"MAP")=FID(1)
OUT("DILIST",1,5)=DIFG
OUT("DILIST",2,5)=327
OUT("DILIST","ID",5,1)=Filegrams
```

Example 3

In this example we'll return just one entry from a file using a compound index. This index is on the .01 field (NAME) and field 1 (DATE OF BIRTH). Note how the two index entries are returned in the 1 nodes. Also note that this file has several field identifiers and WRITE identifiers. After the call, because there are two different entries in the file with a .01 equal to "ADDFIFTEEN", but different dates of birth, the DIFR array has been set up ready for a subsequent call. On this index, the DATE OF BIRTH field has a collation of "backwards", so we see the most current date first in the output.

```
>K DIFR,DIPRT S DIPRT(1)="ADD"

>D LIST^DIC(662001,"","","",1,.DIFR,.DIPRT,"BB","","","OUT")

OUT("DILIST",0)=1^1^1^
OUT("DILIST",0,"MAP")=FID(2)^FID(4)^FID(10)
OUT("DILIST",1,1,1)=ADDFIFTEEN
OUT("DILIST",1,1,2)=JAN 03, 1997
OUT("DILIST",2,1)=17
OUT("DILIST","ID",1,2)=SEVENTEEN*
OUT("DILIST","ID",1,4)=MITTY,WALTER
OUT("DILIST","ID",1,10)=MAY 02, 1997@09:00
OUT("DILIST","ID","WRITE",1,1)=2970103
OUT("DILIST","ID","WRITE",1,2)=
OUT("DILIST","ID","WRITE",1,3)=  FIRST LINE
OUT("DILIST","ID","WRITE",1,4)=
OUT("DILIST","ID","WRITE",1,5)=          SECOND LINETHIRD LINE
OUT("DILIST","ID","WRITE",1,6)=SIXTHCODE

>ZW DIFR

DIFR=ADDFIFTEEN
DIFR(1)=ADDFIFTEEN
DIFR(2)=2970103
DIFR(3)=
DIFR("IEN")=
```

Example 4

However, if we do another Lister call on the same file, using the DIFR array that was passed back from the previous call, this time we'll return two records. We get back the second record in the index with "ADDFIFTEEN" as the .01 field, and the next one that follows it alphabetically. In this call, we suppressed the normal default values returned by the call, and instead asked for the index field values "IX", the internal value of the field identifiers "FIDI", both the internal and external values of field 3 (a set-of-codes type field), and the external value of computed field 8. All of this was done with entries in the FIELDS parameter. As you see, field 4 is

a pointer, field 10 is a variable pointer. Note how the MAP node describes what is found in the "ID" nodes.

```
>D LIST^DIC(662001,"","@;IX;FIDI;3IE;8","",2,.DIFR,.DIPRT,"BB","","","OUT")

OUT("DILIST",0)=2^2^1^
OUT("DILIST",0,"MAP")=IX(1)^IX(2)^FID(2)I^3^3I^FID(4)I^8^FID(10)I
OUT("DILIST",2,1)=15
OUT("DILIST",2,2)=14
OUT("DILIST","ID",1,0,1)=ADDFIFTEEN
OUT("DILIST","ID",1,0,2)=JAN 01, 1969
OUT("DILIST","ID",1,2)=FIFTEEN
OUT("DILIST","ID",1,3,"E")=SIXTHCODE
OUT("DILIST","ID",1,3,"I")=SIX
OUT("DILIST","ID",1,4)=1
OUT("DILIST","ID",1,8)=0
OUT("DILIST","ID",1,10)=327;DIC(19,
OUT("DILIST","ID",2,0,1)=ADDFOURTEEN
OUT("DILIST","ID",2,0,2)=JAN 01, 1949
OUT("DILIST","ID",2,2)=FOURTEEN
OUT("DILIST","ID",2,3,"E")=
OUT("DILIST","ID",2,3,"I")=
OUT("DILIST","ID",2,4)=
OUT("DILIST","ID",2,8)=32.6
OUT("DILIST","ID",2,10)=10;DIZ(662003,
```

Example 5

In this example, we use the P flag to return the next two records in Packed output format. We revert to letting the Lister return default values, rather than controlling them with the FIELDS parameter, but we'll return additional output by using the IDENTIFIER parameter. Note that although we asked for two records, there was only one left that fit our PART criteria. The first piece of the header node tells us one record was returned; the second piece tells us that two records were requested; the third tells us there are no records left that meet the criteria.

Here's what the FROM values are set to going into the call:

```
DIFR=ADDFOURTEEN
DIFR(1)=ADDFOURTEEN
DIFR(2)=
DIFR(3)=
DIFR("IEN")=

>D LIST^DIC(662001,"","","P",2,.DIFR,.DIPRT,"BB","","D
EN^DDIOL(""Hi there"")"," OUT")

OUT("DILIST",0)=1^2^0^
OUT("DILIST",0,"MAP")=IEN^IX(1)^IX(2)^FID(2)^FID(4)^FID(10)^WID(WRITE1)^WID(W
RIT
E2)^WID(WRITE3)^WID(WRITE4)^IDP
```

```
OUT("DILIST",1,0)=16^ADDSIXTEEN^MAR 28, 1970^MA HERE TOO*^^DIFG^2700328^^
FIRST
 LINE~~          SECOND LINETHIRD LINE^^Hi there
```

Error Codes Returned

120 Error occurred during execution of a VA FileMan hook.

202 Missing or invalid input parameter.

205 The File and IENS represent different subfile levels.

206 The data requested for the record is too long to pack together.

207 The value is too long to encode into HTML.

301 The passed flags are missing or inconsistent.

304 The IENS lacks a final comma.

306 The first comma-piece of the IENS should be empty.

401 The file does not exist.

402 The global root is missing or not valid.

406 The file has no .01 field definition.

407 A word-processing field is not a file.

420 The index is missing.

501 The file does not contain that field.

520 That kind of field cannot be processed by this utility.

The Lister may also return any error returned by $$EXTERNAL^DILFD.

Details and Features

Screens Applied

Aside from the optional screen parameter, the Lister applies one other screen to each index entry before adding it to the output list as follows: ^DD(file#,0,"SCR"). Other screens, such

as the 7.5 node and field-level screens on various data types,are not checked because they relate specifically to entry and editing, not selection.

Output Transform

It is possible for any field with an output transform to sort differently than a user would expect. Although the value displayed is the output value, the value that determines its order is its internal value. When the I flag is used, the output transform is never executed, and the output will always appear in the expected order.

HTML Encoding

Since the Lister uses the "^" character as its delimiter for packed output, it cannot let any of the data contain that character. If any does, it will encode all of the data using an HTML encoding scheme.

In this scheme, all "&" characters are replaced with the substring "&" and all "^" characters with the string "&#94". This keeps the data properly parsable and decodable. The data for all records found, not just the ones with embedded ^s, will be encoded if embedded ^s are found in the data of any of the records.

If the Lister has encoded the output, it will include an H flag in ^-piece four of the output header node.

Data can be decoded using the VA FileMan library function call $$HTML^DILF(encoded string,-1). It can properly decode individual fields or complete packed data nodes.

Pointers and Variable Pointers

The Lister treats indexes on fields of these two data types specially. For every other data type, the value of the indexed field is completely contained in the file indicated by the FILE parameter. For pointer and variable pointers, this is not the case. All index values come from the B index of the pointed-to file. The Lister uses the values in the pointed-to file, extending the search to the end of the pointer chain, to select records in the pointing file at the beginning of the chain.

For example, suppose the FILE parameter picks file A, and the INDEX parameter picks the X index, a cross-reference on a pointer field. Suppose further that field points to file B, whose .01 field points to file C, and file C's .01 is a set of codes. Then this Lister call will select records in file A (the pointing file) based on the index values it finds in file C (the pointed-to file).

The FROM("IEN"), SCREEN, and IDENTIFIER parameters always apply to the pointing file, the one identified by the FILE parameter, because they deal with actual record selection. However, for pointers and variable pointers, the FROM and PART parameters apply to the "B" index on the pointed-to file, since they deal with index values.

Variable pointers work similarly, except that their index values usually come from more than one pointed-to file.

WRITE ID nodes

The Lister executes each individual WRITE ID node from the data dictionary. If an individual node results in creating multiple lines in the output from the EN^DDIOL call(s) it contains, then in Standard Output Format the results will appear on multiple lines in the output array. Thus there is not a direct correlation between the number of WRITE ID nodes and the number of nodes that will be returned in the output array of a Lister call for each record. In Packed output format, each WRITE ID node appears in a separate "^" piece and line feeds are designated with a tilde (~) character.

FROM parameter with Compound Indexes

The FROM parameter designates only a starting point on the index defined in the INDEX parameter. For example, we have a compound index where the first subscript is a NAME and the second is a DATE OF BIRTH. Supposing that after a Lister call, FROM(1)="SMITH,JOHN" and FROM(2)="2690101. A subsequent Lister call assumes that there must be another entry with the name "SMITH,JOHN", but a date-of-birth that follows 1/1/69. Any other entries returned will have names that equal or follow SMITH,JOHN, but after processing all of the SMITH,JOHN entries, other output entries could have any date-of-birth. This is NOT like a sort where we say that we want only entries where the date-of-birth follows 1/1/69.

FIELD^DID( ): DD Field Retriever

This procedure retrieves the values of the specified field-level attributes for the specified field.

Format

```
FIELD^DID(FILE,FIELD,FLAGS,ATTRIBUTES,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number.

FIELD (Required) Field name or number.

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

N	No entry in the target array is created if the attribute is null.
Z	Word processing attributes include Zero (0) nodes with text.

ATTRIBUTES (Required) A list of attribute names separated by semicolons. Full attribute names must be used. Following are the attributes that can be requested:

AUDIT

AUDIT CONDITION

COMPUTE ALGORITHM

COMPUTED FIELDS USED

DATE FIELD LAST EDITED

DECIMAL DEFAULT

DELETE ACCESS

DESCRIPTION

FIELD LENGTH

GLOBAL SUBSCRIPT LOCATION

HELP-PROMPT

INPUT TRANSFORM

LABEL

MULTIPLE-VALUED

OUTPUT TRANSFORM

POINTER

READ ACCESS

SOURCE

SPECIFIER

TECHNICAL DESCRIPTION

TITLE

TYPE

WRITE ACCESS

XECUTABLE HELP

TARGET_ROOT (Required) The closed root of the array that should receive the attributes.

MSG_ROOT (Optional) The name of a closed root reference that is used to pass error messages. If not passed, ^TMP("DIERR",$J) is used.

Output

TARGET_ROOT The array is subscripted by the attribute names.

Example

```
>D FIELD^DID(999000,.01,"","LABEL;TYPE","TEST1")

>ZW TEST1
TEST1("LABEL")=NAME
TEST1("TYPE")=FREE TEXT
```

Error Codes Returned

200 There is an error in one of the variables passed.

202 Missing or invalid input parameter.

301 Flags passed are unknown or incorrect.

401 The specified file or subfile does not exist.

403 The file lacks a Header Node.

404 The file Header Node lacks a file #.

501 The field name or number does not exist.

505 The field name passed is ambiguous.

510 The data type for the specified field cannot be determined.

520 An incorrect kind of field is being processed.

537 Field has a corrupted pointer definition.

FIELDLST^DID( ): DD Field List Retriever

This procedure returns a list of field-level attributes that are supported by FileMan. It shows specifically which attributes the Data Dictionary retriever calls can return.

Format

```
FIELDLST^DID(TARGET_ROOT)
```

Input Parameters

TARGET_ROOT (Required) The root of an output array.

Output

TARGET_ROOT The descendents of the array root are subscripted by the attribute names. "WP" nodes indicate that the attribute consists of a word processing field.

Example

Below is a partial capture of what is returned:

```
>D FIELDLST^DID("TEST")

>ZW TEST
TEST("AUDIT")=
TEST("AUDIT CONDITION")=
TEST("COMPUTE ALGORITHM")=
TEST("COMPUTED FIELDS USED")=
      .
      .
      .
```

FILE^DID( ): DD File Retriever

This procedure retrieves the values of the file-level attributes for the specified file. It does not return subfile attributes.

Format

```
FILE^DID(FILE,FLAGS,ATTRIBUTES,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File number (but not subfile attributes).

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

- **N** **N**o entry in the target array is created if the attribute is null.
- **Z** Word processing attributes include **Z**ero (0) nodes with text.

ATTRIBUTES (Required) A list of attribute names separated by semicolons. Full attribute names must be used:

ARCHIVE FILE

AUDIT ACCESS

DATE

DD ACCESS

DEL ACCESS

DESCRIPTION

DEVELOPER

DISTRIBUTION PACKAGE

ENTRIES

GLOBAL NAME

LAYGO ACCESS

LOOKUP PROGRAM

NAME

PACKAGE REVISION DATA

REQUIRED IDENTIFIERS

RD ACCESS

VERSION

WR ACCESS

TARGET_ROOT (Required) The name of a closed array reference.

MSG_ROOT (Optional) The name of a closed root array reference that is used to pass error messages. If not passed, messages are returned in ^TMP("DIERR",$J).

Output

TARGET_ROOT The array is subscripted by the attribute names. Some attributes can have multiple sub-attributes and these are further subscripted with a sequence number and the sub-attribute name. Attributes that contain word processing text also have a sequence number for each line of text.

Example

```
>D FILE^DID(999000,"","NAME;GLOBAL NAME;REQUIRED IDENTIFIERS","TEST")

>ZW TEST
TEST("GLOBAL NAME")=^DIZ(999000,
TEST("NAME")=ZZZDLTEST
TEST("REQUIRED IDENTIFIERS")=TEST("REQUIRED IDENTIFIERS")
TEST("REQUIRED IDENTIFIERS",1,"FIELD")=.01
TEST("REQUIRED IDENTIFIERS",2,"FIELD")=1
```

Error Codes Returned

200 There is an error in one of the variables passed.

202 Missing or invalid input parameter.

301 Flags passed are unknown or incorrect.

401 The specified file or subfile does not exist.

403 The file lacks a Header Node.

404 The file Header Node lacks a file #.

501 The field name or number does not exist.

505 The field name passed is ambiguous.

510 The data type for the specified field cannot be determined.

520 An incorrect kind of field is being processed.

537 Field has a corrupted pointer definition.

FILELST^DID( ): DD File List Retriever

This procedure returns a list of file-level attributes that are supported by FileMan. It shows specifically which attributes the Data Dictionary retriever calls can return.

Format

```
FILELST^DID(TARGET_ROOT)
```

Input Parameters

TARGET_ROOT (Required) The root of an output array.

Output

TARGET_ROOT The descendents of the array root are subscripted by the attribute names. "WP" nodes indicate that the attribute consists of a word processing field. "M" nodes indicate that the attribute can consist of multiple sub-attributes.

Example

```
>D FILELST^DID("TEST")

>ZW TEST
TEST("ARCHIVE FILE")=
TEST("AUDIT ACCESS")=
TEST("DATE")=
TEST("DD ACCESS")=
TEST("DEL ACCESS")=
TEST("DESCRIPTION")=
TEST("DESCRIPTION","#(word-processing)")=
TEST("DEVELOPER")=
TEST("DISTRIBUTION PACKAGE")=
TEST("ENTRIES")=
TEST("GLOBAL NAME")=
TEST("LAYGO ACCESS")=
TEST("LOOKUP PROGRAM")=
TEST("NAME")=
TEST("PACKAGE REVISION DATA")=
TEST("REQUIRED IDENTIFIERS")=
TEST("REQUIRED IDENTIFIERS","#","FIELD")=
TEST("RD ACCESS")=
TEST("VERSION")=
```

```
TEST("WR ACCESS")=
```

"RD ACCESS" in the example above is a new ATTRIBUTES Input Parameter.

$$GET1^DID( ): Attribute Retriever

This extrinsic function retrieves a single attribute from a single file or field.

Format

```
$$GET1^DID(FILE,FIELD,FLAGS,ATTRIBUTE,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE	(Required) File number.
FIELD	Field number or name. (Required only when field attributes are being requested, otherwise this function assumes a file attribute is being requested)
FLAGS	(Optional) Flag to control processing:
	Z — **Z**ero nodes on word processing attributes are included in the array subscripts.
ATTRIBUTE	(Required) Data dictionary attribute name.
TARGET-ROOT	Closed array reference where multi-lined attributes will be returned. (Required only when multi-line values are returned, such as word processing attributes like "DESCRIPTION")
MSG-ROOT	(Optional) The name of a closed root reference that is used to pass error messages. If not passed, ^TMP("DIERR",$J) is used.

Output

A successful call returns the attribute requested. This can either be set into a variable or written to the output device.

Examples

Example 1

```
> S X=$$GET1^DID(999000,"","","DESCRIPTION","ARRAY","ERR") ZW @X
ARRAY(1)=This is the description of the file (ZZZDLTEST).
ARRAY(2)=And this is the second line of the description.
```

Example 2

```
>W $$GET1^DID(999000,"","","GLOBAL NAME")
^DIZ(999000,
```

Example 3

```
>W $$GET1^DID(999000,.01,"","LABEL")
NAME
```

Example 4

```
>S X=$$GET1^DID(999000,.01,"Z","DESCRIPTION","ARRAY","ERR") ZW @X
ARRAY(1,0)=This is the description of the .01 filed
ARRAY(2,0)=in file 999000.

>W X
ARRAY
```

Error Codes Returned

200 Parameter is invalid or missing.

202 Specified parameter in missing or invalid.

505 Ambiguous field.

Details and Features

File/Field This retriever call differentiates whether the request is for a file or a field by the second parameter. If the second parameter is null, the retriever assumes (since no field is passed) that a file attribute is desired. If the second parameter is not null, the retriever assumes a field attribute is requested.

CHK^DIE( ): Data Checker

This procedure checks user-supplied data against the data dictionary definition of a field. If the input data passes the validation, the internal and, optionally, the external forms of the data are returned. In this respect, CHK^DIE is the inverse of the $$EXTERNAL^DILFD call.

While this procedure indicates that a user's response is valid according to a field's definition, it does not assure that a value can be filed in a particular record. In order to verify that a value can be filed, use the VAL^DIE or FILE^DIE calls (with the E flag). CHK^DIE does not have IENS as input; it is ignorant of the state of the data.

Do not pass a VALUE of null or "@" to CHK^DIE. This procedure cannot verify that deletion of values from the database is appropriate. Again, use VAL^DIE or FILE^DIE (with E flag) for this purpose.

Format

```
CHK^DIE(FILE,FIELD,FLAGS,VALUE,.RESULT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number.

FIELD (Required) Field number for which data is being validated.

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

- **H** **H**elp (single "?") is returned if VALUE is not valid.
- **E** **E**xternal value is returned in RESULT(0).

VALUE (Required) Value to be validated, as entered by a user. VALUE can take several forms depending on the data type involved, e.g., a partial, unambiguous match for a pointer or any of the supported ways to input dates (such as "TODAY" or "11/3/93").

.RESULT (Required) Local variable that receives output from the call. If VALUE is valid, the internal value is returned. If not valid, ^ is returned. If the E flag is passed, external value is returned in RESULT(0).

NOTE: This array is killed at the beginning of each call.

MSG_ROOT (Optional) Root into which error, help, and message arrays are put. If this parameter is not passed, these arrays are put into nodes descendent from ^TMP.

Output

See input parameters .RESULT and MSG_ROOT.

RESULT = internal value or ^ if the passed VALUE is not valid.

RESULT(0) = external value if the passed VALUE is valid and E flag is present.

Example

In the following example, data for a date/time data type is being checked. Note that the external form of the user's input, which was "T-180", is passed. In this case, the value was acceptable, as shown below:

```
>S FILE=16200,FIELD=201,FLAG="E",VALUE="T-180"

>D CHK^DIE(FILE,FIELD,FLAG,VALUE,.RESULT)

>ZW RESULT
RESULT=2930625
RESULT(0)=JUN 25,1993
```

Error Codes Returned

In addition to errors that indicate that the input parameters are invalid, the primary error code returned is:

120 Error occurred during execution of a FileMan hook.

701 Value is invalid.

Details and Features

What is checked

This call verifies that the VALUE passed is valid by passing it through the field's INPUT transform. Also, if the field has any screens, those screens must be passed. If the field is a pointer or variable pointer, this call verifies that there is an unambiguous match (or partial match) for VALUE.

Entry number caution

No internal entry numbers are available when the INPUT transform or screens for the field are executed. Therefore, the INPUT transform and screens cannot reference any entry numbers using either the DA() array or the D0, D1, D2, etc., variables. Likewise, Xecutable Help cannot reference an entry number if the H flag is sent.

FILE^DIE( ): Filer

This procedure:

- Puts validated data that is in internal FileMan format into the database.

OR:

- Validates data that is in external (user-provided) format, converts it to internal FileMan format, and files valid data into the database.

If the data to be filed is in external format, you can specify that nothing will be filed unless the values for every field being filed are valid. (Use the T and E flags).

Uniqueness and completeness of keys are enforced (unless the U flag is used). This check is performed on values passed in both internal and external formats.

The associated functions of firing cross-references and of performing data audits are also performed.

NOTE: The Filer only files data into existing entries and subentries. To add new entries or subentries, use the UPDATE^DIE call.

Format

```
FILE^DIE(flags,fda_root,msg_root)
```

Input Parameters

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

E **E**xternal values are processed. If this flag is set, 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 the flag is 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 Loc**K**ing is done by the Filer. (See discussion of Locking.)

S **S**ave FDA. If this flag is 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 **T**ransaction is either completely filed or nothing is filed. The E flag must be used with the T flag, with values passed 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.

If neither the T nor the U flag is` sent, simple keys are checked as they are encountered in the FDA. Compound keys are checked only after the entire record is filed. If the key is invalid, changes to fields making up that key are backed out.

U Don't enforce key **U**niqueness 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:** If this flag is used, the FILE^DIE call may result in records that contain null key fields or records with duplicate keys. It is the programmer's responsibility to ensure that the database is not left in a state in which the integrity of keys is violated.)

fda_root (Required) The root of the FDA that contains the data to file. The array can be a local or global one. The root is the closed array reference to be used with subscript indirection not the traditional FileMan root. See the Database Server Introduction for details of the structure of the FDA.

msg_root (Optional) The 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).

Output

Ordinarily the "output" of this call is the updating of the database. Error messages and information supplied via EN^DDIOL are returned in the standard array in ^TMP or in the array specified by msg_root.

Error Codes Returned

This call returns error messages in many circumstances. Most of the messages report bad input parameters or input to a file, field, or record that does not exist. Primary user-oriented codes include:

110 Record is locked.

120 Error occurred during execution of a FileMan hook.

701 Input data was invalid.

712 Deletion was attempted but not allowed.

740 New values are invalid because they would create a duplicate key.

742 Deletion was attempted on a key field.

744 A key field was not assigned a value.

Details and Features

Security

The Filer does not check user access when filing. This check must be done by the client application.

Deleting data

You can delete the value in a field by setting the value for the field equal to null or "@".

This works for word processing fields, too. Instead of setting the value for the field equal to the root of the array where the new word processing text is to be found, set it equal to null or "@".

NOTE: When the E (external) flag is used, you can't delete the field value if the field is either Required or Uneditable. Without the E flag, deletion occurs in both cases. When key integrity is checked (the U flag is not used), you can't delete the value of a key field whether the E flag is used or not.

You can delete an entire entry or subentry by setting the value of the .01 field to "@" or null. In this case, it does not matter whether the the .01 field is Required, Uneditable, or a key field.

The Filer never asks for confirmation of the deletion.

Scope of a Single Filer Call

Data passed to the Filer should comprise one logical record. Thus, the data can consist of values for fields in the primary file and its multiples and in related files. ("Navigation" to other files is handled by the calling application, not by the Filer.)

Cross-references

New style indexes that have an execution value of RECORD are fired once after all the data for a single record or subrecord is filed.

All other cross-references (and data audits) are fired as the data is filed, that is, on a field-by-field basis.

Any possible conflict between the cross-reference and updated data must be noted by the client application and resolved by modifying the cross-reference. The most common situation in which conflicts can arise is when a cross-reference (most frequently a trigger or MUMPS cross-reference) has been used to provide information to the user while data is being edited. Default values which are dependent on the values of other fields being edited can be provided in this way. These "user interface" cross-references are fired by the Filer with the rest of the cross-references after the data editing is complete. Thus, they cannot have their desired effect of providing the user with information during the editing session. However, they may have the undesired effect of overwriting user-entered values. This type of cross-reference must be removed from the DD as part of the preparation for using the DBS. Also, if the functionality provided by these cross-references is still desirable during the editing session, the client application will need to provide it.

Locking

If requested, the Filer incrementally locks records and subrecords before beginning to file any data. If a lock on any record fails, no filing is done and an error message is returned to the calling program.

It is recommended that locking be done outside of the Filer by the client application. There are several reasons for this:

It may be frustrating to the user to edit a screen's worth of data and then to have the SAVE fail because the necessary lock could not be obtained.

Data successfully validated may become invalid before it is filed.

The client application can more selectively determine which records to lock. Of necessity, the Filer locks all entries and subentries referenced in the FDA passed to it. In many instances, this is more than is actually required.

Locking inside the Filer requires additional processing that slows the filing action down.

However, there are situations in which it is appropriate for the Filer to do the locking; for example, if only a single file is involved and the source of the data is not an interactive editing session.

HELP^DIE( ): Helper

This procedure retrieves user-oriented help for a field from the Data Dictionary and other sources. The help is returned in arrays. (The MSG^DIALOG procedure can be used to display the help.) You control the kind of help obtained by using the FLAGS input parameter—either a specific kind of help, the help normally returned with one or two question marks, or all available help for a field.

Format

```
HELP^DIE(FILE,IENS,FIELD,FLAGS,msg_root)
```

Input Parameters

FILE (Required) File or subfile number.

IENS (Optional) Standard IENS indicating internal entry numbers. This parameter is only needed if code in the Data Dictionary for Xecutable Help or Screen on a Set of Codes references the entry number using DA() array or D0, D1, etc., and if that kind of help is being requested.

FIELD (Required) Field number for which help is requested.

FLAGS (Required) Flags used to determine what kind of help is returned by the call. If a lower case letter is shown, use it to suppress that kind of help—useful in conjunction with **?** or **??**. The possible values are:

- **?** Help equivalent to user entering one "**?**" at an edit prompt. (Also help returned for an invalid response.)
- **??** Help equivalent to user entering "**??**" at an edit prompt.
- **A** **A**ll available help for the field.
- **B** **(b)** **B**rief variable pointer help. A single line beginning with "To see the entries ...".

  NOTE: See also Limitations under Details and Features below.
- **C** Set of **C**odes screen description.

D **D**escription text for the field; this may be multiple lines.

F **F**ields that can be used for lookups. Returned for top-level .01 fields and for pointed-to files for pointer data types. For pointed-to files, the F flag is effective only if the G flag is also sent.

G (g) **G**etting help from pointed-to file. Help for the .01 field of pointed-to file is returned.

H **H**elp prompt text.

M **M**ore variable pointer help. Detailed description of how to enter variable pointer data.

P **P**ointer screen description.

S **S**et of codes possible choices. Any screen that exists on the set of codes field is applied so that only actually selectable choices are presented.

T Date/**T**ime generic help. This help text is customized based on the allowable and required elements of the particular Date/Time field.

U **U**nscreened set of codes choices.

V **V**ariable pointer help that lists the prefixes and messages associated with a particular variable pointer field.

X **X**ecutable help-the M code contained in Xecutable Help is executed. In order to have the help returned in an array, the executed code must use EN^DDIOL to load the help message.

msg_root (Optional) Closed root into which the output from the call is put. If not supplied, output is returned in ^TMP-see Output.

Output

The default output from this call is:

DIHELP	Number of lines of help text returned
^TMP("DIHELP",$J,n)	Array containing the lines of help text. The text is found in integer subscripted nodes (n), beginning with 1. A blank node is inserted between each different type of help returned.

If error messages are necessary, they are returned in the standard manner.

If the MSG_ROOT is included in the input parameters, output is returned there instead of ^TMP. The help text is returned in nodes descendent from MSG_ROOT("DIHELP").

Example

The following example illustrates the use of this call to return help text from a field that is a Set of Codes data type. This is the same help that can be obtained with a "?" in a traditional FileMan call. Note that the help is returned in the specified array descendent from MYHELP(1):

```
>D HELP^DIE(16200,"",5,"?","MYHELP(1)")

>ZW MYHELP
MYHELP(1,"DIHELP")=5
MYHELP(1,"DIHELP",1)=Only YES and MAYBE are acceptable.
MYHELP(1,"DIHELP",2)=
MYHELP(1,"DIHELP",3)=Choose from:
MYHELP(1,"DIHELP",4)=Y        YES
MYHELP(1,"DIHELP",5)=M        MAYBE
```

Error Codes Returned

120 Error occurred during execution of a FileMan hook.

301 An invalid flag was passed.

501 Field does not exist.

Details and Features

Helper and Validator

Based on a flag passed to the Validator call, single question mark help is returned by the Validator if the value being checked is invalid.

Pointed-to Files

By default you receive help for the .01 field of pointed-to files with ? or ?? when the field on which you are requesting help is a pointer. If you do not want this extended help returned, use the g flag.

Limitations

This call does not return lists of entries for .01, pointer, or variable pointer fields. Use the Lister utility to obtain these lists.

The b flag will suppress the line of Variable Pointer help that indicates a user can get a list of entries if they type <Prefix.?>. Use this flag with "?" if you are not supporting this capability.

$$KEYVAL^DIE( ): Key Validator

The Key Validator extrinsic function verifies that new values contained in the FDA do not produce an invalid key. All keys in which any field in the FDA participates are checked. If the value for a field in a key being checked is not present in the FDA, the value used to verify the key is obtained from the previously filed data.

Format

```
$$KEYVAL^DIE(FLAGS,FDA_ROOT,MSG_ROOT)
```

Input Parameters

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

Q **Q**uit when the first problem in the FDA is encountered.

FDA_ROOT (Required) The root of the FDA that contains the data to be checked. The array can be a local or global one. See the Database Server Introduction for details of the structure of the FDA.

The value of fields in the FDA must be the internal value. Do not pass external (e.g., unresolved pointer values, non-FileMan dates) in the FDA.

No action is taken on fields in the referenced FDA if those fields do not participate in a Key defined in the KEY file.

MSG_ROOT (Optional) The root of an array into which error messages are returned. If this parameter is not included, errors are returned in the default array: ^TMP("DIERR",$J).

Output

This Boolean function returns a 1 if key integrity is not violated by any value in the FDA and a 0 if an invalid key was produced by any of the values. Error messages and DIERR are also returned when necessary.

Example

In the following example, two fields from File #99999 (SAMPLE file) are set into an FDA. These are values for a new record; therefore, the IENS is "+1,". The values (".111" and "Albert Jones") are valid internal values for fields .01 and .02. $$KEYVAL^DIE returns "0" indicating that key integrity is violated by these values. The returned error message states the values create a duplicate key. The key that is duplicated is the "A" key.

```
>K MYERRORS,MYFDA

>S MYFDA(99999,"+1,",.01)=.111

>S MYFDA(99999,"+1,",.02)="Albert Jones"

>W $$KEYVAL^DIE("","MYFDA","MYERRORS")
0
>W DIERR
1^1
>ZW MYERRORS
MYERRORS("DIERR")=1^1
MYERRORS("DIERR",1)=740
MYERRORS("DIERR",1,"PARAM",0)=3
MYERRORS("DIERR",1,"PARAM","FILE")=99999
MYERRORS("DIERR",1,"PARAM","IENS")=+1,
MYERRORS("DIERR",1,"PARAM","KEY")=11
MYERRORS("DIERR",1,"TEXT",1)=New values are invalid because they create a
  duplicate Key 'A' for the SAMPLE file.
MYERRORS("DIERR","E",740,1)=
```

Error Codes Returned

740 A duplicate key is produced by a field's new value.

742 A value for a field in a key is being deleted.

744 Not all fields in a key have a value.

Details and Features

Possible IENS The only placeholder the IENS in the FDA can contain is the '+' for records not yet added to the database. You cannot use the '?' or '?+' placeholders since the Key Validator will not attempt to lookup an entry to obtain existing values for a key. (See the Database Server Introduction for details of the IENS; see UPDATE^DIE for description of placeholders.)

UPDATE^DIE( ): Updater

This procedure adds new entries in files or subfiles. The caller uses a standard FDA structure to specify the field values of the new entries. The caller should restrict each Updater call to one logical entry, possibly made up of multiple physical entries. The record numbers for the new entries are returned in an array; the caller may assign their own record numbers to new entries by presetting the array. Any appropriate indexing and auditing automatically occurs for the new record.

Although the Updater can safely add entries to top-level files and to subfiles within those same new entries, there is one caution. If the subfile contains an INPUT transform that assumes the existence of the parent record, the developer should make two separate Updater calls, first to add the parents, then to add the children.

This procedure includes some elementary filing capabilities to permit the adding of required identifiers and key values at the time new records are created. It also includes elementary finding capabilities to facilitate the identification of top-level entries to which subentries are being added. For full filing and finding capabilities beyond the scope of adding new records, programmers should use the Filer (FILE^DIE) or Finder (FIND^DIC). If you are filing data in existing records and you know the record numbers, use the Filer instead of the Updater.

Format

```
UPDATE^DIE(FLAGS,FDA_ROOT,IEN_ROOT,MSG_ROOT)
```

Input Parameters

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

E **E**xternal 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 If a file has a primary key, the primary **K**ey fields, not the .01 field, are used for lookup for Finding and LAYGO Finding nodes.

S The Updater **S**aves the FDA instead of killing it at the end.

U Don't check key integrity. (CAUTION: If this flag is used, the UPDATE^DIE call may result in records that contain null key fields or records with duplicate keys. It is the programmer's responsibility to ensure that the database is not left in a state in which the integrity of keys is violated.)

FDA_ROOT (Required) The name of the root of a VA FileMan Data Array, which describes the entries to add to the database. The Updater accepts Adding Nodes, Filing Nodes, Finding Nodes, and LAYGO Finding Nodes in its FDAs. See Details and Features in this section for a description of the format of the array named by the FDA parameter.

IEN_ROOT (Optional) The name of the Internal Entry Number Array (or IEN Array). This should be a closed root. This array has two functions:

1) Requesting Record Numbers for New Entries

The application can set nodes in the IEN Array to direct the Updater to use specific record numbers for specific new records. These nodes should have a single subscript equal to the sequence number in the IENS subscript of the FDA entry and a value equal to the desired record number.

For example, if the application sets the IEN_ROOT parameter to ORIEN, and sets ORIEN(1)=1701, the Updater will try to assign record number 1701 to the new record denoted by the "+1" value in the FDA subscripts.

This feature also affects LAYGO Finding nodes. When these nodes result in adding a new record, the Updater will check the IEN Array to see if the application wants to place the new record at a specific record number. When LAYGO Finding nodes result in a successful lookup, the IEN Array node passed in by the application is changed to the record number of the record found.

If the application sets an entry in the IEN Array for a Finding node, the Updater will ignore it (actually, it will overwrite it when it finds the record number for that node).

This feature is meaningless for Filing nodes since they have no sequence numbers.

Unlike FDA_ROOT, IEN_ROOT is optional, both partially and as a whole. The Updater will pick the next available record numbers for any new records not listed by sequence number in the IEN Array. If the IEN Array is empty or if the IEN_ROOT is not passed, the Updater will pick all the new record numbers.

2) Locating Feedback on What the Updater Did

As the Updater decodes and processes the sequence numbers, it gradually converts them into genuine record numbers (see Output). The IEN Array named by the IEN_ROOT parameter is where this feedback will be given. Those sequence numbers not already assigned by the application will be filled in by the Updater (or sometimes replaced, in the case of LAYGO Finding nodes).

MSG_ROOT (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

IEN Array As the Updater assigns record numbers to the records described in the FDA, it sets up nodes in the IEN Array to indicate how it decoded the sequence numbers. See Details and Features for more information on sequence numbers. This lets the application find out what was done with the various nodes in the FDA.

The meaning of IEN Array entries varies depending on the type of node the sequence number came from. For example, the significance of an IEN Array entry of ORIEN(3) = 1701 depends on which type of node in the FDA the sequence number 3 came from.

For Adding Node sequence numbers, the value in the IEN Array

indicates the record number of the new record. If our example came from an Adding Node, such as FDA(19,"+3,",.01) ="ZTMDQ", it means the new record was assigned the record number 1701.

For Finding Node sequence numbers, the value indicates at which record number the value was found. If our example came from a Finding Node, such as FDA(19,"?3,",.01) ="ZTMDQ", it means a call to $$FIND1^DIC found record number 1701 based on a lookup value of "ZTMDQ".

For LAYGO Finding sequence numbers, an extra zero-node equal to ? or + identifies whether the entry was found (?) or added (+). If our example came from a LAYGO Finding Node, such as FDA(19,"?+3,",.01)="ZTMDQ", an extra node of ORIEN(3,0)="?" means ZTMDQ was found, whereas ORIEN(3,0)="+" means it was added.

By the time the Updater finishes processing an FDA, every sequence number will be listed with a value in the IEN Array (some set by the application as input for new record numbers and the rest set by the Updater).

If the IEN_ROOT parameter was not passed, the IEN Array is not returned.

Example

The following example illustrates the use of this call to create a new record in a top-level file. In this case, a new option is being added at a specified record number. Notice the triggered 9 on the 0-node and the triggered "U" node:

```
>S FDA(42,19,"+1,",.01)="ZZ FDA TEST NAME"

>S FDA(42,19,"+1,",1)="ZZ Toad Test Menu Text"

>S FDAIEN(1)=2067642283

>D UPDATE^DIE("","FDA(42)","FDAIEN")

>D ^%G

Global ^DIC(19,2067642283
    DIC(19,2067642283
^DIC(19,2067642283,0) = ZZ FDA TEST NAME^ZZ Toad Test Menu Text^^^9
^DIC(19,2067642283,"U") = ZZ FDA TEST MENU TEXT
```

Error Codes Returned

110 The record is currently locked.

111 The File Header Node is currently locked.

120 Error occurred during execution of a VA FileMan hook.

202 An input parameter is missing or not valid.

205 The File and IENS represent different subfile levels.

301 The passed flags are unknown or inconsistent.

302 Entry already exists.

304 The IENS lacks a final comma.

307 The IENS has an empty comma-piece.

308 The IENS is syntactically incorrect.

310 The IENS conflicts with the rest of the FDA.

311 The new record lacks some required identifiers.

330 The value is not valid.

351 FDA Node has a bad IENS.

352 The new record lacks a .01 field.

401 The file does not exist.

402 The global root is missing or not valid.

403 The file lacks a header node.

405 Entries in file cannot be edited.

406 The file has no .01 field definition.

407 A word-processing field is not a file.

408 The file lacks a name.

501 The file does not contain that field.

502 The field has a corrupted definition.

510 The data type cannot be determined.

520 That kind of field cannot be processed by this utility.

601 The entry does not exist.

602 The entry is not available for editing.

603 The entry lacks a required field.

630 The field value is not valid.

701 The value is not valid for that field.

703 The value cannot be found in the file.

712 The value in that field cannot be deleted.

730 The value is not valid according to the DD definition.

740 New values are invalid because they would create a duplicate key.

742 Deletion was attempted on a key field.

744 A key field was not assigned a value.

746 The K flag was used, but no primary key fields were provided in the FDA for Finding and LAYGO Finding nodes.

The Updater may also return any error returned by:

- $$FIND1^DIC
- FILE^DIE

Details and Features

Adding Adding Nodes let applications create new entries in a file. In the place of the actual IENS subscript for the new record in the FDA array, the application instead uses a unique value consisting of a + followed by a positive number.

"+#" will ALWAYS add without regard to duplication.

Thus, for example, an FDA of "FDA(42)" might be accompanied by the following array:

```
FDA(42,19,"+1,",.01)="NAME OF OPTION"
FDA(42,19,"+1,",1)="MENU TEXT OF NEW OPTION"
FDA(42,19.01,"+2,+1,",.01)=45
FDA(42,19.01,"+2,+1,",2)="TM"
FDA(42,19.01,"+3,+1,",.01)=408
```

The FDA_ROOT value directs the Updater to the FDA(42) array, whose format instructs the Updater to add one new entry to the Option file and two new entries to the Menu multiple of that entry.

NOTE: The sequence number for each new entry to be added to a file or subfile must be unique throughout the FDA.

Adding—Identifiers and Keys

The FDA for a new record MUST include the .01 field, all of the required identifiers, and all key fields. If any of these needed fields is missing, the entire FDA transaction fails; none of the entries is added if any one lacks required data.

Filing

Filing Nodes let the application file new data under existing entries. This may be necessary to complete a logical record addition. Any FDA node whose IENS subscript consists solely of record numbers and commas is considered a Filing Node. If you know all of the record numbers, (that is, if all of the nodes in your FDA are Filing Nodes), you should use the Filer instead of the Updater to file the data.

For example, FDA(42,19,"408,",1)="NEW MENU TEXT" instructs the Updater to update field 1 of record 408, so no actual record creation takes place as a result of this node.

Finding

Finding Nodes let applications work with existing entries for which the application does not yet have a record number. Instead of +#, the application uses the notation ?# to stand in for an unknown record number. The sequence number that follows the ? must be unique throughout the FDA.

Every FDA of this type must include an FDA node for the .01 field, or, if the K flag is passed, nodes for at least one field in the primary key. The value of this FDA node is used to perform a lookup on the file. It must match only one entry in that file;

ambiguity or failure to find a match is an error condition. The record number found will then be used for this FDA entry.

For example the following FDA adds a new menu item to the ZTMMGR menu and changes the menu's text:

```
FDA(42,19,"?1,",.01)="ZTMMGR"
FDA(42,19,"?1,",1)="New Menu Text"
FDA(42,19.01,"+2,?1,",.01)=45
FDA(42,19.01,"+2,?1,",2)="TM"
```

In this example, the Updater first uses the value ZTMMGR in a lookup to find the record number that replaces ?1. It then adds a new entry to subfile 19.01 under that entry, and changes the menu text of the option to "New Menu Text". The first node shown is a Finding Node that specifies the value of the .01 field to be used for lookup. The next node specifies a new value for field 1, the menu's text. The last two nodes are Adding Nodes that specify the values for fields .01 and 2 of the new menu item.

When the E flag is used, the .01 Finding node can equal any valid input value for the Lookup. For example, to pick based on a set of codes where WA stands for WASHINGTON, when using the E flag, you may enter WASH.

However, when the E flag is NOT used, the .01 Finding node must equal an internal value, though the special lookup values—space-bar and accent grave (`) concatenated with the IEN—will still work.. So for example, a .01 Finding node equal to WASH would return an error in the above scenario if the E flag were not passed. To succeed, the .01 Finding node would need to equal WA, the internal value.

LAYGO Finding

LAYGO Finding Nodes let the application refer to entries that may or may not already exist. If they do exist, the Updater finds and uses their record numbers. If not, the Updater adds the entries. The IENS notation used to stand in for these entries is ?+#. # is a unique positive number which acts as a placeholder until an actual internal entry number can be produced by the Updater.

For example, this call expects to find the option ZTMMGR, but adds it if it's missing:

```
FDA(42,19,"?+1,",.01)="ZTMMGR"
FDA(42,19.01,"+2,?+1,",.01)=45
FDA(42,19.01,"+2,?+1,",2)="TM"
```

The IEN Array node for this entry includes an extra zero node equal to ? or + to identify whether the entry was found or added. For example, if the entry for the previous example was found, the IEN Array node for this FDA might look like this:

```
IEN(1)=388
IEN(1,0)="?"
IEN(2)=9
```

All LAYGO Finding Nodes are processed in order after Finding Nodes and before other kinds of nodes.

Like Finding Nodes, .01 LAYGO Finding Nodes must match the format of the overall call: external if the E flag has been passed, internal if not. See the Finding section above for details.

Sequence Numbers

A positive number which acts as a placeholder to identify a record until an actual internal entry number can be created or found by the Updater. This positive number must be unique throughout the FDA array. For example, if "+1," is used in an FDA, you cannot also use "?1," or "?+1".

VAL^DIE( ): Validator

The purpose of the Validator procedure is to take the external form of user input and determine if that value is valid, i.e., if that value can be put into the VA FileMan database. In addition, the Validator converts the user-supplied value into the FileMan internal value when necessary. It is this internal value that is stored. If the Validator determines that the value passed is invalid, an up-arrow (^) is returned.

Word processing and computed fields cannot be validated. The .01 field of a multiple must be input using FILE = subfile number and FIELD = .01.

Optionally, the Validator does the following:

- Returns the resolved external value of the data.
- Returns help text for invalid values.
- Loads the internal value into the FileMan Data Array (FDA) to prepare for a later Filer call.

Format

```
VAL^DIE(FILE,IENS,FIELD,FLAGS,VALUE,.RESULT,FDA_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number.

IENS (Required) Standard IENS indicating internal entry numbers.

FIELD (Required) Field number for which data is being validated.

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

- **E** **E**xternal value is returned in RESULT(0).
- **F** **F**DA node is set for valid data in array identified by FDA_ROOT.
- **H** **H**elp (single ?) is returned if VALUE is not valid.
- **R** **R**ecord identified by IENS is verified to exist and to be editable. Do not include "R" if there are placeholders in the IENS.

U Don't perform key validation. Without this flag, the data in VALUE is checked to ensure that no duplicate keys are created and that key field values are not deleted.

VALUE (Required) Value to be validated as input by a user. VALUE can take several forms depending on the data type involved; e.g., a partial, unambiguous match for a pointer; any of the supported ways to input dates (such as "TODAY" or "11/3/93").

.RESULT (Required) Local variable which receives output from call. If VALUE is valid, the internal value is returned. If not valid, ^ is returned. If E flag is present, external value is returned in RESULT(0).

NOTE: This array is killed at the beginning of each Validator call.

FDA_ROOT (Optional; required if F flag present) Root of FDA into which internal value is loaded if F flag is present.

MSG_ROOT (Optional) Root into which error, help, and message arrays are put. If this parameter is not passed, these arrays are put into nodes descendent from ^TMP.

Output

See input parameters .RESULT, FDA_ROOT, and MSG_ROOT.

RESULT = internal value or ^ if the passed VALUE is not valid.

RESULT(0) = external value if the passed VALUE is valid and E flag is present.

Example

This example checks the validity of a value for a set of codes field. Note that the flags indicate that the external value should be returned and that a node in the FDA should be built. In this situation a VALUE of "YES" would also have been acceptable and would have resulted in exactly the same output as shown below:

```
>S FILE=16200,FIELD=5,IENS="3,",FLAG="EHFR",VALUE="Y"

>D VAL^DIE(FILE,IENS,FIELD,FLAG,VALUE,.ANSWER,"MYFDA(1)")

>ZW ANSWER
```

```
ANSWER=Y
ANSWER(0)=YES

>ZW MYFDA(1)
MYFDA(1,16200,"3,",5)=Y
```

Error Codes Returned

In addition to codes indicating that the input parameters are incorrect and that the file, field, or entry does not exist, primary error messages include:

120 Error occurred during execution of a FileMan hook.

299 Ambiguous value. (Variable Pointer data type only.)

405 The file is uneditable.

520 The field's data type or INPUT transform is inappropriate.

602 The entry cannot be edited.

701 Value is invalid.

710 The field is uneditable.

712 An inappropriate deletion of a field's value is being attempted.

740 A duplicate key is produced by a field's new value.

742 A value for a field in a key is being deleted.

1610 Help was improperly requested.

Details and Features

What is Validated

The Validator takes the following steps in validating the input data:

Rejects value starting with "?". Help should be requested using HELP^DIE call.

If R flag is sent, verifies that the entry is present and that editing is not blocked because the entry is being archived.

If the field is uneditable, rejects the input if there is already data in the field.

If the passed value is null or "@", signifying data deletion, rejects the input if the field is required, if the field is a key field, or if the tests present in any "DEL" nodes for the field are not passed. For multiples, the deletion of the last subentry in the multiple is rejected if the multiple is required.

Verifies that the value of the field is not DINUMed.

Checks all keys in which the field participates to ensure the new value does not create any duplicate keys.

Passes the value through the field's INPUT transform and executes any screens on pointer, variable pointer, or set of codes fields. For pointer and variable pointer, values that do not yield at least a partial match are rejected (no LAYGO); ambiguous values are rejected (see note below for variable pointers). If these tests are passed, the input value is accepted and the internal value becomes the value resulting in the execution of the INPUT transform or the pointer value resulting from the lookup.

NOTE: No file or field access security checks on either the file or field level are done.

Note for Pointers

The internal entry number of the entry in the pointed-to file that corresponds to the input is returned. If the lookup value partially matches more than one entry in the pointed-to file, the call fails.

Note for Variable Pointers

For variable pointer data types, the VALUE may include the variable pointer PREFIX, MESSAGE, or FILENAME followed by a period (.) before the lookup value. If no particular file is specified in this way, all of the pointed-to files are searched. If the lookup value is not found in any file searched or if more than one match is found in any file(s), the call fails—VALUE is not valid.

Note for Set of Codes

For set of codes data types, VALUE is treated as case insensitive. If the VALUE is ambiguous, the validation fails.

Returning External Values

If the E flag is sent, the Validator returns the external value of VALUE in addition to its internal value. This is returned in RESULT(0). For free text, number and MUMPS data types, the external value is created by passing VALUE through the INPUT transform (if any) and then the OUTPUT transform (if any). For

date/time data types, the external value is the standard FileMan external date/time format. For pointers and variable pointers, the external value is the .01 of the entry in the pointed-to file. For set of codes, the external value is the "translation" of the code.

Validate and File

If you want to validate a set of data and then file the valid data, make a call to FILE^DIE (the Filer) with an E flag passed in the first parameter. The nodes in the FDA identified by the second parameter should be set to the external, unvalidated value used as input to the Validator. Based on this flag, the Filer calls the Validator for each field and only files the valid, internal values. Error messages are returned for the fields that could not be filed.

NOTE: You cannot mix internal and external values in the FDA when calling the Filer.

VALS^DIE( ): Fields Validator

The Fields Validator procedure validates data for a group of fields and converts valid data to internal VA FileMan format. It is intended for use with a set of fields that comprise a logical record; fields from more than one file can be validated by a single call. By default, the integrity of any keys affected by the new values is checked.

The Fields Validator performs the same checks performed by VAL^DIE (see for details).

Format

```
VALS^DIE(FLAGS,FDA_EXT_ROOT,FDA_INT_ROOT,MSG_ROOT)
```

Input Parameters

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

- **R** **R**ecords identified by IENSs in the FDA_EXT are verified to exist and to be editable. (Same as R flag for VAL^DIE.)
- **U** Don't perform key validation. Without this flag, the data in the FDA is checked to ensure that no duplicate keys are created and that key field values are not deleted.

FDA_EXT_ROOT (Required) The root of a standard FDA. This array should contain the external values that you want to validate. This is the input array. See the Database Server Introduction for details of the structure of the FDA.

FDA_INT_ROOT (Required) The root of a standard FDA. This FDA is the output array, and upon return is set equal to the internal values of each validated field. If a field fails validation, its value is set to an up-arrow (^). (**NOTE:** If a field is valid, the corresponding node in the output array is set to the internal value, not an up-arrow (^), even if that field violates key integrity.) See the Database Server Introduction for details of the structure of the FDA

MSG_ROOT (Optional) The 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).

Output

See the description of the FDA_INT_ROOT for an explanation of how internal values are returned to the client application.

If an error occurs in any of the validations, the DIERR variable will be set and appropriate error messages will be returned.

Examples

Example 1

This simple example validates and converts the values for two fields:

```
>S MYFDA("EXT",16997,"1,",1)="SOME TEXT"

>S MYFDA("EXT",16997,"1,",2)="JAN 1, 1996"

>D VALS^DIE("","MYFDA(""EXT"")","MYFDA(""INT"")")

>W $G(DIERR)

>ZW MYFDA("INT")
MYFDA("INT",16997,"1,",1)=SOME TEXT
MYFDA("INT",16997,"1,",2)=2960101
```

Example 2

This example reports that one of the values does not pass validation. Note that the value for the invalid field equals ^ in MYFDAINT.

```
>S MYFDA("EXT",16997,"1,",1)="SOME TEXT"

>S MYFDA("EXT",16997,"1,",2)="JAN 1, 6"

>D VALS^DIE("","MYFDA(""EXT"")","MYFDA(""INT"")")

>W DIERR
1^1
>D ^%G
```

```
Global ^TMP("DIERR",$J
        TMP("DIERR",$J
^TMP("DIERR",610279233,1) = 701
^TMP("DIERR",610279233,1,"PARAM",0) = 4
^TMP("DIERR",610279233,1,"PARAM",3) = JAN 1, 6
^TMP("DIERR",610279233,1,"PARAM","FIELD") = 2
^TMP("DIERR",610279233,1,"PARAM","FILE") = 16997
^TMP("DIERR",610279233,1,"PARAM","IENS") = 1,
^TMP("DIERR",610279233,1,"TEXT",1) = The value 'JAN 1,
6' for field REVERSE DATE FIELD IN KEY in file ZZD
KEYTEST is not valid.
^TMP("DIERR",610279233,"E",701,1) =
Global ^

>ZW MYFDA("INT")
MYFDA("INT",16997,"1,",1)=SOME TEXT
MYFDA("INT",16997,"1,",2)=^
```

Example 3

In this example, the values pass field validation, but an error is returned because they fail the requested key integrity check.

```
>K MYFDA

>S MYFDA("EXT",16997,"1,",1)="TEXT INTO SECOND"

>S MYFDA("EXT",16997,"1,",2)="MAR 4, 1996"

>D VALS^DIE("U","MYFDA(""EXT"")","MYFDA(""INT"")")

>W $G(DIERR)
1^1
>D ^%G

Global ^TMP("DIERR",$J
        TMP("DIERR",$J
^TMP("DIERR",610279233,1) = 740
^TMP("DIERR",610279233,1,"PARAM",0) = 3
^TMP("DIERR",610279233,1,"PARAM","FILE") = 16997
^TMP("DIERR",610279233,1,"PARAM","IENS") = 13,
^TMP("DIERR",610279233,1,"PARAM","KEY") = 34
^TMP("DIERR",610279233,1,"TEXT",1) = New values are invalid
because they create a duplicate Key 'C' for the ZZD KEYTEST file.
^TMP("DIERR",610279233,"E",740,1) =
Global ^

>ZW MYFDA("INT")
MYFDA("INT",16997,"1,",1)=TEXT INTO SECOND
MYFDA("INT",16997,"1,",2)=2960304
```

Error Codes Returned

In addition to codes indicating that the input parameters are incorrect and that the file, field, or entry does not exist, primary error messages include:

120 Error occurred during execution of a FileMan hook.

299 Ambiguous value. (Variable Pointer data type only.)

405 The file is uneditable.

520 The field's data type or INPUT transform is inappropriate.

602 The entry cannot be edited.

701 Value is invalid.

710 The field is uneditable.

712 An inappropriate deletion of a field's value is being attempted.

740 A duplicate key is produced by a field's new value.

742 A value for a field in a key is being deleted.

744 Not all fields in a key have a value.

1610 Help was improperly requested.

Details and Features

Key Integrity Validation Unless the U flag is passed, the internal values produced by the validation of the values passed in the FDA_EXT are checked to make sure that no key's integrity is violated.

WP^DIE( ): Word Processing Filer

This procedure files a single word processing field.

Format

```
WP^DIE(FILE,IENS,FIELD,FLAGS,wp_root,msg_root)
```

Input Parameters

FILE (Required) File or subfile number.

IENS (Required) Standard IENS indicating internal entry numbers.

FIELD (Required) Field number of the word processing field into which data is being filed.

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

- **A** **A**ppend new word processing text to the current word processing data. If this flag is not sent, the current contents of the word processing field are completely erased before the new word processing data is filed.
- **K** Loc**K** the entry or subentry before changing the word processing data.

WP_ROOT (Required) The root of the array that contains the word processing data to be filed. The data must be in nodes descendent from this root. The subscripts of the nodes below the WP_ROOT must be positive numbers. The subscripts do not have to be integers, and there can be gaps in the sequence. The word processing text must be in these nodes or in the 0-node descendent from these nodes. To delete the word processing field, set WP_ROOT equal to "@".

MSG_ROOT (Optional) Root into which errors are put. If this parameter is not passed, these arrays are put into nodes descendent from ^TMP.

Output

The typical result of this call is the updating of the database with new word processing data. If the call fails, an error message is returned either in ^TMP or, if it is passed, descendent from MSG_ROOT.

Example

The following call files the data into Field #4 of File #16200 for record number 606. The entry is locked before filing and the new data is added to any word processing data that is already there.

```
>D WP^DIE(16200,"606,",4,"KA","^TMP($J,""WP"")")
```

In this example, the word processing text must be located at:

```
^TMP($J,"WP",1,0) =Line 1
^TMP($J,"WP",2,0) =Line 2
    ...etc.
```

or at:

```
^TMP($J,"WP",1) =Line 1
^TMP($J,"WP",2) =Line 2
    ...etc.
```

Error Codes Returned

In addition to errors indicating that input parameters are missing or incorrect and that the file, field, or entry does not exist, this procedure can return the following error codes:

110 Lock could not be obtained because the entry was locked.

305 There is no data in the array identified by WP_ROOT.

726 The specified field is not a word processing field.

CLEAN^DILF: Array and Variable Clean-up

This procedure kills the standard message arrays and variables that are produced by VA FileMan.

Format

```
CLEAN^DILF
```

Input Parameters

None

Output

The call kills the following arrays:

```
^TMP("DIERR",$J)
^TMP("DIHELP",$J)
^TMP("DIMSG",$J)
```

The call kills the following variables:

```
DIERR
DIHELP
DIMSG
DUOUT
DIRUT
DIROUT
DTOUT
```

Error Codes Returned

None

$$CREF^DILF( ): Root Converter (Open to Closed Format)

This extrinsic function converts the traditional open-root format to the closed-root format used by subscript indirection. It converts an ending comma to a close parenthesis. If the last character is an open parenthesis, the last character is dropped.

Format

```
$$CREF^DILF(OPEN_ROOT)
```

Input Parameters

OPEN_ROOT (Required) An open root which is a global root ending in either an open parenthesis or a comma.

Example

```
>W $$CREF^DILF("^DIZ(999000,")
^DIZ(999000)
```

DA^DILF( ): DA( ) Creator

This procedure converts an IENS into an array with the structure of a DA() array.

Format

```
DA^DILF(IENS,.DA)
```

Input Parameters

IENS (Required) A string with record and subrecord numbers in IENS format.

.DA (Required) The name of the array which receives the record numbers.

NOTE: This array is cleaned out (killed) before the record numbers are loaded.

Output

An array with the record numbers from the IENS—the array is structured like the traditional VA FileMan DA() array.

Example

```
>S IENS="4,1,2,532,"

>D DA^DILF(IENS,.MYDA)

>ZW MYDA
MYDA=4
MYDA(1)=1
MYDA(2)=2
MYDA(3)=532
```

Error Codes Returned

None

DT^DILF( ): Date Converter

This procedure converts a user-supplied value into VA FileMan's internal date format and (optionally) into the standard FileMan external, readable date format.

Format

```
DT^DILF(FLAGS,IN_DATE,.RESULT,LIMIT,MSG_ROOT)
```

Input Parameters

FLAGS (Optional) Flags to control processing of user input and the type of output returned. Generally, FLAGS is the same as %DT input variable to ^%DT entry point, with the following exceptions: "A" is not allowed and the meaning of "E" is different (see below). The possible values are:

- **E** **E**xternal, readable date returned in zero-node of RESULT.
- **F** **F**uture dates are assumed.
- **N** **N**umeric-only input is not allowed.
- **P** **P**ast dates are assumed.
- **R** **R**equired time input.
- **S** **S**econds will be returned.
- **T** **T**ime input is allowed but not required.
- **X** E**X**act date (with month and day) is required.

IN_DATE (Required) Date input as entered by the user in any of the formats known to FileMan. Also, help based on the FLAGS passed can be requested with a "?".

.RESULT (Required) Local array that receives the internal value of the date/time and, if the E flag is sent, the readable value of the

date. If input is not a valid date, -1 is returned.

LIMIT (Optional) A value equal to a date/time in FileMan internal format or NOW. IN_DATE is accepted only if it is greater than or equal to LIMIT if it is positive, or less than or equal to LIMIT if it is negative. This is equivalent to the %DT(0) variable in the ^%DT call.

MSG_ROOT (Optional) Root into which error, help, and message arrays are put.

Output

Output is returned in the local array passed by reference in the RESULT parameter, shown below:

RESULT Date in internal FileMan format. If input is invalid or if help is requested with a "?", -1 is returned.

RESULT(0) If requested, date in external, readable format. When appropriate, error messages and help text are returned in the standard manner in ^TMP or in MSG_ROOT (if it is specified).

Example

Example 1

Following is an example of one of the many kinds of user inputs that can be processed by this call. Use of the E flag ensures that the readable form of the data is returned in the 0-node as follows:

```
>D DT^DILF("E","T+10",.ANSWER)

>ZW ANSWER
ANSWER=2931219
ANSWER(0)=DEC 19, 1993
```

Example 2

This is an example of a request for help when time is allowed as input:

```
>D DT^DILF("T","?",.ANSWER,"","MYHELP")

>ZW ANSWER
ANSWER=-1
>ZW MYHELP
MYHELP("DIHELP")=10
MYHELP("DIHELP",1)=Examples of Valid Dates:
MYHELP("DIHELP",2)=   JAN 20 1957 or JAN 57 or 1/20/57 or 012057
MYHELP("DIHELP",3)=   T   (for TODAY), T+1 (for TOMORROW), T+2,
   T+7, etc.
MYHELP("DIHELP",4)=T-1 (for YESTERDAY), T-3W (for 3 WEEKS AGO), etc.
MYHELP("DIHELP",5)=If the year is omitted, the computer uses the
   CURRENT YEAR.
MYHELP("DIHELP",6)=You may omit the precise day, as:  JAN, 1957.
MYHELP("DIHELP",7)=
MYHELP("DIHELP",8)=If the date is omitted, the current date is assumed.
MYHELP("DIHELP",9)=Follow the date with a time, such as JAN 20@10,
   T@10AM, 10:30, etc.
MYHELP("DIHELP",10)=You may enter NOON, MIDNIGHT, or NOW to indicate
   the time.
```

Error Codes Returned

In addition to errors indicating that the input parameters are incorrect or missing, the following error code may be returned:

330 Date/time is not acceptable.

Details and Features

Acceptable User Input

This call processes a wide range of formats for dates and times. Example 2 above that shows the response to an IN_DATE of "?" summarizes the acceptable formats. Remember that the allowable values are controlled by the FLAGS sent and by the LIMIT parameter.

Internal Format

The primary use of this call is to transform the date/time passed in the IN_DATE parameter into the format used by FileMan to store values in Date/Time data type fields. That format is "YYYDDMM.HHMMSS" where YYY is the number of

years since 1700. When the E flag is sent to request that the readable form of the data be returned, the format is always "MON dd,yyyy@ hh:mm:ss."

FDA^DILF( ): FDA Loader

This procedure can be used to load data into the FDA. It accepts either the traditional DA( ) array or the IENS for specifying the entry. No validation of VALUE is done.

Format

1. `FDA^DILF(FILE,IENS,FIELD,FlagS,VALUE,FDA_ROOT,MSG_ROOT)`

2. `FDA^DILF(FILE,.DA,FIELD,FlagS,VALUE,FDA_ROOT,MSG_ROOT)`

Input Parameters

FILE (Required) File or subfile number.

.DA (Required for format 2) DA() array containing entry and subentry numbers.

IENS (Required for format 1) Standard IENS indicating internal entry numbers.

FIELD (Required) Field number for which data is being loaded into the FDA.

FLAGS (Optional) Flag to control processing:

- **R** **R**ecord identified by IENS or .DA is verified to exist. Do not use the R FLAG if the IENS or DA() array contain placeholder codes instead of actual record numbers.

VALUE (Required, can be null) Value to which the FDA node will be set. Depending on how the FDA is used, this could be the internal or external value. For word processing fields, this is the root of the array that contains the word processing data. Internal and external values cannot be mixed in a single FDA.

FDA_ROOT (Required) The root of the FDA in which the new node is loaded.

MSG_ROOT (Optional) Root into which error, help, and message arrays are put. If this parameter is not passed, these arrays are put into nodes descendent from ^TMP.

Output

Successful completion of this call results in the creation of a node descendent from the root passed in FDA_ROOT. The format of the node is:

```
FDA_ROOT(FILE,"IENS",FIELD)=VALUE
```

For more information on the format of the FDA, see the Database Server Introduction.

By default, error messages are returned in ^TMP. If MSG_ROOT is passed, messages are returned there.

Example

This example loads the FDA for the first sub-subentry in the second subentry of entry number 4 for field number 4 in subfile number 16200.32 with a value of "NEW DATA" [the FDA is descended from ^TMP("MYDATA",$J)]:

```
>S FILE=16200.32,IENS="1,2,4,",FIELD=4,VALUE="NEW DATA",ROOT=
  "^TMP(""MYDATA"",$J)"

>D FDA^DILF(FILE,IENS,FIELD,"",VALUE,ROOT)

>D ^%G

Global ^TMP("MYDATA",$J
        TMP("MYDATA",$J
^TMP("MYDATA",736101456,16200.32,"1,2,4,",4) = NEW DATA
```

Error Codes Returned

202 One of the input parameters is not properly specified.

401 The file does not exist.

501 The field does not exist.

601 The entry does not exist.

$$HTML^DILF( ): HTML Encoder/Decoder

This function has two capabilities:

1. It encodes a string that may contain embedded "^" characters according to the rules of HTML so that the "^" characters are replaced with the string "^". As a side effect, "&" characters are encoded as the string "&". Other encodings typical of HTML are not performed by this function, since its focus is on encoding the "^" character used as the delimiter in FileMan databases.

2. This function also decodes an encoded string, restoring its "^" and "&" characters.

Format

```
$$HTML^DILF(STRING,ACTION)
```

Input Parameters

STRING (Required) The string to be either encoded or decoded. Encoding a string that contains no "^" or "&" characters has no effect on the string. Nor does decoding one that lacks "^" and "&" substrings.

ACTION (Optional) Set this parameter to 1 to encode the string, or -1 to decode it. Defaults to 1.

Output

The function evaluates to the encoded or decoded string. If encoding the string makes it overflow the string length limit, it returns error 207. Decoding will never make it overflow.

Error Codes Returned

207 The value is too long to encode into HTML.

$$IENS^DILF( ): IENS Creator

This extrinsic function returns the IENS when passed an array in the traditional DA() structure.

Format

```
$$IENS^DILF(.DA)
```

Input Parameters

.DA (Required) An array with the structure of the traditional VA FileMan DA() array-that is, DA=lowest subfile record number, DA(1)=next highest subfile record number, etc.

Output

A string of record numbers in the IENS format-that is, "DA,DA(1),...DA(n),".

NOTE: The string always ends with a comma (,). If the array passed by reference is empty, a 0 is returned.

Example

```
>S NMSPDA=4,NMSPDA(1)=1,NMSPDA(2)=2,NMSPDA(3)=532

>W $$IENS^DILF(.NMSPDA)
4,1,2,532,
```

Error Codes Returned

None

$$OREF^DILF( ): Root Converter (Closed to Open Format)

This extrinsic function converts a closed root to an open root. It converts an ending close parenthesis to a comma.

Format

```
$$OREF^DILF(CLOSED_ROOT)
```

Input Parameter

CLOSED_ROOT (Required) A closed root, which is a global root ending in a close parenthesis.

Example

```
>W $$OREF^DILF("^DIZ(999000)")
^DIZ(999000,
```

$$VALUE1^DILF( ): FDA Value Retriever (Single)

This extrinsic function returns the value associated with a particular file and field in a standard FDA. Only a single value is returned. If there is more than one node in the FDA array for the same field, the first value encountered by this function is returned. Use the VALUES^DILF call if you want more than one value returned.

Format

```
$$VALUE1^DILF(FILE,FIELD,FDA_ROOT)
```

Input Parameters

FILE	(Required) File or subfile number.
FIELD	(Required) Field number for which data is being requested.
FDA_ROOT	(Required) The root of the FDA from which data is being requested.

Output

This function returns the value for the specified file and field that is stored in the FDA identified by FDA_ROOT. If the field is a word processing field, only the root at which word processing data is stored is returned. No IENS information is returned. If more than one value is associated with a particular field (for example, in a subfile), only a single value is returned.

If there is no node in the FDA for a particular field, an '^' is returned. If the node has a null value, null is returned.

Example

```
>ZW MYFDA
MYFDA("DATA",16200,"33,",4)=FREE TEXT DATA
MYFDA("DATA",16200.04,"1,33,",1)=16
MYFDA("DATA",16200.04,"2,33,",1)=45

>W $$VALUE1^DILF(16200,4,"MYFDA(""DATA"")")
FREE TEXT DATA
```

Error Codes Returned

None

VALUES^DILF( ): FDA Values Retriever

This procedure returns values from an FDA for a specified field. The IENS associated with a particular value is also returned. Use $$VALUE1^DILF if you want the single value associated with a particular file and field in a standard FDA.

Format

```
VALUES^DILF(FILE,FIELD,FDA_ROOT,.RESULT)
```

Input Parameters

FILE	(Required) File or subfile number.
FIELD	(Required) Field number for which data is being requested.
FDA_ROOT	(Required) The root of the FDA from which data is being requested.
.RESULT	(Required) Local array that receives output from the call. The array is killed at the beginning of each call. See the next section below, Output, for the structure of the array.

Output

See the .RESULT input parameter.

The output from the call is returned in the array identified by RESULT. Its structure is:

RESULT	Number of values found for the specified field. If no node exists in the FDA for the field, RESULT=0
RESULT(seq#)	Value for a particular instance of the field. Seq# is an integer starting with 1 that identifies the particular value

RESULT(seq#,"IENS") The IENS of the entry or subentry with the value in RESULT(seq#)

Example

```
>ZW MYFDA
MYFDA("DATA",16200,"33,",4)=FREE TEXT DATA
MYFDA("DATA",16200.04,"1,33,",1)=16
MYFDA("DATA",16200.04,"2,33,",1)=45

>D VALUES^DILF(16200.04,1,"MYFDA(""DATA"")",.MYVALUES)

>ZW MYVALUES
MYVALUES=2
MYVALUES(1)=16
MYVALUES(1,"IENS")=1,33,
MYVALUES(2)=45
MYVALUES(2,"IENS")=2,33,
```

Error Codes Returned

None

$$EXTERNAL^DILFD( ): Converter to External

This extrinsic function converts any internal value to its external format. It decodes codes, makes FileMan dates readable, and follows pointer or variable pointer chains to resolve their values. OUTPUT transforms are applied to their fields. For more information about how FileMan handles OUTPUT transforms and pointers, read this function's Details and Features.

Format

```
$$EXTERNAL^DILFD(FILE,FIELD,FLAGS,INTERNAL,MSG_ROOT)
```

Input Parameters

FILE (Required) The number of the file or subfile that contains the field that describes the internal value passed in.

FIELD (Required) The number of the field that describes the internal value passed in.

FLAGS (Optional) To control processing.

A single-character code that explains how to handle OUTPUT transforms found along pointer chains. The default describes how fields not found along a pointer chain are always handled, regardless of whether a flag is passed. See Details and Features in this section for definition and explanation of pointer chains

The default, if no flag is passed, is the way this function generally handles OUTPUT transforms. If a field has an OUTPUT transform, the transform is applied to the internal value of the field and FileMan does not process the value further. This means it is the responsibility of the OUTPUT transform to resolve codes, transform dates, and follow pointer or variable pointer chains to their destination.

.The default handling of pointer chains, therefore, is to follow the chain either until the last field is found, at which point the field is transformed according to its data type, or until a field with an OUTPUT transform is found, at which point FileMan

applies the OUTPUT transform to the field where it is found and quits. The possible values are:

F If the **F**irst field in a pointer chain has an OUTPUT transform, apply the transform to that first field and quit. Ignore any other OUTPUT transforms found along the pointer chain. With the exception of this function, FileMan regularly handles OUTPUT transforms this way.

L If the **L**ast field in a pointer chain has an OUTPUT transform, apply the transform to that last field and quit. Ignore any other OUTPUT transforms found along the pointer chain.

U **U**se the first OUTPUT transform found on the last field in the pointer chain. Following the pointer chain, watch for OUTPUT transforms. When one is found, remember it, but keep following the pointer chain. When the last field in the chain is reached, apply the remembered transform to that last field.

INTERNAL (Required) The internal value that is to be converted to its external format.

MSG_ROOT (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

This function evaluates to an external format value, as defined by a field in a file in the database. In the event of an error, this function outputs the empty string instead.

Examples

Example 1

```
>W $$EXTERNAL^DILFD(19,4,"","A")
action
```

Example 2

```
>W $$EXTERNAL^DILFD(4.302,.01,"",2940209.0918)
FEB 09, 1994@09:18
```

Example 3

```
>W $$EXTERNAL^DILFD(3.7,.01,"",DUZ)
DOE,JOHN
```

Example 4

```
>W $$EXTERNAL^DILFD(3298428.1,.01,"",1)
11111 1 11111
```

Example 5

```
>W $$EXTERNAL^DILFD(3298428.1,.01,"F",1)
11111 1 11111
```

Example 6

```
>W $$EXTERNAL^DILFD(3298428.1,.01,"L",1)
22222 TOAD 22222
```

Example 7

```
>W $$EXTERNAL^DILFD(3298428.1,.01,"U",1)
11111 TOAD 11111
```

Example 8

```
>W $$EXTERNAL^DILFD(3298428.1,.01,"GGG",1) W DIERR D ^%G
1^1
Global ^TMP("DIERR"
        TMP("DIERR"
^TMP("DIERR",731987397,1) = 301
^TMP("DIERR",731987397,1,"PARAM",0) = 1
^TMP("DIERR",731987397,1,"PARAM",1) = GGG
^TMP("DIERR",731987397,1,"TEXT",1) = The passed flag(s) 'GGG' are
   unknown or inconsistent.
^TMP("DIERR",731987397,"E",301,1) =
```

Error Codes Returned

202 The input parameter is missing or invalid.

301 The passed flag(s) are unknown or inconsistent.

348 The passed value points to a file that does not exist or lacks a Header Node.

401 File # does not exist.

403 File # lacks a Header Node.

404 The Header node of the file lacks a file number.

501 File # does not contain a field.

510 The data type cannot be determined.

537 Corrupted pointer definition.

603 Entry lacks the required Field #.

648 The value points to a file that does not exist or lacks a Header Node.

Details and Features

Data Types

The internal value of a field is the way it is stored in the database. The external value is the way a user expects the field to look. (See also OUTPUT Transforms, below.) FileMan must perform the transformation whenever such a value is displayed. The data types that undergo this process are:

Date/Time The internal value is a numeric code, while the external is readable text. For example, the internal value of 2940214.085938 has an external value of FEB 14,1994@ 08:59:57.

Numeric The internal and external values are identical.

Set of Codes The full external value is decoded from abbreviated internal value. Each set of codes field defines which codes are allowed and what they mean. For example, the internal value of F may have the external value of FEMALE for a certain field.

Free Text The internal and external values are identical.

Word Processing $$EXTERNAL^DILFD does not handle this data type.

Computed This data type does not have an internal value, so $$EXTERNAL^DILFD does not handle this data type.

Pointer to a File The internal value of this field is the internal entry number of one record in the pointed-to file. The external format of a pointer value is the external format of the .01 field of the record identified by the pointer's internal value. The definition of a pointer must always identify the pointed-to file. For example, if 1 is the internal value of a pointer to the State file, then the external value is ALABAMA, because the .01 of the State file is defined as Free Text (needing no transform) and the .01 field of record # 1 in

the State file is ALABAMA.

Variable Pointer Unlike the Pointer data type, the internal value of a variable pointer identifies the pointed-to file. Like the Pointer, the variable pointer's external format is the external value of the .01 field of the pointed-to record. The Prefix.Value notation many users are familiar with is not the external format of a variable pointer; that is merely a user interface convention. For example, the internal value 1;DIC(5, has the external format of ALABAMA (it is the variable pointer equivalent of the previous example).

MUMPS The internal and external values are identical.

OUTPUT Transforms

OUTPUT transforms assume full responsibility for transforming the internal value to its external format. So transforms on sets of codes work with values like F, not FEMALE; those on pointers deal with 1, not ALABAMA; etc. This includes following pointer chains to their conclusions (see immediately below).

Pointer Chains

A pointer chain is a list of one or more pointer fields that point to one another in sequence, the final pointer of which points to a file with a non-pointer .01 field. Thus, for example, if the .01 field of File A points to the State file, that is a pointer chain with one link. If File B points to File A, that makes a pointer chain with two links. Chains can be made up of any mix of pointers and variable pointers. Every field in the chain except the first one must be a .01 field, since pointers point to files, not fields; the first pointer field may or may not be a .01 field.

When FileMan converts a pointer or variable pointer to its external value, it must follow the links to the final field and convert that field to its external value. An OUTPUT transform on a pointer field, therefore, must do the same. The flags available for this function allow developers to try out different ways of handling OUTPUT transforms on pointer fields. These flags only alter this function's behavior, however. The rest of FileMan continues to treat OUTPUT transforms on pointer chains as described under the F flag (under Input Parameters, above).

$$FLDNUM^DILFD( ): Field Number Retriever

This extrinsic function returns a field number when passed a file number and a field name.

Format

```
$$FLDNUM^DILFD(FILE,FIELDNAME)
```

Input Parameters

FILE (Required) The file number of the field's file or subfile.

FIELD NAME (Required) The full name of the field for which you want the number.

Output

The field number of the requested field is returned by this extrinsic function. If the field name does not exist or if there is more than one field with that name, a 0 is returned.

Example

```
>W $$FLDNUM^DILFD(200,"DUZ(0)")
3
```

Error Codes Returned

401 The file does not exist.

501 The file does not contain the field.

505 More than one field has the name.

PRD^DILFD( ): Package Revision Data Initializer

This procedure sets the PACKAGE REVISION DATA attribute for a file. The file Data Dictionary must exist in order to successfully set this attribute.

Format

```
PRD^DILFD(FILE,DATA)
```

Input Parameters

FILE (Required) File or subfile number.

DATA (Required) Free text information, determined by the developer.

Output

A successful call sets the data into the appropriate Data Dictionary location.

Example

The following call sets the PACKAGE REVISION DATA as follows:

```
>D PRD^DILFD(999088,"REVISION #5")

>W $$GET1^DID(999088,"","","PACKAGE REVISION DATA")
REVISION #5
```

Error Codes Returned

None

RECALL^DILFD( ): Recall Record Number

This procedure saves a record number for later retrieval using spacebar recall. While Classic FileMan has automatically performed this procedure for applications in the past, the FileMan DBS lookup calls cannot do so. The decision to perform this procedure can only be made by code that knows its context, that knows whether the selection taking place results from a user's selection or from some silent activity. In addition, FileMan often is inactive when a user selection occurs (such as when a user picks a single entry from a listbox managed by the application). For these reasons, the maintenance of the spacebar recall feature will increasingly be the responsibility of the applications.

Format

```
RECALL^DILFD(FILE,IENS,USER)
```

Input Parameters

FILE (Required) The file or subfile number.

IENS (Required) The IENS that identifies the record selected.

USER (Required) The user number (i.e., DUZ) of the user who made the selection.

Example

```
>D RECALL^DILFD(19,"1,",9) W $G(DIERR) D ^%G

Global ^DISV(9,"^DIC(19,")
        DISV(9,"^DIC(19,")
^DISV(9,"^DIC(19,") = 1
```

Error Codes

202 An input parameter is missing or invalid.

205 The FILE and IENS represent different subfile levels.

401 File # does not exist.

402 The global root is missing or not valid.

$$ROOT^DILFD( ): File Root Resolver

This extrinsic function resolves the file root when passed file or subfile numbers. At the top level of the file $$ROOT returns the global name. When passing a subfile number, $$ROOT uses the IENS to build the root string.

Format

```
$$ROOT^DILFD(FILE,IENS,FLAGS,ERROR_FLAG)
```

Input Parameters

FILE	(Required) File number or subfile number.
IENS	(Required when passing subfile numbers) Standard IENS indicating internal entry number.
FLAGS	(Optional) If set to 1 (true), returns a closed root. The default is to return an open root.
ERROR_FLAG	(Optional) If set to 1 (true), processes an error message if error is encountered.

Examples

Example 1

```
>S DIC=$$ROOT^DILFD(999000.07,"1,38,")

>W DIC
^DIZ(999000,38,2,
```

Example 2

```
>S DIC=$$ROOT^DILFD(999000)

>W DIC
^DIZ(999000,
```

Example 3

```
>S CROOT=$$ROOT^DILFD(999000,"",1)

>W CROOT
^DIZ(999000)
```

Error Codes Returned

200 Invalid parameter

205 The File and IENS represent different subfile levels.

$$VFIELD^DILFD( ): Field Verifier

This extrinsic function verifies that a field in a specified file exists.

Format

```
$$VFIELD^DILFD(FILE,FIELD)
```

Input Parameters

FILE (Required) The number of the file or subfile in which the field to be checked exists.

FIELD (Required) The number of the field to be checked.

Output

This Boolean function returns a 1 if the field exists in the specified file and a 0 if it does not exist.

Example

```
>W $$VFIELD^DILFD(200,99999)
0
```

Error Codes Returned

None

$$VFILE^DILFD( ): File Verifier

This extrinsic function verifies that a file exists.

Format

```
$$VFILE^DILFD(FILE)
```

Input Parameters

FILE (Required) The number of the file or subfile that you want to check.

Output

This Boolean extrinsic function returns a 1 if the file exists or a 0 if it does not.

Example

```
>W $$VFILE^DILFD(200)
1
```

Error Codes Returned

None

$$GET1^DIQ( ): Single Data Retriever

This extrinsic function retrieves data from a single field in a file.

Data may be retrieved from any field, including computed or word processing fields, and fields specified using relational syntax. A basic call does not require that any local variables be present and the symbol table is not changed by this utility. However, computed expressions may require certain variables to be present and can change the symbol table because the data retriever does execute Data Dictionary nodes.

The text for word processing fields is returned in a target array. If data exists for word processing fields, this function returns the resolved TARGET_ROOT. Otherwise null is returned.

Format

```
$$GET1^DIQ(FILE,IENS,FIELD,FLAGS,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) A file number or subfile number.

IENS (Required) Standard IENS indicating internal entry numbers.

FIELD (Required) Field number, or field name, or field identified in another file by simple extended pointer (i.e., POINTER:FIELD) relational syntax.

NOTE: You cannot use a variable pointer as part of relational syntax in this parameter (i.e., varpointer:field).

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

I	**I**nternal format is returned. (The default is external.)
Z	**Z**ero node included for word processing fields on target array.

TARGET_ROOT (Required for word processing fields only) The root of an array into which word processing text is copied.

MSG_ROOT (Optional) Closed root into which the error message arrays are put. If this parameter is not passed, the arrays are put into nodes descendent from ^TMP.

Examples

Example 1

Following is an example of retrieving the value from the .01 field of record #1 in file 999000:

```
>W $$GET1^DIQ(999000,"1,",.01)
DOE,JOHN
```

Example 2

Following is an example of retrieving the internally-formatted value from the SEX field of Record #1 in file 999000:

```
>S X=$$GET1^DIQ(999000,"1,","SEX","I")

>W X
M
```

Example 3

Use the SUBTYPE pointer field in file 3.5 to navigate to the Terminal type file and retrieve the DESCRIPTION field as follows:

```
>S X=$$GET1^DIQ(3.5,"55,","SUBTYPE:DESCRIPTION")

>W X
WYSE 85
```

Example 4

Following is an example of retrieving the contents of a word processing field and storing the text in the target array, WP:

```
>S X=$$GET1^DIQ(999000,"1,",12,"","WP")

>ZW

WP(1)=THIS WP LINE 1
WP(2)=WP LINE2
WP(3)=AND SO ON
X=WP
```

Example 5

Retrieve the contents of a word processing field, storing the text in the target array, WP. The format parameter "Z" means the target array is formatted like the nodes of a FileMan Word Processing field. If no data exists, WP is equal to null as follows:

```
>S WP=$$GET1^DIQ(999000,1,12,"Z","WP")  ZW WP

WP=WP
WP(1,0)=THIS WP LINE 1
WP(2,0)=WP LINE2
WP(3,0)=AND SO ON
```

Example 6

Following is an example of retrieving data from a subfile. Here's a partial record entry, number 323, in ^DIZ(999000:

```
^DIZ(999000,323...
     .
     .
^DIZ(999000,323,4,2,1,0) = ^999000.163^1^1
^DIZ(999000,323,4,2,1,1,0) = XXX2M3F.01^XXX2M3F1^XXX2M3F2
^DIZ(999000,323,4,2,1,"B","XXX2M3F.01",1) =
^DIZ(999000,323,4,"B","XXX1",1) =
^DIZ(999000,323,4,"B","XXX2",2) =

>S IENS="1,2,323,"

>W $$GET1^DIQ(999000.163,IENS,2)
XXX2M3F2
```

Error Codes Returned

200 There is an error in one of the variables passed.

202 Missing or invalid input parameter.

301 Flags passed are unknown or incorrect.

309 Either the root of the multiple or the necessary entry numbers are missing.

348 The passed value points to a file that does not exist or lacks a Header Node.

401 The specified file or subfile does not exist.

403 The file lacks a Header Node.

404 The file Header Node lacks a file #.

501 The field name or number does not exist.

505 The field name passed is ambiguous.

510 The data type for the specified field cannot be determined.

520 An incorrect kind of field is being processed.

537 Field has a corrupted pointer definition.

601 The entry does not exist.

602 The entry is not available for editing.

603 A specific entry in a specific file lacks a value for a required field.

648 The value points to a file that does not exist or lacks a Header Node.

GETS^DIQ( ): Data Retriever

This procedure retrieves one or more fields of data from a record or sub-record(s) and places the values in a target array.

Format

```
GETS^DIQ(FILE,IENS,FIELD,FLAGS,TARGET_ROOT,MSG_ROOT)
```

Input Parameters

FILE (Required) File or subfile number.

IENS (Required) Standard IENS indicating internal entry numbers.

FIELD (Required) Can be one of the following:

A single field number

A list of field numbers, separated by semicolons

A range of field numbers, in the form M:N, where M and N are the end points of the inclusive range. All field numbers within this range are retrieved.

* for all fields at the top level (no sub-multiple record).

** for all fields including all fields and data in sub-multiple fields.

Field number of a multiple followed by an * to indicate all fields and records in the sub-multiple for that field.

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

E Returns **E**xternal values in nodes ending with "E".

I Returns **I**nternal values in nodes ending with "I". (Otherwise, external is returned).

N	Does not return **N**ull values.
R	**R**esolves field numbers to field names in target array subscripts.
Z	Word processing fields include **Z**ero nodes.

TARGET_ROOT (Required) The name of a closed root reference.

MSG_ROOT (Optional) The name of a closed root reference that is used to pass error messages.

Output

TARGET_ROOT The output array is in the FDA format, i.e., TARGET_ROOT(FILE,IENS,FIELD)=DATA. WP fields have data descendent from the field nodes in the output array.

Examples

Retrieve the values of all fields for a record.

```
>D GETS^DIQ(999000,"1,","**","","ARRAY")

>ZW
ARRAY(999000,"1,",.01)=TEST1
ARRAY(999000,"1,",1)=OCT 01, 1992
ARRAY(999000,"1,",2)=YES
ARRAY(999000,"1,",3)=1
ARRAY(999000,"1,",4)=DTM-PC
ARRAY(999000,"1,",5)=SUPPORTED
ARRAY(999000,"1,",6)=S Y="SET Y=TO THIS"
ARRAY(999000,"1,",8)=AUDIT,Z
ARRAY(999000,"1,",9)=ACCESS,Z
ARRAY(999000,"1,",10)=GRP,Z
ARRAY(999000,"1,",11)=DESCRIP,Z
ARRAY(999000,"1,",12)=ARRAY(999000,"1,",12)
ARRAY(999000,"1,",12,1)=THIS WP LINE 1
ARRAY(999000,"1,",12,2)=WP LINE2
ARRAY(999000,"1,",12,3)=AND SO ON
ARRAY(999000,"1,",13)=LASTNAME,FIRST
ARRAY(999000.07,"1,1,",.01)=TEST1 ONE
ARRAY(999000.07,"1,1,",1)=
ARRAY(999000.07,"2,1,",.01)=TEST1 TWO
```

```
ARRAY(999000.07,"2,1,",1)=
ARRAY(999000.07,"3,1,",.01)=TEST1 THREE
ARRAY(999000.07,"3,1,",1)=
ARRAY(999000.07,"4,1,",.01)=TEST1 FOUR
ARRAY(999000.07,"4,1,",1)=MUMPS
```

Retrieve the values of all fields for a record, excluding multiples.

```
>D GETS^DIQ(999000,"1,","*","","ARRAY1")

>ZW
ARRAY1(999000,"1,",.01)=TEST1
ARRAY1(999000,"1,",1)=OCT 01, 1992
ARRAY1(999000,"1,",2)=YES
ARRAY1(999000,"1,",3)=1
ARRAY1(999000,"1,",4)=DTM-PC
ARRAY1(999000,"1,",5)=SUPPORTED
ARRAY1(999000,"1,",6)=S Y="SET Y=TO THIS"
ARRAY1(999000,"1,",8)=AUDIT,Z
ARRAY1(999000,"1,",9)=ACCESS,Z
ARRAY1(999000,"1,",10)=GRP,Z
ARRAY1(999000,"1,",11)=DESCRIP,Z
ARRAY1(999000,"1,",12)=ARRAY(999000,"1,",12)
ARRAY1(999000,"1,",12,1)=THIS WP LINE 1
ARRAY1(999000,"1,",12,2)=WP LINE2
ARRAY1(999000,"1,",12,3)=AND SO ON
ARRAY1(999000,"1,",13)=LASTNAME,FIRST
```

Retrieve both internal and external values of three specific fields for a record.

```
>D GETS^DIQ(999000,"1,",".01;3;5","IE","ARRAY3")

>ZW
ARRAY3(999000,"1,",.01,"E")=TEST1
ARRAY3(999000,"1,",.01,"I")=TEST1
ARRAY3(999000,"1,",3,"E")=1
ARRAY3(999000,"1,",3,"I")=1
ARRAY3(999000,"1,",5,"E")=SUPPORTED
ARRAY3(999000,"1,",5,"I")=
```

Retrieve both internal and external values for a range of fields in a record.

```
>D GETS^DIQ(999000,"1,",".01:6","IE","ARRAY4")

>ZW
ARRAY4(999000,"1,",.01,"E")=TEST1
ARRAY4(999000,"1,",.01,"I")=TEST1
ARRAY4(999000,"1,",1,"E")=OCT 01, 1992
ARRAY4(999000,"1,",1,"I")=2921001
ARRAY4(999000,"1,",2,"E")=NO
ARRAY4(999000,"1,",2,"I")=0
ARRAY4(999000,"1,",3,"E")=66
ARRAY4(999000,"1,",3,"I")=66
ARRAY4(999000,"1,",4,"E")=DTM-PC
```

```
ARRAY4(999000,"1,",4,"I")=9
ARRAY4(999000,"1,",5,"E")=SUPPORTED
ARRAY4(999000,"1,",5,"I")=
ARRAY4(999000,"1,",6,"E")=S Y="SET Y=TO THIS"
ARRAY4(999000,"1,",6,"I")=S Y="SET Y=TO THIS"
```

Retrieve the values of five specific fields, including all of the values of a multiple field.

```
>D GETS^DIQ(999000,"1,",".01;3;7*;11;13","","ARRAY5")

>ZW
ARRAY5(999000,"1,",.01)=TEST1
ARRAY5(999000,"1,",3)=1
ARRAY5(999000,"1,",11)=DESCRIP,Z
ARRAY5(999000,"1,",13)=LASTNAME,FIRST
ARRAY5(999000.07,"1,1,",.01)=TEST1 ONE
ARRAY5(999000.07,"1,1,",1)=
ARRAY5(999000.07,"2,1,",.01)=TEST1 TWO
ARRAY5(999000.07,"2,1,",1)=
ARRAY5(999000.07,"3,1,",.01)=TEST1 THREE
ARRAY5(999000.07,"3,1,",1)=
ARRAY5(999000.07,"4,1,",.01)=TEST1 FOUR
ARRAY5(999000.07,"4,1,",1)=MUMPS 0S
```

Error Codes Returned

200 There is an error in one of the variables passed.

202 Missing or invalid input parameter.

301 Flags passed are unknown or incorrect.

309 Either the root of the multiple or the necessary entry numbers are missing.

348 The passed value points to a file that does not exist or lacks a Header Node.

401 The specified file or subfile does not exist.

403 The file lacks a Header Node.

404 The file Header Node lacks a file #.

501 The field name or number does not exist.

505 The field name passed is ambiguous.

510 The data type for the specified field cannot be determined.

520 An incorrect kind of field is being processed.

537 Field has a corrupted pointer definition.

601 The entry does not exist.

602 The entry is not available for editing.

603 A specific entry in a specific file lacks a value for a required field.

648 The value points to a file that does not exist or lacks a Header Node.

Part II: ScreenMan

3. ScreenMan Forms

INTRODUCTION

The basic steps to prepare and present screens to the user are:

1. Design the physical layout of the screens and determine data editing rules.
2. Use the Form Editor to create the form.
3. Test the form.
4. Invoke the form from an application.

The ScreenMan Form Editor, described in its own chapter in this manual, provides sophisticated tools for creating new forms and editing existing ones. The Form Editor facilitates the composition process from the initial design through editing and completion. It allows you to place blocks and fields wherever you wish on the screen and later to select and drag them to new positions. In addition to allowing you to experiment with the "look" of the screen, the Form Editor eases the process of positioning pop-up pages, blocks, captions, and edit windows.

See also

- The "Form Editor" chapter in this manual.
- The "ScreenMan API" chapter in this manual, which describes the ScreenMan programmer calls you can use to load a form and to use within a form.
- The ScreenMan Tutorial.

FORM LAYOUT: FORMS AND PAGES

Form Structure

A form is a series of screens that are presented to the user. A form contains one or more pages, a page contains one or more blocks, and a block contains one or more fields.

Structurally, the form is an entry in the FORM file (#.403). The FORM file contains a PAGE multiple, and the PAGE multiple contains a BLOCK multiple. The .01 field of the BLOCK multiple is a pointer to the BLOCK file (#.404). The BLOCK file contains a multiple for fields.

Because of this structure, blocks in the BLOCK file are reusable; that is, the same block can be placed on more than one page and on more than one form.

Each block in the BLOCK file that contains VA FileMan fields has a DD (data dictionary) Number. Each block can contain fields from only one file or subfile, as determined by this DD Number.

Linking Pages of a Form

When a form is first invoked and the user is presented with the first page, conceptually, the user is at the top level of the form. When the user goes to the next or previous pages, the user remains at the top level. Only at this level can the user exit or quit the form or save changes made during the editing session.

When the user opens up a subpage, however, the user has descended a level. At this level, and at lower levels, the user can only close the current page, or issue the Refresh command to repaint the screen; the user cannot exit or quit the form or save any changes.

Pages on a form can be linked together in a variety of ways. The following lists the places where links can be defined:

- **Pages at the same level**
  - The Next Page property of a page
  - The Previous Page property of a page
  - The DDSBR variable in the Branching Logic of a field or in Pre and Post Actions
- **Pages at different levels**
  - The Parent Field property of a page
  - The Subpage Link property of a field
  - The DDSSTACK variable in the Branching Logic of a field

Both the Next Page and Previous Page properties link pages at the same level. The user can go to the next and previous pages by pressing **<PF1><ARROW DOWN>** and **<PF1><ARROW UP>**, respectively. Pages linked via the Next and Previous page links must be regular pages; they cannot be "pop-up" pages. The DDSBR

variable, discussed in the Field Properties section below, can be used to take the user to another page under conditions you specify.

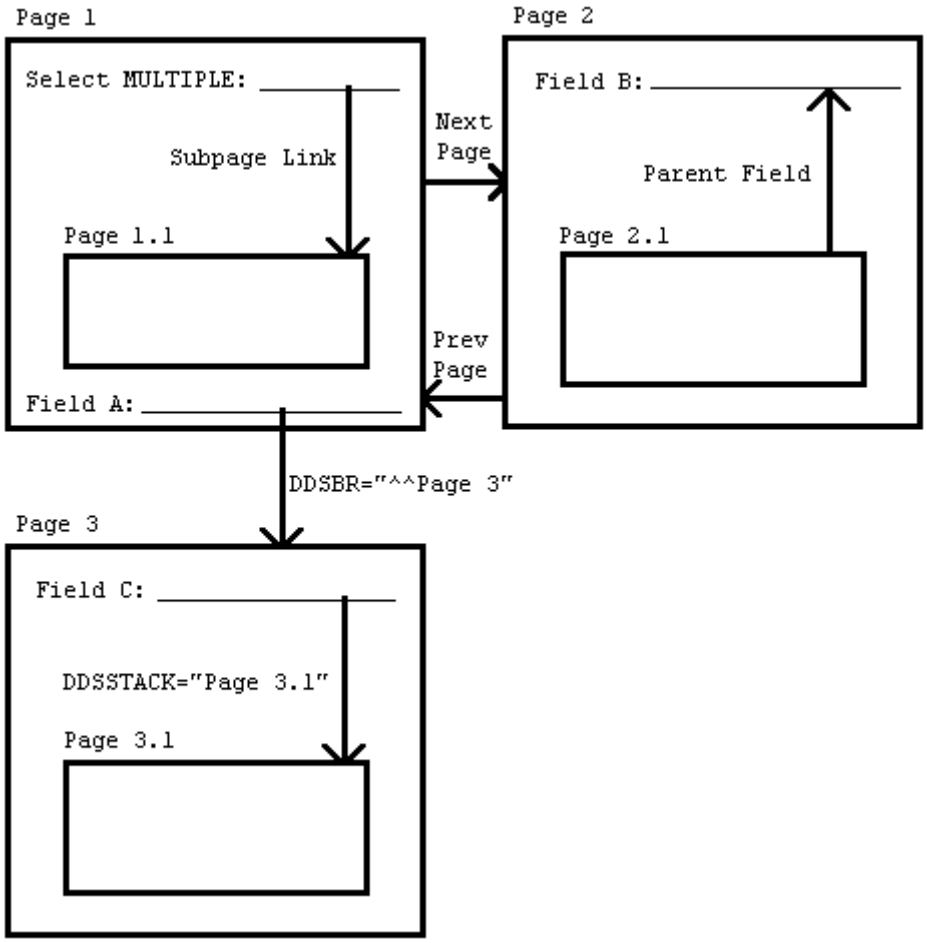
Both the Parent Field and Subpage Link properties allow you to take the user to a subpage at a lower level when the user presses the Enter/Return key at a field on the parent page. The subpage can be either a regular or a "pop-up" page. A "pop-up" page is usually preferable since it gives users a better indication that they have descended a level and must close the subpage to return to the previous level. After the user closes the subpage, ScreenMan automatically returns to the previous level—that is, to the parent page from where the branch occurred.

The difference between the Parent Field property and the Subpage Link property is where the link is defined. The Parent Field property is defined with the subpage and indicates the field from which the branch should occur. The Subpage Link property, on the other hand, is defined with the field and indicates the subpage to which the branch should occur. In a sense, then, the difference between these two properties is the direction of the "pointer." Parent Field points from the subpage to the field, and Subpage Link points from the field to the subpage. Where you choose to define the link is a matter of personal preference. However, the disadvantage of defining the link in the Subpage Link property is that the block on which the field is defined may not be reusable on other forms, since the link points to a specific page on the form.

You must use either the Parent Field or the Subpage Link property to link a multiple field on a form to a subpage that contains the fields within the multiple.

The DDSSTACK variable, discussed in the Field Properties section, can also be used to link a field to a subpage. It behaves just like the Parent Field and Subpage Link properties, but because it is set in M code in the Branching Logic property of a field, DDSSTACK lets you branch conditionally.

The following diagram illustrates the various page links:



FEATURES

Displaying Multiples in Repeating Blocks

You can display more than one subrecord in a multiple simultaneously on the screen. See the Multiples section in the *VA FileMan Getting Started Manual*. You do this by defining a **repeating block**, a block that has a Replication value greater 1. The Replication number defines the number of lines in the scrolling list—or, in other words, the number of times the fields on the block appear on the screen. All fields on the repeating block can occupy no more than one line on the screen. The DD Number of the block corresponds to the subfile number of the multiple.

You should reserve one column to the left of the repeating block for ScreenMan to display the plus sign (+) indicator before the first and last lines of the list.

The following illustration shows two subfields of a multiple displayed in a repeating block:

```
                              A TEST FORM

   NAME: SMITH,JOHN
   DATE: DEC 1,1994

         NAME MULT1                  SET OF CODES
         ----------                  ------------
        +SECOND SUBRECORD            FEBRUARY
         THIRD SUBRECORD             MARCH
         FOURTH SUBRECORD            APRIL
         FIFTH SUBRECORD             MAY
        +SIXTH SUBRECORD             JUNE


Exit     Save     Refresh

Enter a command or '^' followed by a caption to jump to a specific field.

COMMAND:                                Press <PF1>H for help          Insert
```

The subfields are NAME MULT1 and SET OF CODES. The repeating block has a Replication value of 5; therefore, up to five subrecords can be displayed simultaneously. The coordinate of the repeating block corresponds to the position of the first line in the list.

The column headings are defined as caption-only fields on another block that is non-repeating.

The last line in the scrolling list is blank. This is where the user can add a subrecord by entering a new name or jump to a particular entry in the list by entering the name of an existing subrecord. By default, this blank line is positioned in the same column as the first editable field in the repeating block.

The following variables are available in the pre- and post-actions of fields on the repeating block, as well as in the Executable Caption code:

Variables Available in Repeating Blocks

Local Variable	Description
DDSSN	The sequence number in the list of the current subrecord.
DDSLN	The line number in the repeating block on which the cursor is currently resting.

Block Properties that Apply Only to Repeating Blocks

Repeating Block Property	Description
Replication	The number of times the fields defined in this block should be replicated. This number must be greater than 1.
Index	The name of the index that should be used to pick up the subrecords in the multiple. The subrecords will initially be sorted in the order defined by this index. The default Index is B. If the multiple has no B index, or if you wish to display the subentries in record number order, enter !IEN. (Optional)
Initial Position	This is where the cursor should rest when the user first navigates to the repeating block. Possible values are FIRST,

Repeating Block Property	Description
Position	LAST, and NEW, where NEW indicates that the cursor should initially rest on the blank line at the end of the list. The default Initial Position is FIRST. (Optional)
Disallow LAYGO	If set to YES, this prohibits the user from entering new subrecords into the multiple. (Optional)
Field for Selection	This is the field order of the field that defines the column position of the blank line at the end of the list. The default is the first editable field in the block. This is also the field before which ScreenMan prints the plus sign (+) to indicate there are more entries above or below the displayed list. (Optional)

Form-Only Fields

Form-only fields are fields that are defined only on the form. They allow you to request from the user data that is not linked to a FileMan field. You might use a form-only field to control the flow of data input. For example, when the user presses the Enter/Return key at a form-only field, you might branch to a "pop-up" page (window) or branch only if the user enters a certain value. You might also use a form-only field to request data from the user, store the response in a local or global variable, and process the response after the user exits the form.

When you define a form-only field, you specify parameters that look like the VA FileMan Reader (^DIR) parameters. In addition, you can define Save Code, code that is executed when the user issues the "Save" command. You might use the Save Code to store the value entered by the user in local or global variables.

The following chart describes the field properties that pertain only to Form Only fields. See the Reader: ^DIR section for more detailed information about the Reader parameters:

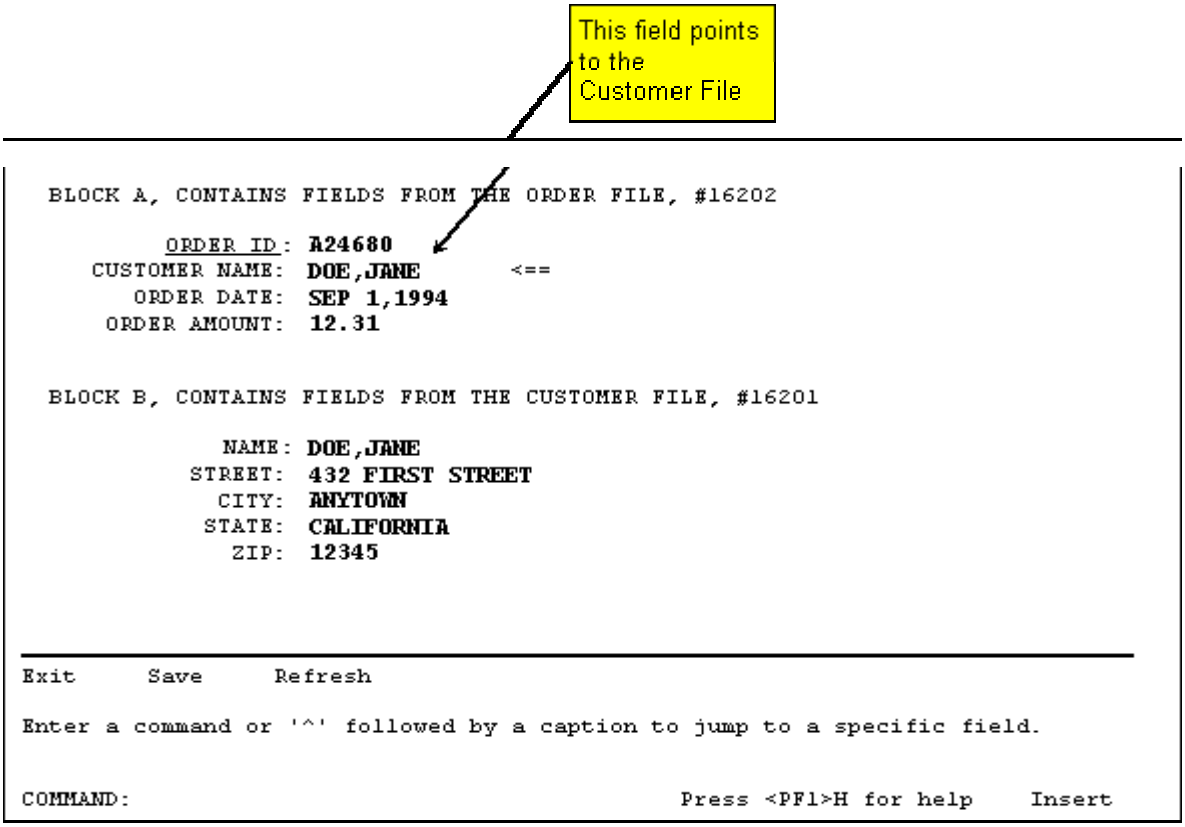
Properties of Form-Only Fields

Form Only Field Property	Description
Read Type	This property defines the type of the form-only field. Valid values are: D = DATE F = FREE TEXT L = LIST OR RANGE N = NUMERIC P = POINTER S = SET OF CODES Y = YES OR NO DD = DATA DICTIONARY
Parameters	This property corresponds to the parameters that can be used in the first ^-piece of the DIR(0) input variable to ^DIR. The "O" parameter has no effect, since the Required property can be used to make a field required.
Qualifiers	This property corresponds to the second ^- piece of the DIR(0) input variable to ^DIR.
Help (WP)	The lines in this word processing field correspond to the nodes in the DIR("?",#) input array to ^DIR.
INPUT Transform	This property corresponds to the third ^-piece of the DIR(0) input variable to ^DIR.
Screen	This is M code that sets the variable DIR("S").
Save Code	This is M code that is executed when the user issues a "Save" command and ScreenMan hasdetected a change to the value of the form-only field.

Relational Navigation: Forward Pointers

On a page of a form, you can place a block that contains fields from a file other than the Primary File of the form. If the file is reached via a forward pointer, you must define a Pointer Link for that block. The syntax of the Pointer Link property is similar to FileMan's relational syntax. When you define the Pointer link, your point of reference is the Primary File of the form.

In the following illustration, the Primary File of the form is the ORDER file (#16202). There are two blocks on the page. Block A contains fields from the ORDER file, and Block B contains fields from the CUSTOMER file (#16201). CUSTOMER NAME in the ORDER file points to the CUSTOMER file.



If CUSTOMER NAME is field #1, the Pointer Link property for Block B can be set to either "CUSTOMER NAME" or 1. The following sections describe in more detail the syntax for the Pointer Link property.

Syntax for Pointer Link—Navigating Via DD Fields

In the valid formats listed below, "Pfield" is a pointer-type field. Both "Pfield" and "Field" can be either field names or field numbers. "Mult_field" is the name or number of a multiple field. "File" is the name or number of a file. A file or field name can be enclosed in quotation marks.

Format	Explanation
Pfield	The Primary File of the form has a field Pfield that points to the file associated with the block. That pointer field determines the record to display in the pointed-to block.
Pfield_1:Pfield_2: ... :Pfield_n	The pointed-to block is reached after relational jumps across many files. Here, Pfield_1 in the Primary File points to File 2 that contains a Pfield_2 that points to File 3, etc. Finally, Pfield_n points to the file associated with the block being defined.
Field;Opt_spec	The value of Field in the Primary File should be used to do a lookup into the file associated with the block. You can control how the lookup is done by using any of the following optional specifiers (Opt_spec): ;I — Use the Internal form of the field value for the lookup ;L — Allow LAYGO ;IX(xref list) — Use specific IndeX(es) in the lookup. (For example ;IX(B^C) specifies that the B and C index should be used.) If the specifier is not used, all indexes starting with the B index are used in the lookup.

Format	Explanation
Field;Opt_spec:File:Pfield_1: Pfield_2: ... :Pfield_n	The pointed-to block is reached after relational jumps across many files. The first jump is accomplished with a lookup into File. See above for an explanation of Opt_spec.
Mult_field_1:Mult_field_2: ... :Mult_field_n:Pfield	The pointed-to block is reached after descending into subfiles of the Primary File and finally a relational jump via a pointer field within a subfile.

Syntax for Pointer Link—Navigating Via Form Only Fields

Form-only fields can also be used to relationally link blocks.

In the formats below, the characters "FO" indicate that a form-only field is being identified. "Pform_only" is a pointer-type form-only field and "Form_only" is a form-only field that is not a pointer. Form_only and Pform_only are three-piece comma-delimited strings that uniquely identify form-only fields on the form. They have the following format:

Format of Form_only: Field_id,Block_id,Page_id
where

- Field_id = Field Order number; or Caption of the field; or Unique Name of the field
- Block_id = Block Order number; or Block Name
- Page_id = Page Number; or Page Name (required only if Block Order number is used to identify the block.)

Valid formats are:

Format	Explanation
FO(Pform_only)	The pointer-type form-only field is a pointer to the file associated with the block being defined. The contents of the form-only field determines the record to display in the

Format	Explanation
	pointed-to file.
FO(Pform_only):Pfield_1: ... Pfield_n	The pointed-to file is reached after relational jumps across many files. Here, the pointer-type form-only field points to File 1 that contains a Pfield_1 that points to File 2, etc. Finally, Pfield_n points to the file associated with the block being defined.
FO(Form_only);Opt_spec	The value of the form-only field is used to do a lookup into the file associated with the block. You can control how the lookup is done by using any of the following optional specifiers (Opt_spec): ;I — Use the Internal form of the field value for the lookup ;L — Allow LAYGO ;IX(xref list) — Use specific IndeX(es) in the lookup. (For example ;IX(B^C) specifies that the B and C index should be used.) If this specifier is not used, all indexes starting with the B index are used in the lookup.
FO(Form_only);Opt_spec:File: Pfield_1: ... Pfield_n	The pointed-to file is reached after relational jumps across many files. The first jump is accomplished with a lookup. See above for an explanation of Opt_spec.

Relational Navigation: Backward Pointers

Records reached via backward pointers appear to the user much like subrecords within a multiple. To display the records in the pointing file, you can define a repeating block that has a DD Number equal to the file number of the pointing file

and an Index property equal to the name of the whole file cross-reference of the pointer field. See the section Displaying Multiples in Repeating Blocks above for more information on how to define repeating blocks.

Computed Fields

ScreenMan computed fields, like form-only fields, are fields that are defined only on the form. You cannot place computed fields from FileMan files on a form because the M code for those fields often directly references data in files, which is outside the context of ScreenMan's transaction.

When you define a ScreenMan computed field, you enter a Computed Expression. The computed expression has the following format:

M code that sets the local variable Y

For example:

```
S:$D(FLAG) Y=$P(MYVAR,",",2)_" "_$P(MYVAR,",")
```

The computed expression can reference data dictionary fields, form-only fields, and computed fields used elsewhere on the form. If the user changes the value of a field used in a computed expression, ScreenMan automatically recalculates and repaints the computed field.

The expression atom that identifies other form elements has a syntax that uses curly braces ({}) as described below.

Referencing Data Dictionary Fields

In the formats below, "Field" is the name or number of a data dictionary field. "Pfield" is the name or number of a pointer-type data dictionary field. "File" is the name or number of a file.

Syntax for Computed Expression Atom That References a DD Field

Format	Explanation
{Field;Opt_spec}	The value of Field is retrieved. An Opt_spec (optional specifier) can be used to retrieve the internal, rather than the external form: ;I — Retrieve the **I**nternal form of the Field value.
{Pfield:Field;Opt_spec}	Pfield is a pointer to a file. The value of Field in that file is retrieved. The Opt_spec value of ;I can be used as described immediately above to retrieve the internal, rather than the external form.
{Field_1;Opt_spec1: File:Field_2;Opt_spec}	Field_1 is not a pointer field. The value of Field_1 is used to do a lookup into File. Field_2 from that file is retrieved. An Opt_spec value of ;I can be used to retrieve the internal rather than the external form. In addition, you can control how the lookup is done by using any of the following optional specifiers for Opt_spec1: ;I — Use the **I**nternal form of the field value for the lookup

Syntax for Computed Expression Atom That References a DD Field

Format	Explanation	
	;IX(xref list)	Use specific **I**nde**X**(es) in the lookup. (For example ;IX(B^C) specifies that the B and C index should be used.) If this specifier is not used, all indexes starting with the B index are used in the lookup.

Referencing Form-Only and Computed Fields

A computed expression atom can also reference form-only fields and computed fields used on the form.

In the formats below, the syntax is similar to the that for referencing data dictionary fields, except that "FO(Form_only)" is used instead of "Field." "Form_only" is a three-piece comma-delimited string that identifies a form-only or computed field. See the description of Syntax for Pointer Link—Navigating Via Form Only Fields for a description of the format of "Form_only."

Syntax for Computed Expression Atom That References a Form Only Field

Format	Explanation	
{FO(Form_only);Opt_spec}	The value of Form_only is retrieved. An Opt_spec (optional specifier) can be used to retrieve the internal, rather than the external form.	
	;I	Retrieve the **I**nternal form of the Form_only field.
{FO(Pform_only: Field;Opt_spec}	Pfield_order is a pointer-type form-only field that points to a file. The value of Field in that file is retrieved.	

Syntax for Computed Expression Atom That References a Form Only Field

Format	Explanation
	The Opt_spec value of ;I can be used as described above to retrieve the internal, rather than the external form.
{FO(Form_only);Opt_spec1 :File:Field_2;Opt_spec}	Form_only is a form-only field that is not a pointer-type form-only field. The value of Field_order is used to do a lookup into File. Field_2 from that file is retrieved. An Opt_spec value of ;I can be used to retrieve the internal, rather than the external form. In addition, you can control how the lookup is done by using any of the following optional specifiers for Opt_spec1: ;I — Use the **I**nternal form of the field value for the lookup ;IX(xref list) — Use specific **I**nde**X**(es) in the lookup. (For example ;IX(B^C) specifies that the B and C index should be used.) If this specifier is not used, all indexes starting with the B index are used in the lookup.

Examples

```
S Y="The value is: "_{NUMERIC}
S:$D(var)#2 Y="The value is: "_{NUMERIC}
S Y={LAST NAME}_","_{FIRST NAME}
S Y={NAME}_" "_{NAME:SSN}
S Y={FO(PRICE)}*1.085
S Y={FO(NAME):NEW PERSON:SSN}
```

The DDSBR Variable

ScreenMan allows you to branch the user to a field under conditions you specify. You can do this by defining M code in the Branching Logic, Pre Action, Post Action, and Post Action on Change properties at the field level, and at the Data Validation property at the form level. The M code can set the local variable DDSBR to a value that defines the location of the field to which you wish to take the user.

DDSBR has the following format:

```
DDSBR=Field id^Block id^Page id
```

where,

- Field id = Field Order number; or Caption of the field; or Unique Name of the field
- Block id = Block Order number; or Block Name
- Page id = Page Number; or Page Name

For example,

```
S:X="Y" DDSBR="FIELD 1^BLOCK 1^PAGE 2"
```

would take the user to the field with unique name or caption "FIELD 1" on the block named "BLOCK 1" on the page named "PAGE 2", if the internal value of the field equals "Y".

ScreenMan assumes values for any of the ^-pieces of DDSBR that are empty, as illustrated in the following table:

Assumptions When Pieces of DDSBR Are Null

If DDSBR is set to:	ScreenMan assumes:
Field id	Current block and current page
Field id^Block id	Current page
Field id^^Page id	Current block
^Block id	Field with lowest Field Order, current page

Assumptions When Pieces of DDSBR Are Null

If DDSBR is set to:	ScreenMan assumes:
^Block id^Page id	Field with lowest Field Order
^^Page id	Field with lowest Field Order, Block with lowest Block Order

To branch the user to the Command Line, DDSBR takes the following format:

```
S DDSBR="COM"
```

The DDSSTACK Variable

The DDSSTACK variable can be set only in the Branching Logic property of a field. It can be used to branch users to another page when they press the Enter/Return key at the field. After the user closes the page defined in DDSSTACK, ScreenMan takes the user to the parent page, to the field immediately following the field from which the branch occurred.

Set DDSSTACK equal to a Page Number or Page Name. For example:

```
S:X="Y" DDSSTACK="Page 1.1"
```

would take the user to Page 1.1 if the internal value of the field is "Y" and the user presses the Enter/Return key at the field. When Page 1.1 is closed, the user returns to the parent page, to the field immediately following the field that contained the Branching Logic.

Note that ScreenMan provides another way to achieve this kind of "branch and return" behavior. You can link a field to a subpage by defining a Subpage Link for the field or by defining a Parent Field for the subpage. The Subpage Link and Parent Field methods, however, do not allow branching conditionally.

Data Filing (When Is It Performed?)

With some important exceptions, the database is unaffected during a ScreenMan editing session. Changes are filed only at the user's request.

However, there are two situations in which changes to the database are made immediately:

- When an entry is deleted from a file or subfile.
- When an entry is added to a file or subfile.

When the user attempts to delete an entry, ScreenMan issues a warning that deletions are immediate and permanent. Even if the user quits the form without saving the changes, the entry is not restored to the database.

Similarly, when the user adds an entry to a file or subfile, that entry is immediately added to the database. The entry is added with values for the .01 field and all required identifiers. After the entry is added, however, changes made to the data for that entry are part of ScreenMan's transaction and are filed only at the user's request. Also, in contrast to deletions of entries, if the user subsequently quits the form without saving changes, entries added during the editing session are deleted.

Because of this, you should consider cross-references that can cause an overall state change when the user adds an entry and when ScreenMan subsequently deletes the entry. Triggers, bulletins, and MUMPS-type cross-references can cause irreversible events to occur. Therefore, when you design cross-references for the .01 field and the required identifiers for entries the user may add or delete during an editing session, it is best to ensure that the kill logic can undo the effects of the set logic.

FORM PROPERTY REFERENCE

Form Properties

Form Name

(Required) This is the .01 field of the FORM file (#.403). Form Names should be namespaced.

Title

The Title property can be used by the form designer to help identify a form. It is cross-referenced and need not be unique. ScreenMan does not automatically display the Title to the user, but the form designer can choose to create a caption-only field that displays the Title to the user.

Pre Action and Post Action

The Pre Action property is M code that is executed when the form is first invoked, before any of the pages are loaded and displayed. The Post Action property is M code that is executed before ScreenMan returns to the calling application.

Data Validation

This is M code that is executed when the user attempts to save changes to the form. If the code sets DDSERROR, the user is unable to save changes. If the code sets DDSBR, the user is taken to the specified field.

In addition to $$GET^DDSVAL, PUT^DDSVAL, $$GET^DDSVALF, PUT^DDSVALF, and HLP^DDSUTL, you can use MSG^DDSUTL to print on a separate screen messages to the user about the validity of the data.

Post Save

This is M code that is executed when the user saves changes. It is executed only if all data is valid, and after all data has been filed.

Record Selection Page

If you define a Record Selection Page, the user can select another entry in the file, and, if LAYGO is allowed, add another entry into the file without exiting the form. The Record Selection Page should be a pop-up page, and the first field on that page is a pointer-type form-only field. The file specified in the Qualifiers property of the form-only field should be the Primary File of the form. The Record Selection Page property should be set equal to the Page Number of the Record Selection Page.

The user can open the Record Selection Page by pressing <PF1>L. After the user selects a record and closes the Record Selection Page, the data for the selected record is displayed.

Page Properties

Page Number

(Required) The Page Number uniquely identifies a page on a form. You can use this number to refer to the page in ScreenMan functions and utilities. ScreenMan does not display Page Numbers to the user.

Page Name

(Required) Like the Page Number, you can use the Page Name to refer to a page in ScreenMan functions and utilities. ScreenMan displays the Page Name to the user if, during an attempt to file data, ScreenMan finds required fields with null values. ScreenMan uses the Caption of the field and the Page Name to inform the user of the location of the required field.

Page Coordinate and Lower Right Coordinate

(Required) The Page Coordinate property defines the location of the top left corner of the page on the screen. The format of a coordinate is: Row,Column. Regular pages normally have a Page Coordinate of "1,1". They do not have a Lower Right Coordinate.

The Page Coordinate of "pop-up" pages defines the position of the top left corner of the border of the "pop-up" page. "Pop-up" pages must have a Lower Right Coordinate, which defines the position of the bottom right corner of the border of the "pop-up" page.

All blocks on the page are positioned relative to the page on which they are defined. If a page is moved—that is, if the Page Coordinate is changed—all blocks and all fields on that page move with it.

The Form Editor described below greatly simplifies the process of assigning coordinates to "pop-up" pages. It allows you to drag and drop an entire "pop-up" page and to drag and drop the lower right corner to resize the "pop-up" page. It therefore eliminates the need for you to manually enter Page Coordinate values.

Header Block

A page can have one Header Block that contains uneditable information. ScreenMan always positions the Header Block at coordinate "1,1" relative to the page.

There is no need to place header blocks on a page. Display blocks with a coordinate of "1,1" provide the same functionality as header blocks.

Is This a "Pop-Up" Page?

If this page is a "pop-up" page, rather than a regular page, set this property to YES.

Next Page and Previous Page

The Next Page and Previous Page properties are set to Page Numbers. The user can go to the next and previous pages by pressing **<PF1><ARROW DOWN>** and **<PF1><ARROW UP>**, respectively. The user can also ^-jump to fields on other pages that are linked to the current page via the next and previous page links. See the section Linking Pages of a Form above for more information about using these properties to link pages.

ScreenMan also uses the Next Page property during filing. Starting with the first page displayed to the user, ScreenMan follows the Next Page links, loads those pages not already loaded, and checks that all required fields on those pages have values. If any of the required fields have null values, no filing occurs; otherwise, ScreenMan files the data along with any defaults.

Parent Field

This property can be used to link a subpage to a field on the parent page. Parent Field has the following format:

Field id,Block id,Page id

where

- Field id = Field Order number; or Caption of the field; or Unique Name of the field
- Block id = Block Order number; or Block Name
- Page id = Page Number; or Page Name

For example:

```
ZZFIELD 1,ZZBLOCK 1,ZZPAGE 1
```

identifies the field with Caption or Unique Name "ZZFIELD 1," on the block named "ZZBLOCK 1," on the page named "ZZPAGE 1".

Pre Action and Post Action

ScreenMan executes the M code in the Pre Action property when the user reaches the page and the M code in the Post Action property when the user leaves the page.

Block Properties

Block properties are stored in two locations:

- Block properties stored in the FORM File
- Block Properties stored in the BLOCK File

BLOCK PROPERTIES STORED IN THE FORM FILE

Since these properties are stored in the Form file, they apply only as it is used on a particular form.

Block Name

(Required) This is the .01 field of the block multiple of the page multiple of the Form file (#.403). This field is a pointer to Block file (#.404).

Block Order

(Required) The Block Order determines the order in which users traverse fields on a page when they press **<PF1><PF4>** to go to the next block or press the Enter/Return key to move from the last field on one block to the first field on the next. When the user first reaches a page, ScreenMan places the user on the block with the lowest Block Order number.

Type of Block

(Required) The Type of Block property can be either "DISPLAY" or "EDIT."

EDIT blocks allow fields to be changed by the user and DISPLAY blocks allow fields to be displayed but *not* changed by the user. Adding an EDIT block to a form enables the editing of any data dictionary fields placed on the EDIT-type block. Fields in a DISPLAY block are read-only.

When you first add a block to a form, you enter the properties for the block, including the type of block it is. You can also edit the properties of the block later. See the *ScreenMan Tutorial for Developers Manual*, particularly Lessons 1-2 (DISPLAY blocks) and 1-7 (EDIT blocks) for more information. This manual is available both in Adobe Acrobat PDF and HTML formats on the VA FileMan Home Page.

Block Coordinate

(Required) This property defines the location of the block. The format of a coordinate is: Row,Column.

A Block Coordinate is defined relative to the page on which the block is defined. A Block Coordinate of "1,1", for example, corresponds to the top left corner of the page. If a page is moved to a new position—that is, if it is given a new coordinate—all blocks on the page move with it.

Pointer Link

This property is used if the fields displayed in this block are reached through a relational jump from the primary file of the form. The Pointer Link is a relational expression that describes this jump. See the section "Relational Navigation" in the *VA FileMan Advanced User Manual* for more information.

Pre Action and Post Action

The Pre Action property is M code that is executed whenever the user reaches this block. The Post Action property is M code that is executed whenever the user navigates away from this block. Since these two properties are stored in the Form file, they apply to the block only as it is used on this form.

Replication, Index, Initial Position, Disallow LAYGO, Field for Selection

These properties pertain only to repeating blocks. See the section Displaying Multiples in Repeating Blocks above for more information about these properties.

BLOCK PROPERTIES STORED IN THE BLOCK FILE

Since these properties are stored in the BLOCK file, they are part of the definition of the block itself. These properties apply to the block wherever it is used.

Name

(Required) This is the .01 field of the BLOCK file (#.404). Block Names should be namespaced.

DD Number

This is the data dictionary number of the file or subfile that contains the fields that are placed on this block. A block can contain fields from only one file or subfile.

Disable Navigation

If you set this property to "YES," navigation within the block is disabled. When navigation is disabled, users cannot ^-jump to other fields, they cannot ^-jump to the Command Line, and the <ARROW UP>, <ARROW DOWN>, <Tab>, and <PF4> keys traverse the fields in the same order when pressing the Enter/Return key—that is, in the order established by the Field Order property of the fields. The <PF1> S, <PF1> E, <PF1> Q, and <PF1> C key sequences cannot be used if this property is set to YES.

If you set this property to OUTOK, navigation is disabled, but the user can ^-jump to the Command Line and use <PF1> S, <PF1> E, <PF1> Q, and <PF1> C.

Pre Action and Post Action

The Pre Action property is M code that is executed whenever the user reaches this block. The Post Action property is M code that is executed whenever the user navigates away from this block. Since these two properties are stored in the BLOCK file, they apply to the block as it is used on any page of any form.

Field Properties

Field Type

(Required) Four different types of fields can be defined on a block:

- Caption only
- Data dictionary
- Form only
- Computed

Caption-only fields are for displaying text on the screen. They have no data portion associated with them.

Data dictionary fields correspond to fields in a file. They have a data portion, which is the value of the field, and an optional caption portion, which serves to identify the data on the screen for the user.

Form-only fields are fields that are defined only on the form and are not tied to a field in a FileMan file. See the section Form Only Fields for more information about this field type.

Computed fields, like form-only fields, are fields that are defined only on the form. You cannot place computed fields from FileMan files on a form. The computed expression is defined on the form and can be based on other fields on the form. Users cannot navigate to computed fields.

Field Order

(Required) The Field Order number determines the order in which users traverse the fields in the block as they press the Enter/Return key. Field Order is the .01 field of the Field multiple of the BLOCK file.

Field

(Required for Data Dictionary type fields.) The Field property applies only to data dictionary type fields. It identifies a field in a VA FileMan file or subfile. The DD Number of the block identifies the file or subfile that contains the field.

Unique Name

You can optionally give fields on a block a Unique Name. Unique Names are never seen by the user. They can be used to identify fields in some of the ScreenMan utilities, such as PUT^DDSVAL and $$GET^DDSVAL, and in the computed expressions of computed fields. No two fields on a block can have the same Unique Name.

Caption, Executable Caption, and Suppress Colon After Caption

A Caption is uneditable text that appears on the screen. Captions of data dictionary, form-only, and computed fields serve to identify for the user the data portion of the fields. Captions of these types of fields are automatically followed by a colon, unless the Suppress Colon After Caption property is set to YES. Captions of caption-only fields have no associated data element and are not automatically followed by a colon.

If you want the text of a caption to be determined whenever the page is painted, you can enter M code as an Executable Caption. The code should set the local variable Y equal to the text you want displayed. A field with an Executable Caption must have "!M" as a Caption.

Default and Executable Default

You can assign a Default to a Data Dictionary or form-only type field on a form. ScreenMan presents the Default value to the user if the value of the field is null when the page on which the field is located is first displayed. Since ScreenMan validates the Default, it must be valid, unambiguous, and in external form; otherwise, it is not used.

If the field is a multiple field, you can assign one of the following defaults:

Valid Default Values for Multiple Fields

Default	Subrecord displayed
FIRST	The subrecord with the lowest IEN
LAST	The subrecord with the highest IEN

Valid Default Values for Multiple Fields

Default	Subrecord displayed
Subrecord number	The subrecord with the specified IEN

Here, the characters "FIRST" and "LAST" are keywords that ScreenMan interprets as the subrecords with the lowest and highest IENs, respectively.

If the value of the default should be determined at the time the page is first presented to the user, you can enter M code as an Executable Default. The code should set the local variable Y equal to a valid, unambiguous value in external form. If the default in Y is invalid, it is not used. A field with an Executable Default must have "!M" as a Default.

Data Length

(Required for all field types, except caption-only.) Data Length defines the length of the edit window for the data portion of fields. Ideally, the Data Length should equal the maximum length of the external form of the data—the form displayed to the user.

The Data Length of a word processing field need only be 1, since the contents of the field are not displayed in the edit window. A Data Length of 1 gives the cursor a place to rest when the user navigates to the field. When the user presses the Enter/Return key at the field, control is passed to the user's Preferred Editor, where the text can be displayed and, if allowed, edited.

If you define a Data Length smaller than a field's maximum size, ScreenMan still provides two ways for the user to see the entire value of the field:

- Since the edit window is a "scrolling window," text outside the confines of the edit window can scroll in, as text in the window scrolls out.
- When the cursor is within an edit window, the user can press **<PF1>Z** to invoke the "zoom editor." An area opens in the Command Area where up to 245 characters can be seen and edited at once.

NOTE: You cannot define an edit window that wraps around to a second line. In addition, the edit window must not extend into the rightmost column of the screen. This space must be left blank so that the cursor has a place to rest beyond the last character of the data value. You must, therefore, never define a data length that

causes the edit window to extend beyond the next to last column of the screen—that is, column 79 on an 80-column display.

Caption and Data Coordinates

(Required if a field has a caption or data portion.) Caption and Data Coordinates define the location of fields on the screen and are relative to the coordinate of the block on which they are defined. The format of coordinate is: Row,Column. The coordinate "1,1" for example, corresponds to the block's top left corner—the first column on the first row of the block.

The Form Editor described below greatly simplifies the process of assigning coordinates to captions and data. It allows you to drag and drop fields on the screen, and thus eliminates the need to explicitly assign values to the coordinate properties.

Right Justify

Set the Right Justify property to YES to display the data for the field to be right-justified.

Required

You can make any non-multiple field on a form required. Making a field required on a form does not affect the definition of the field in the data dictionary. You need not make a field required that is already required by its data definition. The captions of required fields are underlined for easy identification. See the section Data Filing for more information on how ScreenMan checks required fields before filing.

Before filing, ScreenMan checks that:

- Required fields on all pages that can be accessed via the next and previous page links have values, even if you have not accessed those pages during the editing session.
- If you have accessed any subpages, required fields in those subpages must also have values.

If any required field is empty, the user cannot file any data changes. When they attempt to file, ScreenMan displays a list of those fields that require values.

NOTE: "Pop-up" pages NOT accessed during the editing session will not be checked for Required fields.

You can change the Required property on the fly while a form is running by making a call to REQ^DDSUTL. See that section for more information on this call.

Display Group

Display group helps users resolve ambiguity when they attempt to ^-jump to a field that has a caption that is not unique. If more than one field has the same caption, when users try to ^-jump to a field with that caption, they are presented with a list of fields to choose from. The text in the Display Group property is displayed in parentheses after the caption to help the user identify the correct field.

For example, if two fields have the caption "NAME:", but one of those fields has a Display Group "Next of Kin," when users enter ^NAME, they will be asked to choose between "NAME" and "NAME (Next of Kin)".

Disable Editing and Disallow LAYGO

If you set the Disable Editing property to "YES," users cannot navigate to the field, unless the field is a word processing field. If the field is defined as uneditable in the Data Dictionary, users cannot navigate to it—even if the field's value is null. See the section Word Processing Fields in the *VA FileMan Getting Started Manual* for information about uneditable word processing fields.

If you set the Disable Editing property to "REACHABLE," users can navigate to the field, but they cannot change the value. You might want to make an uneditable field reachable if, for example, you want to attach branching logic to that field, to take the user to another page when they press the Enter/Return key. You might also want to make an uneditable field reachable if the data value cannot fit in an edit window. Then the user can navigate to the field and see the entire contents of the field, either by scrolling the data in the edit window or by invoking the "zoom editor."

You can change Disable Editing property on the fly while a form is running by making a call to UNED^DDSUTL. See this section for more information on this call.

NOTE: Fields on display blocks are always uneditable. On display blocks, users can navigate only to multiple and word processing fields.

Disabling editing for multiple fields has no meaning. However, you can prevent users from adding new entries into a multiple by setting the Disallow LAYGO property to YES. Multiple fields on display blocks automatically prohibit LAYGO.

Data Validation

ScreenMan uses the definition of a field to automatically validate values input by the user. You can use the Data Validation property to validate the value even further. Data Validation is M code that is executed after the user enters a new value for a field and after the automatic validation that ScreenMan normally performs. If the code sets the variable DDSERROR, ScreenMan rejects the value. You might also want to ring the bell and make a call to HLP^DDSUTL to display a message to the user that indicates the reason the value was rejected.

Subpage Link

A subpage can be linked to a parent page by the Subpage Link property. The Subpage Link must be equal to the Page Number of the subpage.

Branching Logic, Pre Action, Post Action, and Post Action on Change

These properties contain M code that is executed at the following times:

Descriptions of Field-Level Pre and Post Actions

Property	Executed
Branching Logic	When the user presses the Enter/Return key at the field
Pre Action	Right before the user lands on the field
Post Action	When the user leaves the field
Post Action on Change	When the user leaves the field, and only if the user changed the value of the field

The code in the Branching Logic, Pre Action, Post Action, Post Action on Change, and Data Validation at the field level can rely on the following variables:

Variables Available in Field-Level Pre and Post Actions

Local Variable	Description
X	The current internal value of the field
DDSEXT	The current external value of the field
DDSOLD	The previous internal value of the field

The Post Action and Post Action on Change are not executed when the user times out at a field, enters an ^ to go to the Command Area, or ^-jumps to another field.

SCREENMAN MENU OPTIONS

The ScreenMan menu options are found on a submenu of the Other Options menu:

```
Select OPTION: OTHER OPTIONS
Select OTHER OPTION: SCREENMAN
Select SCREENMAN OPTION: ?
 Answer with SCREENMAN OPTION NUMBER, or NAME
Choose from:
   1               EDIT/CREATE A FORM
   2               RUN A FORM
   3               DELETE A FORM
   4               PURGE UNUSED BLOCKS

Select SCREENMAN OPTION:
```

Edit/Create a Form

The first option, Edit/Create a Form, invokes the Form Editor, the screen-oriented utility for editing and building ScreenMan forms. The Form Editor is described in detail in the "ScreenMan Form Editor" chapter in this manual.

Run a Form

Instead of setting up input variables and making a call to ^DDS, you can use the second option on the ScreenMan menu to run a form shown below:

```
Select SCREENMAN OPTION: RUN A FORM

MODIFY SCREEN TEMPLATE FOR WHAT FILE: ZZEZ SCREENDOC


Select FORM: ZZEZ DOC  ZZEZ DOC

Enter number of first page: 1// <RET>  Select ZZEZ SCREENDOC NAME:
TURING,ALAN
M.          AL
```

You are asked to select a file, a form, an initial page, and a record. This option cannot run a form used to a edit a subfile directly.

Delete a Form

You can use the third option on the ScreenMan menu, shown below, to delete a form from the Form file, and any or all of the blocks used on that form from the BLOCK file.

```
Select SCREENMAN OPTION: DELETE A FORM

MODIFY SCREEN TEMPLATE FOR WHAT FILE: ZZEZ SCREENDOC


Select FORM to delete: ZZTEST DOC  ZZTEST DOC
                         #55      02/16/91  User #14    File #16500
```

Once you've selected a file and form to delete, a short report is printed that lists all blocks used on the form, as illustrated below:

```
BLOCKS USED ON FORM "ZZTEST DOC" (IEN #55)
Internal                                  Used on
Entry Number   Block Name                 Other Forms?   Deletable?
------------   ----------                 ------------   ----------
178            ZZTEST DOC HDR1                NO            YES
179            ZZTEST DOC1                    NO            YES
180            ZZTEST DOC2                    NO            YES
181            ZZTEST DOC3                    NO            YES
182            ZZTEST DOC HDR3                NO            YES
```

The first column lists the internal entry numbers of the blocks used on the form, and the second column lists the names of the blocks. The last two columns indicate whether the blocks are used on other forms and whether you can delete those blocks from the BLOCK file. Only those blocks that are not used on other forms can be deleted.

You are then asked whether you want to delete the blocks used on the form from the BLOCK file.

```
Delete all deletable blocks used on form ZZTEST DOC
from the BLOCK file (Y/N)? YES// ?

  Enter 'Y' to delete blocks used on form
    ZZTEST DOC from the BLOCK file.
    (Only blocks not used on other forms can be deleted.)

  Enter 'N' to delete the form but not the blocks.

Delete all deletable blocks used on form ZZTEST DOC
from the BLOCK file (Y/N)? YES//<RET>
```

If you answer "NO," the form will be deleted from the Form file, but none of the blocks used on the form will be deleted. Note that if you choose not to delete a block, and that block is not used on any form, the only way to delete the block is to run the Purge Unused Blocks option described below.

If you answer "YES," you are asked whether you want to delete those blocks without confirmation.

```
Delete blocks without prompting (Y/N)? NO// ?

  Enter 'Y' to delete blocks from the BLOCK file
    without confirmation.

  Enter 'N' to confirm each delete.

Delete blocks without prompting (Y/N)? NO//
```

If you answer "YES," all blocks used on the form that are not used on any form will be deleted. If you answer "NO," you will be prompted before any block is deleted. This gives you a chance to delete only specific blocks.

```
Continue (Y/N)? NO// YES

Deleting form ZZTEST DOC (IEN #55) ...

ZZTEST DOC HDR1               Delete (Y/N)? NO// YES
ZZTEST DOC1                   Delete (Y/N)? NO// YES
ZZTEST DOC2                   Delete (Y/N)? NO// YES
ZZTEST DOC3                   Delete (Y/N)? NO// YES
ZZTEST DOC HDR3               Delete (Y/N)? NO// YES

DONE!
```

Purge Unused Blocks

You can use the fourth option on the ScreenMan menu to delete any or all of the unused blocks from the BLOCK file that are associated with a specific file.

```
Select OPTION: OTHER OPTIONS
Select OTHER OPTION: SCREENMAN
Select SCREENMAN OPTION: PURGE UNUSED BLOCKS

PURGE UNUSED BLOCKS FROM WHAT FILE: ZZEZ SCREENDOC
```

Once you've selected a file, a short report is printed that lists the blocks that aren't used on any forms:

```
UNUSED BLOCKS ASSOCIATED WITH FILE ZZEZ SCREENDOC (#16500)

Internal
Entry Number   Block Name
------------   ----------
72             ZZZEE EDIT3
178            ZZTEST DOC1
179            ZZTEST DOC2
180            ZZTEST DOC3
181            ZZTEST DOC HDR3
```

You are then asked whether to delete the blocks without confirmation:

```
    Delete all unused blocks without prompting (Y/N)? NO// <RET>
```

If you answer "YES," all unused block are deleted. If you answer "NO," you will be prompted before any block is deleted. This gives you a chance to delete only specific blocks.

```
Continue (Y/N)? NO// YES

ZZZEE EDIT3                   Delete (Y/N)? NO// YES
ZZTEST DOC1                   Delete (Y/N)? NO// YES
ZZTEST DOC2                   Delete (Y/N)? NO// YES
ZZTEST DOC3                   Delete (Y/N)? NO// YES
ZZTEST DOC HDR3               Delete (Y/N)? NO// YES

DONE!
```

CALLABLE ROUTINES

ScreenMan provides a number of callable routines. Many of these routines can be called from the various form properties that execute M code. These callable routines are described in the "ScreenMan API" chapter in this manual.

PROGRAMMER MODE UTILITIES

^DDGF

You can use this routine to invoke the Form Editor from programmer mode.

NOTE: You can also reach the Form Editor through the FileMan menu options. On the FileMan menu, select Other Options, ScreenMan, and Edit/Create a Form. The Form Editor is described in detail in the "ScreenMan Form Editor" chapter in this manual.

CLONE^DDS

You can use this entry point to make a copy of a form. All blocks used on the form are copied and a new form that uses the new blocks is created.

In the following illustration, CLONE^DDS is used to make a copy of the XUEDIT CHARACTERISTICS form of the NEW PERSON file:

```
>D CLONE^DDS


CLONE FORM FROM WHAT FILE: NEW PERSON

Select FORM to clone: ??

Choose from:
   XUEDIT CHARACTERISTICS        #1     12/06/90         File #200
   XUEXISTING USER               #2     12/12/90         File #200

Select FORM to clone: XUEDIT CHARACTERISTICS  XUEDIT CHARACTERISTICS

                                 #1     12/06/90         File #200
```

Once you have selected a form to clone, a report that lists the blocks used on the form is printed:

```
  BLOCKS USED ON FORM "XUEDIT CHARACTERISTICS" (IEN #1)

  Internal
  Entry Number   Block Name
  ------------   ----------
  1              XUEDIT CHARACTERISTICS HDR
  2              XUEDIT CHARACTERISTICS

Enter RETURN to continue or '^' to exit: <RET>
```

You must assign names to the new form and blocks you are creating. If the original form and blocks are namespaced—that is, start with the same set of characters—you can choose to give the new form and blocks the same name, but with the namespace replaced with another set of characters. Then, when you are asked to enter new names, names that have the namespace replaced with the set of characters are displayed as defaults:

```
The new form and blocks must be given unique names.

Give the new form and blocks the same names as the original,
but a different namespace? YES// <RET>
```

```
Original namespace: XU
     New namespace: ZZ

Enter names for the new form and blocks.

Original form name: XUEDIT CHARACTERISTICS
     New form name: ZZEDIT CHARACTERISTICS


Original block name: XUEDIT CHARACTERISTICS HDR
     New block name: ZZEDIT CHARACTERISTICS HDR

Original block name: XUEDIT CHARACTERISTICS
     New block name: ZZEDIT CHARACTERISTICS
```

After you have given names to the new form and blocks, you are ready to clone the form as follows:

```
Ready to clone form? YES

Creating new blocks ...

  ZZEDIT CHARACTERISTICS HDR       #71
  ZZEDIT CHARACTERISTICS           #72

Creating new form ...
  ZZEDIT CHARACTERISTICS           #36

Repointing to new blocks ...
Reindexing new form ...


DONE!
>
```

NOTE: Be sure to check the properties of the cloned form and blocks for namespaced variables, block references, etc., that may need to be modified manually.

PRINT^DDS

You can use this entry point to print a form. PRINT^DDS prints the properties of the form and the properties of all the blocks used on that form.

```
>D PRINT^DDS

Select FORM: ZZSAMPLE <RET>        (Nov 16, 1994)   User #3   File 16201

Start each page of the form on a new page? Yes// NO
DEVICE: HOME// ;;9999 <RET>  DECSERVER

FORM LISTING - ZZSAMPLE (#38)
FILE: ZZTEST (#16201)                NOV 16, 1994  13:29    PAGE 1
-----------------------------------------------------------------
         PRIMARY FILE: 16201                   READ ACCESS: @
         DATE CREATED: NOV 16, 1994@08:24     WRITE ACCESS: @
       DATE LAST USED: NOV 16, 1994@08:25          CREATOR: 3

Page    Page
Number  Properties
------  ----------
1       Page 1

PAGE COORDINATE:        1,1

Block  Block
Order  Properties (Form File)
-----  ----------------------
1      ZZSAMPLE (#104)

TYPE OF BLOCK:          EDIT
BLOCK COORDINATE:       1,1

Block Properties (Block File)
-----------------------------
DATA DICTIONARY NUMBER: 16201

Field  Field
Order  Properties
-----  ----------
1      FIELD TYPE:            DATA DICTIONARY FIELD
       CAPTION:               NAME
       FIELD:                 .01
       CAPTION COORDINATE:    1,1
       DATA COORDINATE:       1,7
       DATA LENGTH:           30

2      FIELD TYPE:            DATA DICTIONARY FIELD
       CAPTION:               SET
       FIELD:                 1
       CAPTION COORDINATE:    2,2
       DATA COORDINATE:       2,7
       DATA LENGTH:           10
```

RESET^DDS

If during a call to ^DDS you get a hard error, you can DO RESET^DDS to reset the terminal characteristics, unlock any locked records, clean up some variables in the local symbol table, and remove the temporary data ScreenMan stores in ^TMP. Since RESET^DDS doesn't clean up all local variables, you should do P^DI afterwards to clean up any variables that RESET^DDS missed.

You can also use RESET^DDS if you get a hard error while using the Form Editor.

4. ScreenMan Form Editor

INTRODUCTION

The ScreenMan Form Editor is a screen-oriented tool for creating and editing ScreenMan forms. It allows you to select and drag form elements and edit their properties through a ScreenMan interface. It can run on character-based terminals, such as the DEC VT-100 and Qume QVT-102, if properly defined through the Device Handler.

As you use the Form Editor, it is helpful to have printouts of the data dictionaries of the files containing the fields you will be placing on ScreenMan forms. You will need to know such things as the data dictionary numbers of files and subfiles and the maximum length of the external form of data.

See also

- The "ScreenMan Forms" chapter in this manual.
- The "ScreenMan API" chapter, which describes the ScreenMan programmer calls you can use to load a form and to use from within a form.
- The ScreenMan Tutorial.

INVOKING THE FORM EDITOR

To invoke the Form Editor, perform the following steps from the VA FileMan menu:

```
Select OPTION: OTHER OPTIONS
Select OTHER OPTION: SCREENMAN
Select SCREENMAN OPTION: ?
 Answer with SCREENMAN OPTION NUMBER, or NAME
Choose from:
   1               EDIT/CREATE A FORM
   2               RUN A FORM
   3               DELETE A FORM
   4               PURGE UNUSED BLOCKS

Select SCREENMAN OPTION: EDIT/CREATE A FORM
```

You are asked to select a file:

```
    EDIT/CREATE FORM FOR WHAT FILE:
```

and a form:

```
Select FORM:
```

At the "Select FORM:" prompt, you can either select an existing form to edit or create a new form by entering a new form name.

If you create a new form, the Form Editor automatically creates one page on that form. The new page is given a Page Number of 1, a Page Name of "Page 1", and a Page Coordinate of "1,1".

NOTE: You can also use the programmer mode utility ^DDGF to invoke the Form Editor.

COMMAND SUMMARY

Navigating on the Main Screen and Block Viewer Screen

To move the cursor	Press
Up one line	<ARROW UP>
Down one line	<ARROW DOWN>
Right one column	<ARROW RIGHT>
Left one column	<ARROW LEFT>
One field to the right	<Tab>
One field to the left	Q
Five columns to the right	S
Five columns to the left	A
Top of screen	<PF1><ARROW UP>
Bottom of screen	<PF1><ARROW DOWN>

Right edge of screen	<PF1><ARROW RIGHT>
Left edge of screen	<PF1><ARROW LEFT>

To switch between the Main Screen and the Block Viewer Screen

1. Press **<PF1>V**

Quick Page Navigation

To	Press
Go to the next page	<PF1><PF1><ARROW DOWN>
Go to the previous page	<PF1><PF1><ARROW UP>
Go into a subpage associated with a field	Select the field with <SpaceBar> or <RET>and press <PF1>D

To select a screen element (field caption, field data, or block name)

1. Position the cursor over the element and press **<SpaceBar> or <RET>**.
2. Press **<SpaceBar> or <RET>** again to deselect the element.

To reorder all fields on a block

1. Select the block on the Block Viewer Screen.
2. Press **<PF1>O**.

Moving Screen Elements

To drag a selected element	Press
Up one line	<ARROW UP>
Down one line	<ARROW DOWN>

To drag a selected element	Press
Right one column	<ARROW RIGHT>
Left one column	<ARROW LEFT>
Five columns to the right	<Tab> or S
Five columns to the left	Q or A
Top of screen	<PF1><ARROW UP>
Bottom of screen	<PF1><ARROW DOWN>
Right edge of screen	<PF1><ARROW RIGHT>
Left edge of screen	<PF1><ARROW LEFT>

Adding, Selecting, and Editing

To	Press
Select or create a new form	<PF1>M or <PF2>M
Select another page	<PF1>P
Add a new page	<PF2>P
Add a new block	<PF2>B
Add a new field	<PF2>F
Edit properties of current form	<PF4>M
Edit properties of current page	<PF4>P

To invoke the ScreenMan form to edit field or block properties

1. Select the field or block and press **<PF4>**.

To edit the caption of a field on the Main Screen

1. Position the cursor over the caption and press **<PF3>**.
2. Press the Enter/Return key when finished editing.

To edit the data length of a field on the Main Screen

1. Position the cursor over the underline that represents the data and press **<PF3>**.
2. Press **<ARROW RIGHT>** and **<ARROW LEFT>** to change the length.
3. Press the Enter/Return key when finished.

THE MAIN SCREEN

Below is an example of the Form Editor's **Main Screen**.

The **top** portion of the Main Screen is the **Work Area**. Here you see field captions, as well as underscores representing data fields, for fields that are defined on the blocks of the current page. Each of these items is called a **screen element**. This area of the screen is the one that you control when you display information to the user on a form.

```
            NAME: ____________________________
  STREET ADDRESS: ____________________________
            CITY: ____________________________
           STATE: __
        ZIP CODE: __________

File: ZZFILE NAME (# nnnn)                                    Rn,Cn
Form: ZZFORM NAME
Page: n (ZZName of Page)

<PF1>Q=Quit  <PF1>E=Exit  <PF1>S=Save  <PF1>V=Block Viewer  <PF1>H=Help
```

The **bottom** portion of the screen contains status information, such as the name and number of the file to which the form is attached, the name of the form, and the number and name of the page you are currently editing. The "Rn,Cn" at the lower

right of the screen indicates the current row and column position of the cursor. When a user runs a form, this portion of the screen is occupied by ScreenMan's command area.

Exiting, Quitting, Saving, and Obtaining Help

You can exit from the Form Editor's Main Screen in one of two ways:

1. Press <PF1>E to exit and save any changes you made to field captions, data lengths of fields, block names, and page, block, and field coordinates. These are the properties that are visible on the Form Editor screens.
2. Press <PF1>Q to quit and discard the changes you made to those properties.

You can also save changes without leaving the Form Editor by pressing <PF1>S.

Pressing <PF1>H accesses the Form Editor's online help screens.

General Key Sequences

To	Press
Exit and save changes	<PF1>E
Quit without saving changes	<PF1>Q
Save without exiting	<PF1>S
Bring up help screens	<PF1>H
Move to Block Viewer screen	<PF1>V

THE BLOCK VIEWER SCREEN

To view the blocks on the current page, press **<PF1>V** to go to the **Block Viewer Screen**. The Block Viewer Screen displays the names of the blocks defined on the current page. For example, if the current page contains blocks called ZZBLOCK NAME 1 and ZZBLOCK NAME 2, the Block Viewer Screen looks like this:

```
ZZBLOCK NAME 1




ZZBLOCK NAME 2



File: ZZFILE NAME (#nnnn)                    BLOCK VIEWER          R1,C1
Form: ZZFORM NAME
Page: n (ZZPage Name)

<PF1>V=Main Screen  <PF1>H=Help
```

Like the captions and data fields displayed on the Main Screen, the block names on the Block Viewer are **screen elements**. Notice that on the Block Viewer Screen the words "BLOCK VIEWER" appear in the bottom portion of the screen.

To return to the Main Screen, press **<PF1>V**.

NAVIGATING ON THE FORM EDITOR SCREENS

To move the cursor on the Main Screen and the Block Viewer Screen, you can use the key sequences listed below.

NOTE: You can move the cursor only within the boundaries of the current page, as determined by the page coordinate.

Navigating

To move the cursor	Press
Up one line	<ARROW UP>
Down one line	<ARROW DOWN>
Right one column	<ARROW RIGHT>
Left one column	<ARROW LEFT>
One field to the right	<Tab> or S
One field to the left	Q or A
Five columns to the right	S
Five columns to the left	A
Top of screen	<PF1><ARROW UP>
Bottom of screen	<PF1><ARROW DOWN>
Right edge of screen	<PF1><ARROW RIGHT>
Left edge of screen	<PF1><ARROW LEFT>

GOING TO ANOTHER PAGE

In the Form Editor, you work with one page at a time. The page with which you are currently working is indicated in the status area at the bottom portion of the screen. To go to another page, press **<PF1>P**. The Form Editor asks you to select another page on the form:

```
Select PAGE: n
```

Here you can enter **?** (a single question mark) to get a list of the pages defined on the form. The page you select becomes the current page and the Form Editor displays the fields on that page in the Work Area of the Main Screen.

Shortcut keys also allow you to quickly change the current page. If the current page has a Next Page defined, you can press **<PF1><PF1><ARROW DOWN>** to go to the next page. Similarly, if the current page has a Previous Page defined, you can press **<PF1><PF1><ARROW UP>** to go to the previous page.

If one of the fields on the current page has a subpage associated with it, you can go to that subpage by first selecting the field (press **<SpaceBar> or <RET>** over the caption of that field) and then pressing **<PF1>D**. To close a subpage and return to the page underneath, press **<PF1>C**.

ADDING PAGES, BLOCKS, AND FIELDS

Adding Pages

To add a new page to the form, press **<PF2>P**. The Form Editor asks you to enter the page number of the new page:

```
NEW PAGE NUMBER:
```

Here you must enter a page number that hasn't yet been used on the form. (Press **<PF1>Q** to close this "pop-up" page and abort adding a new page.) Once you've selected a new page number, the Form Editor asks:

```
Are you adding Page n
as a new page on this form?
```

If you answer YES, the Form Editor invokes a ScreenMan form in which you can edit the properties of the new page. (See Editing Page Properties.)

Adding Blocks

To add a new block to the current page, move the cursor to the location on the page where you want the upper left corner of the block positioned, and press **<PF2>B**. The Form Editor asks you for the name of the block you want to add to the current page:

```
Select NEW BLOCK NAME:
```

Here, you can either select an existing block from the BLOCK file, or enter the name of a new block. If you enter the name of a new block (e.g., ZZTEST BLOCK 1), the Form Editor asks you whether you wish to add the block to the BLOCK file:

```
ARE YOU ADDING 'ZZTEST BLOCK 1' AS A NEW BLOCK (THE 36TH)?
```

and whether you want to add the block to the current page of the form:

```
Are you adding ZZTEST BLOCK 1 as a new block on this page?
```

If you answer YES to these questions, the Form Editor invokes a ScreenMan form where you can edit the properties of the new block. (See Editing Block Properties.)

Header Blocks

For backward compatibility, the Form Editor displays and allows you to edit the properties of header blocks already defined on the form. It does not, however, provide a way to add header blocks to a form, since display-type blocks provide the

same functionality as header blocks. Instead of creating a header block on a page, you can create a display-type block with a coordinate of "1,1" relative to the page.

Adding Fields

To add fields to a block on the current page of the form, you must be on the Form Editor's Main Screen. If you are currently on the Block Viewer Screen, press **<PF1>V** to return to the Main Screen. Before you can add fields, at least one block must be defined on the current page.

To add a field, move the cursor to the desired location of the new field and press **<PF2>F**. The Form Editor presents the following dialog:

```
Select BLOCK:
 FIELD ORDER:
  FIELD TYPE:
```

(To close this "pop-up" page and abort adding a new field, press **<PF1>Q**.)

You can change any of the default answers the Form Editor provides. The Form Editor asks you to select a block on which to add the new field. You can select only those blocks that are defined on the current page. The Form Editor also asks you for the Field Order number and the Field Type of the new field.

Once you have filled in all the information in this "pop-up" page, press **<PF1>E**. The Form Editor adds the new field to the block, and invokes a form where you can edit the properties of the field just created. (See Editing Field Properties.)

SELECTING AND MOVING SCREEN ELEMENTS

Selecting Screen Elements

The items you see on the Form Editor's Main Screen and Block Viewer Screen are called **screen elements**. They include field captions and data fields shown on the Main Screen, and block names shown on the Block Viewer Screen. To select a screen element, press **<SpaceBar> or <RET>** over the element. To deselect an element, press **<SpaceBar> or <RET>** again.

Moving Screen Elements

To move a screen element such as a field or block to a new location, position the cursor over the element, select it with **<SpaceBar> or <RET>**, and then use the following key sequences to move the element:

Moving Screen Elements

To drag an element	Press
Up one line	<ARROW UP>
Down one line	<ARROW DOWN>
Right one column	<ARROW RIGHT>
Left one column	<ARROW LEFT>
Five columns to the right	<Tab> or S
Five columns to the left	Q or A
Top of screen	<PF1><ARROW UP>
Bottom of screen	<PF1><ARROW DOWN>
Right edge of screen	<PF1><ARROW RIGHT>
Left edge of screen	<PF1><ARROW LEFT>

You can drag a field only within the boundaries of the block on which it is defined, and you can drag a block only within the boundaries of the page on which it is defined. Note that no matter where you move a field, it remains associated with the block on which it was originally defined.

If you select the caption of a field, both the caption and data portion of the field, if one exists, are selected and can be dragged as a single unit. If you select the data portion of a field, only the data portion is selected and can be dragged independently of the caption.

If you drag a block name to a new location on the Block Viewer Screen, all fields on that block move to a new location.

The Block Coordinate of a block defines the upper left boundary of the block. The block boundary extends from that coordinate to the lower right edge of the Display/Edit Area.

Similarly, the Page Coordinate of a page defines the upper left boundary of the page. If the page is a regular page, the page boundary extends from that coordinate to the lower right edge of the Display/Edit Area. If the page is a "pop-up" page, the Lower Right Coordinate of the page defines the lower right boundary of the page.

EDITING PROPERTIES

Editing Field Properties

To edit the properties of a field, select the field with **<SpaceBar> or <RET>**, and press **<PF4>**. The Form Editor invokes a ScreenMan form where the properties of the field can be edited.

The specific form that is invoked depends on the type of the selected field. For example, the form for editing data dictionary fields looks like this:

```
Data Dictionary Field Properties

          FIELD ORDER:                                  FIELD:
   OTHER PARAMETERS...           SUPPRESS COLON AFTER CAPTION?
          UNIQUE NAME:

              CAPTION:
              DEFAULT:
   EXECUTABLE CAPTION:
   EXECUTABLE DEFAULT:

       BRANCHING LOGIC:
           PRE ACTION:
          POST ACTION:
POST ACTION ON CHANGE:
```

When you enter a value at the "FIELD:" prompt for data dictionary fields, the Form Editor automatically defines the Caption as the field's label. If the field is a multiple field, the Form Editor adds the word "Select" before the field's label. If the field is a word processing field, the Form Editor adds the characters "(WP)" after the field's label. At the "CAPTION:" prompt, you can accept the Form Editor's default, enter a new caption, or enter one of the following:

Shortcuts at the CAPTION Prompt

To define the caption as	Enter at the CAPTION prompt
Field label	!L
Field title	!T
Unique name of field	!U

Shortcuts at the CAPTION Prompt

To define the caption as	Enter at the CAPTION prompt
Duplicated string	!DUP(string,number of occurrences) For example, !DUP("-",79)

The "OTHER PARAMETERS:" prompt is followed by an ellipsis (**...**) to indicate that this field leads to a new page. To view that page, navigate to the Other Parameters field and press the Enter/Return key. A "pop-up" window appears where you can edit additional properties of the field.

```
Data Dictionary Field Properties

Other Parameters

          REQUIRED: ___            DISPLAY GROUP: ____________________
   DISABLE EDITING: ________       RIGHT JUSTIFY: ___
     SUB PAGE LINK: _____

       DATA LENGTH: ___
CAPTION COORDINATE: _______
   DATA COORDINATE: _______

   DATA VALIDATION: ___________________________________
```

To close the Other Parameters "pop-up" window, press **<PF1>C**. To return to the Form Editor's Main Screen, press **<PF1>E** to exit and save your changes, or press **<PF1>Q** to quit the form without saving your changes.

Editing Field Captions and Data Length

As described above, you can press **<PF4>** to invoke a ScreenMan form to edit the caption and data length of fields. You can also edit these properties directly from the Form Editor's Main Screen.

To change the caption of a field, position the cursor over the caption, and press **<PF3>**. You can then edit the caption with the same editing keys available in ScreenMan's Field Editor. Press the Enter/Return key when you are finished editing the caption.

To change the data length of a field, position the cursor over the data portion of the field, and press **<PF3>**. You can then increase and decrease the data length by pressing **<ARROW RIGHT**> and **<ARROW LEFT>**. An indicator (L=n) at the lower right portion of the Main Screen indicates the current data length. Press the Enter/Return key when you are finished editing the data length.

Reordering All Fields on a Block

After creating and arranging all the fields on a block, you can quickly make the Field Orders of all the fields equivalent to the tab order by doing the following:

1. Go to the Block Viewer Screen (**<PF1>V**)
2. Select the block (**<SpaceBar> or <RET>** over the block name)
3. Press **<PF1>O**

Remember that the Field Order is the order in which the elements on the block are traversed when the user presses the Enter/Return key. The **<PF1>O** key sequence reassigns Field Order numbers to all the elements on the block, so that the Enter/Return key takes the user from field to field in the same order as the Tab key (left to right, top to bottom).

Note that if you refer to fields by Field Order in places such as Branching logic and Pre and Post Actions, reordering the fields on the block could cause that code to refer to the wrong fields. You must then modify the code to either reflect the new Field Order numbers, or refer to those fields by Caption or Unique Name instead.

Editing Block Properties

To edit the properties of a block on the current page, press **<PF1>V** to go to the Block Viewer Screen, select the block name with **<SpaceBar> or <RET>**, and press **<PF4>**. The Form Editor invokes a ScreenMan form where the properties of the block can be edited.

The form for editing block properties looks like this:

```
Block Properties Stored in FORM File

   BLOCK NAME:                                  BLOCK ORDER:
TYPE OF BLOCK:                             OTHER PARAMETERS...
 POINTER LINK:

   PRE ACTION:
  POST ACTION:

Block Properties Stored in BLOCK File

       NAME:                                DESCRIPTION (WP):
  DD NUMBER:                              DISABLE NAVIGATION:

 PRE ACTION:
POST ACTION:
```

The fields on the top portion of the preceding screen are fields from the FORM file. Changes to the values of the fields in this area affect the block only as it is used on this particular form. The fields on the bottom portion of the screen are fields from the BLOCK file. Changes to the values of the fields in this area affect the properties of the block itself, and thus affect any form that uses this block.

When you create a new block, make sure that the DD Number is correct. The Form Editor provides a default DD Number equal to the Primary File of the form. If you are creating a block that contains fields from a subfile, or from a file to which you are navigating, you must change the DD Number.

Editing Page Properties

See "Going to Another Page" for how to move from page to page when editing a form.

To edit the properties of the current page, press <PF4>P from the Form Editor's Main Screen. The form for editing page properties looks like this:

```
                              Page Properties

      PAGE NUMBER:
        PAGE NAME:
     HEADER BLOCK:

  PAGE COORDINATE:                        IS THIS A POP UP PAGE?
                                         LOWER RIGHT COORDINATE:
        NEXT PAGE:
    PREVIOUS PAGE:
     PARENT FIELD:

 DESCRIPTION (WP):
       PRE ACTION:
      POST ACTION:
```

If you want the page to be a "pop-up" page (window), enter YES at the "IS THIS A POP UP PAGE?" prompt, and enter a value for "LOWER RIGHT COORDINATE".

Editing "Pop-Up" Page Coordinates

As described above, you can press **<PF4>P** to invoke a ScreenMan form to edit the properties of the current page. You can also change the coordinate of a "pop-up" page directly from the Form Editor's Main Screen, by selecting and dragging the border of the "pop-up" page.

The following is an example of the Main Screen of the Form Editor where the current page is a "pop-up" page.

```
THIS IS A CAPTION ONLY FIELD

         NAME: __ __
          SSN: ____
Select PHONE: ____      This is the pop-up page

_____________________________________________________________________
File: ZZFILE NAME (#nnnn)                                      R1,C1
Form: ZZFORM NAME
Page: n (ZZName of Pop-Up Page)

<PF1>Q=Quit  <PF1>E=Exit  <PF1>S=Save  <PF1>V=Block Viewer  <PF1>H=Help
```

To move the entire "pop-up" page around on the Main Screen, position the cursor anywhere on the top boundary of the "pop-up" page and press **<SpaceBar> or <RET>** to select it. You can then use the navigational keys described in the section "Moving Screen Elements" to drag the entire "pop-up" page to a new location. Press **<SpaceBar> or <RET>** again to lock the page in its new position.

To resize the "pop-up" page—that is, to change the lower right coordinate of the page—position the cursor over the lower right corner of the page boundary and press **<SpaceBar> or <RET>** to select it. You can then use the navigational keys to move the corner to a new location. Press **<SpaceBar> or <RET>** again when the page is the correct size.

Editing Form Properties

To edit the properties of the form, press **<PF4>M** from the Form Editor's Main Screen. The form for editing form properties looks like this:

```
                              Form Properties

             NAME:
            TITLE:

       PRE ACTION:
      POST ACTION:
 DATA VALIDATION:
        POST SAVE:

      DESCRIPTION:                            RECORD SELECTION PAGE:


      READ ACCESS:
     WRITE ACCESS:
```

CHOOSING ANOTHER FORM

You can select another form to edit or create a new form without leaving the Form Editor. Press **<PF1>M** or **<PF2>M** to select another file and form. You will see the same prompts described in the section "Invoking the Form Editor":

```
EDIT/CREATE FORM FOR WHAT FILE:
```

and:

```
Select FORM:
```

If you select a different form or create a new form, and changes to the previous form (e.g., ZZTEST) haven't yet been saved, the Form Editor asks:

```
Save changes to form ZZTEST? YES//
```

to give you the opportunity to save or discard your changes before moving on to the next form.

DELETING SCREEN ELEMENTS (FIELDS, BLOCKS, PAGES, AND FORMS)

In general, to delete a **screen element**, select and edit the properties of the element, then enter an at-sign (@) at the first field of the ScreenMan form.

To delete a **field**, select the field by pressing **<SpaceBar>** or **<RET>** over the caption of the field, press **<PF4>** to invoke the form to edit the properties of the field, and then enter an at-sign (@) at the "FIELD ORDER:" prompt.

Similarly, to delete a **block**, select the block on the Block Viewer Screen, press **<PF4>** to invoke the form to edit block properties, and enter an at-sign (@) at the "BLOCK NAME:" prompt. Answer YES to the warning that deletions are done immediately. If the block is not used on any other forms, the Form Editor also asks whether you want to delete the block from the BLOCK file. If you choose not to delete the block from the BLOCK file, you can subsequently delete the block only by running the ScreenMan option "PURGE UNUSED BLOCKS."

To delete a **page**, make that page the current page (see "Going to Another Page"), press **<PF4>P** to invoke the form to edit page properties, and enter an at-sign (@) at the "PAGE NUMBER:" prompt.

You cannot delete a **form** from the Form Editor. To delete a form, exit the Form Editor and perform the following steps from the VA FileMan menu:

```
Select OPTION: OTHER OPTIONS
Select OTHER OPTION: SCREENMAN
Select SCREENMAN OPTION: ?
 Answer with SCREENMAN OPTION NUMBER, or NAME
Choose from:
   1               EDIT/CREATE A FORM
   2               RUN A FORM
   3               DELETE A FORM
   4               PURGE UNUSED BLOCKS

Select SCREENMAN OPTION: DELETE A FORM
```

See the "ScreenMan Menu Options" section for more information on this menu option.

5. ScreenMan API

INTRODUCTION

VA FileMan's ScreenMan utility provides a screen-oriented interface for editing and displaying data. The API described in this chapter provides entry points for loading a ScreenMan form and entry points you can use at various places within a ScreenMan form.

See Also

- The "ScreenMan Forms" chapter in this manual.
- The "ScreenMan Form Editor" chapter in this manual.
- The ScreenMan Tutorial.

INVOKE SCREENMAN

^DDS

You can call this entry point directly from an M routine to invoke the specified form.

This routine invokes a ScreenMan form attached to the specified file. ScreenMan automatically uses incremental locks to lock all records accessed during an editing session.

Input Variables

DDSFILE	(Required) The number or global root of the Primary File of the form.
DR	(Required) The name of the form (an entry in the Form file) enclosed in square brackets.
DA	(Optional) The record number of the file entry to display or edit. If DA is null or undefined, the form must either contain no data dictionary fields or have a Record Selection page, which is the first page ScreenMan presents to the user and is where the user can select a record from the file. (See Example 2 below

when a subfile is being accessed directly.)

DDSPAGE (Optional) The Page Number of the first page to display to the user. If '$G(DDSPAGE), a page with a Page Number equal to 1 must exist on the form and that is the first page ScreenMan presents to the user.

DDSPARM (Optional) A string of alphabetic characters that control ScreenMan's behavior are listed below:

C Return the variable DDSCHANG=1 if ScreenMan detects that the user saved a **C**hange to the database.

E Return **E**rror messages in ^TMP("DIERR",$J) and return DIERR if ScreenMan encounters problems when initially trying to load the form. If DDSPARM does not contain an E, ScreenMan prints messages directly on the screen, and returns the variable DIMSG equal to null.

S Return the variable DDSSAVE=1 if the user pressed <PF1>S or <PF1>E, or entered an "Exit" or "**S**ave" command from the Command Line, whether or not any changes were actually made on the form.

If ^DDS is used to display or edit data in a subfile directly, the following variables must be set in addition to the variables listed above:

DDSFILE(1) (Required) Contains the subfile number or the global root of the subfile.

DA(1) ... DA(n) The DA array, where DA is the subrecord number at the deepest level and DA(n) is the record number at the top level.

All the input variables are returned unchanged by the ^DDS call. DDSFILE(1) should be killed when the call is complete to avoid conflict with subsequent ^DDS calls.

Output Variables

DDSCHANG $G(DDSCHANG)=1, if the DDSPARM input variable to ^DDS contains a C and ScreenMan detects that the user saved a change to the database.

DDSSAVE $G(DDSSAVE)=1, if the DDSPARM input variable to ^DDS contains an S and the user pressed <PF1>E or <PF1>S, or issued the "Save" or "Exit" command from the Command Line.

DIMSG $D(DIMSG)>0, if the form could not be loaded, and the DDSPARM input variable to ^DDS does **not** contain an E. (See the description of the DDSPARM input variable above.)

DTOUT $D(DTOUT)>0, if the user times out during the editing session.

The DDS Variable

$D(DDS) can be checked within programming hooks such as Executable Help and Input Transforms to determine whether the hook is being executed from within a ScreenMan form. In that case, $D(DDS) evaluates to true.

Examples

Example 1

Invoke the form EE FORM1 to edit the 15th entry in file #16500, as shown below:

```
>S DDSFILE=16500,DA=15,DR="[EE FORM1]" D ^DDS
```

Example 2

As shown below, invoke the form EE FORM2 to edit the 31st subentry in subfile #16100.01, for the 9th entry in file #16100; Page Number 11 is the first page to present to the user; and have ScreenMan return DDSCHANG if it detects a change to the database when the user exits:

```
>S DDSFILE=16100,DDSFILE(1)=16100.01
>S DA=31,DA(1)=9,DR="[EE FORM2]"
>S DDSPAGE=11,DDSPARM="C"
>D ^DDS
```

Error Codes Returned

NOTE: Error codes are returned only if the DDSPARM input variable to ^DDS contains an E.

201 The specified input variable is missing or invalid.

202 One of the input variables is not properly specified.

405 Entries in the file cannot be edited.

810 At least one of the required ^%ZOSF nodes is missing.

840 The Terminal Type file does not have an entry that matches IOST(0).

842 At least one required piece of data in the Terminal Type file is null for the terminal type identified by IOST(0).

845 A call to HOME^%ZIS returns $G(POP)>0.

3021 The specified form does not exist in the Form file, or DDSFILE is not the Primary File of the form.

3022 The specified form contains no pages.

3023 The form does not contain the specified page.

RETRIEVE/STUFF FIELDS

$$GET^DDSVAL( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This extrinsic function retrieves data from a data dictionary field.

Text for a word processing field is moved into a global array and $$GET^DDSVAL returns the closed root of that array. The array has the same format as a FileMan word processing field.

Computed fields in FileMan files cannot be retrieved. To retrieve the value of a computed field defined on the form, use the $$GET^DDSVALF function described below.

If, while a form is running, a call to $$GET^DDSVAL fails, ScreenMan prints an error message in the Command Area.

Format

```
$$GET^DDSVAL(FILE,[.]RECORD,FIELD,.ERROR,FLAGS)
```

Input Parameters

FILE (Required) The global root or number of the file or subfile. At the field level, the local variable DIE contains the current global root.

[.]RECORD (Required) The internal entry number or an array of internal entry numbers. This parameter has the same form as the DA array. At the field level, the local array DA contains the current array of internal entry numbers.

FIELD (Required) The field name or number or a relational expression that follows a forward pointer (e.g., POINTER:FIELD). The "Forward Pointers" section of the "ScreenMan Forms" chapter in this manual describes in detail the syntax accepted by ScreenMan to describe a relational jump via a DD field.

.ERROR (Optional) $D(ERROR)>1, if the function call fails.

FLAGS (Optional) Controls whether the internal or external form is returned, as shown below (the I and E flags have no effect if FIELD is a word processing field):

I Return the **I**nternal form of the data. (Default)

E Return the **E**xternal form of the data.

Examples

Example 1

Retrieve the internal form of the .01 field of the record currently being edited:

```
S nmspNAME=$$GET^DDSVAL(DIE,.DA,.01)
```

Example 2

Retrieve the external form of field #20, record #362, in file #16000:

```
S nmspDATE=$$GET^DDSVAL("^DIZ(16000,",362,20,"","E")
```

Example 3

Retrieve the text contained in a word processing field named DESCRIPTION:

```
S nmspWP=$$GET^DDSVAL(DIE,.DA,"DESCRIPTION")
```

The text of the DESCRIPTION field is moved to the array as follows:

```
@nmspWP@(0)=Header node of word processing field
@nmspWP@(1,0)=Line 1
@nsmpWP@(2,0)=Line 2
    ...etc.
```

PUT^DDSVAL( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This procedure stuffs data into a data dictionary field as part of ScreenMan's transaction. The data passed to this procedure is filed in the database only when the user explicitly saves changes.

If the specified field is a word processing field, the value passed to the procedure is the closed root of the array that contains the text.

If the specified field is a multiple field, the value passed is the subrecord first displayed to the user as a default at the multiple field. This value is a default for selection and is not actually filed.

Values cannot be stuffed into computed fields.

If, while a form is running, a call to PUT^DDSVAL fails, ScreenMan prints an error message in the Command Area.

Format

```
PUT^DDSVAL(FILE,[.]RECORD,FIELD,VALUE,.ERROR,FLAGS)
```

Input Parameters

FILE (Required) The global root or number of the file or subfile. At the field level, the local variable DIE contains the current global root.

[.]RECORD (Required) The internal entry number or an array of internal entry numbers. This parameter has the same form as the DA array. At the field level, the local array DA contains the current array of record numbers.

FIELD (Required) The field name or number.

VALUE (Required) The value to stuff into the data dictionary field. If FLAGS (described below) does not contain an I, the value must be in the form of a valid, unambiguous user response.

If FIELD is a word processing field, VALUE must be the closed root of the array that contains the text. The subscripts of the nodes below the root must be positive numbers, although they need not be integers, and there can be gaps in the sequence. The text must be in these nodes or in the 0 node descendent from these nodes.

If FIELD is a multiple field, VALUE determines the subrecord to display to the user as a default for selection. It is not a value that is ever filed. VALUE can be "FIRST", "LAST", or the specific internal entry number of the subrecord to display. "FIRST" indicates that the subrecord with the lowest internal entry number should be displayed and "LAST" indicates that the subrecord with the highest internal entry number should be displayed.

.ERROR (Optional) $D(ERROR)>1, if the procedure call fails.

FLAGS Indicates whether VALUE is in internal or external form, as shown below:

A **A**ppend new word processing text to the current text. This flag can be used only when stuffing text into a word processing field. If the A flag is not sent, the current word processing text is completely erased before the new text is added.

I VALUE is in **I**nternal form; it is not validated.

E VALUE is in **E**xternal form. (Default)

The I and E flags have no effect when FIELD is a word processing field.

Examples

Example 1

Stuff the value 2940801 into a date field #20. The value passed is in internal form:

```
D PUT^DDSVAL(DIE,.DA,20,2940801,"","I")
```

No data validation is performed.

Example 2

Stuff word processing text from an array into a word processing field named DESCRIPTION as shown below:

```
D PUT^DDSVAL(DIE,.DA,"DESCRIPTION","^nmspWP(""TEXT"")")
```

The array that contains the text looks like:

```
^nmspWP("TEXT",1,0)=Line 1
^nmspWP("TEXT",2,0)=Line 2
    ...etc.
```

OR:

```
^nmspWP("TEXT",1)=Line 1
^nmspWP("TEXT",2)=Line 2
    ...etc.
```

$$GET^DDSVALF( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This extrinsic function retrieves data from a form-only field or a computed field.

If, while a form is running, a call to GET^DDSVALF fails, ScreenMan prints an error message in the Command Area.

Format

```
$$GET^DDSVALF(FIELD,BLOCK,PAGE,FLAGS,IENS)
```

Input Parameters

FIELD (Required) The Field Order number, Caption, or Unique Name of the form-only field.

BLOCK (Required at the page and form levels) The Block Order or Block Name. The default is the current block.

PAGE (Required at the form level) The Page Number or Page Name. The default is the current page.

FLAGS (Optional) Controls whether the internal or external form is returned, as shown below:

- **I** Return the **I**nternal form of the data. (Default)
- **E** Return the **E**xternal form of the data.

IENS (Required at the page and form levels) The standard IENS that identifies the entry or subentry associated with the form-only field. The default is the current entry or subentry. For a detailed description of IENS, see "IENS—A New Way to Identify Entries and Subentries" in the "Database Server (DBS)" chapter in this manual.

Examples

Example 1

Retrieve the value of a computed field called TOTAL on the current block:

```
S nmspTOT=$$GET^DDSVALF("TOTAL")
```

Example 2

Retrieve the external form of a form-only date field with caption "DATE OF BIRTH" on a block named "ZZBLOCK 1":

```
S nmspDATE=$$GET^DDSVALF("DATE OF BIRTH","ZZBLOCK 1","","E")
```

PUT^DDSVALF( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This procedure stuffs data into a form-only field.

If, while a form is running, a call to PUT^DDSVALF fails, ScreenMan prints an error message in the Command Area.

Format

```
PUT^DDSVALF(FIELD,BLOCK,PAGE,VALUE,FLAGS,IENS)
```

Input Parameters

FIELD (Required) The Field Order number, Caption, or Unique Name of the form-only field.

BLOCK (Required at the page and form levels) The Block Order or Block Name. The default is the current block.

PAGE (Required at the form level) The Page Number or Page Name. The default is the current page.

VALUE (Required) The value to stuff into the form-only field. If FLAGS (described below) does not contain an I, the value must be in the form of a valid, unambiguous user response.

FLAGS (Optional) Indicates whether VALUE is in internal or external form, as shown below:

- **I** VALUE is in **I**nternal form; it is not validated.
- **E** VALUE is in **E**xternal form (default).

IENS (Required at the page and form levels) The standard IENS that identifies the entry or subentry associated with the form-only field. The default is the current entry or subentry. For a detailed description of IENS, see "IENS—A New Way to Identify Entries and Subentries" in the Introduction to the "Database Server (DBS)"

chapter in this manual.

Example

Stuff the value 2940801 into a form-only date field with the caption "DATE", as shown below:

```
    D PUT^DDSVALF("DATE","","","AUG
1,1994")
```

The value passed is in external form (the default).

HELP MESSAGES

HLP^DDSUTL( )

You can use this entry point only within a ScreenMan form, at all places where M code can be placed on the form.

This procedure prints messages in the Command Area.

If you pass the string "$$EOP", then ScreenMan will issue the prompt "Press RETURN to continue" in the Command Area. This is useful if, for example, you want to print messages as part of the post action of a page, and need to pause to give the user a chance to read the messages before ScreenMan leaves that page.

Formats

1. `HLP^DDSUTL(STRING)`
2. `HLP^DDSUTL(.STRING)`

Input Variables

STRING (Required) The message to print in the Command Area.

.STRING (Required) An array of messages to print in the Command Area. STRING(1), STRING(2), ..., STRING(n) each contain a line of text.

MSG^DDSUTL( )

You can call this entry point only within a ScreenMan form and only in the Form level Data Validation.

This procedure prints Data Validation messages on a separate screen. These messages are printed after the user issues the Save command or attempts to save the form on Exit, but before ScreenMan actually updates the database.

Formats

1. `MSG^DDSUTL(STRING)`
2. `MSG^DDSUTL(.STRING)`

Input Variables

STRING (Required) The message to print in the Command Area.

.STRING (Required) An array of messages to print in the Command Area.

STRING(1), STRING(2), ..., STRING(n) each contain a line of text.

REFRESH SCREEN

REFRESH^DDSUTL( )

This entry point repaints all pages on the screen.

You can use this entry point only within a ScreenMan form, and only in:

- Field level Pre Action
- Field level Post Action
- Field level Branching Logic
- Field level Data Validation

Format

```
REFRESH^DDSUTL
```

RUN-TIME FIELD STATUS

REQ^DDSUTL( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This procedure changes the REQUIRED property of a field on the form.

Format

```
REQ^DDSUTL(FIELD,BLOCK,PAGE,VALUE,IENS)
```

Input Variables

FIELD (Required) The Field Order number, Caption, or Unique Name of the field.

BLOCK (Required at the page and form levels) The Block Order or Block Name. The default is the current block.

PAGE (Required at the form level) The Page Number or Page Name. The default is the current page.

VALUE (Required) The value to give the REQUIRED property, listed as follows:

- "" Restore the REQUIRED property to the value defined in the BLOCK file.
- **0** Make the field not required.
- **1** Make the field required.

IENS (Required at the page and form levels) The standard IENS that identifies the entry or subentry associated with the form-only field. The default is the current entry or subentry. For a detailed description of IENS, see "IENS—A New Way to Identify Entries and Subentries" in the Introduction to the "Database Server (DBS)" chapter in this manual.

UNED^DDSUTL( )

You can use this entry point only within a ScreenMan form, in all places where M code can be placed on the form.

This procedure changes the DISABLE EDITING property of a field on the form.

Format

```
UNED^DDSUTL(FIELD,BLOCK,PAGE,VALUE,IENS)
```

Input Parameters

FIELD (Required) The Field Order number, Caption, or Unique Name of the field.

BLOCK (Required at the page and form levels) The Block Order or Block Name. The default is the current block.

PAGE (Required at the form level) The Page Number or Page Name. The default is the current page.

VALUE (Required) The value to give the DISABLE EDITING property, shown below:

- "" Restore the DISABLE EDITING property to the value as defined in the BLOCK file.
- **0** Enable editing, and allow the user to navigate to the field.
- **1** Disable editing, and prevent the user from navigating to the field.
- **2** Disable editing, but allow the user to navigate to the field.

IENS (Required at the page and form levels) The standard IENS that identifies the entry or subentry associated with the form-only field. The default is the current entry or subentry. For a detailed

description of IENS, see "IENS—A New Way to Identify Entries and Subentries" in the Introduction to the "Database Server (DBS)" chapter in this manual.

Part III: Other APIs

6. Browser API

BROWSER (DDBR)

The Browser displays ASCII text on a terminal which supports a scroll region. It enables a user to view text *but not to edit it*. The text can be in the form of a FileMan word processing field or sequential local or global array. The call allows the user to navigate within the document, displaying desired parts of the text. It enables the user to scroll up, down, right, left, to top or bottom, or to go directly to a line, column, or screen location within the document. The user can switch to another document instantaneously, find a string and select the search direction, or split the screen to view separate parts of two documents simultaneously.

FileMan provides a set of entry points so that you can include Browser functionality in your applications.

For more information about the Browser interface, see the "Browser" chapter of the *VA FileMan Getting Started Manual*.

EN^DDBR

This interactive procedure asks the user for file, word processing field, and entry, then displays the text using FileMan's Browser facility. The call allows the user to navigate within the document, displaying parts of the text.

The title bar contains the filename, entry or subentry name, and the fieldname.

The status bar at the bottom displays the leftmost column number, line, and screen number of the cursor location, as well as how to exit and to get help. Users can only access word processing fields in FileMan files to which they have Read access.

Format

```
EN^DDBR
```

Output

After selecting the desired file, field, and record, the word processing text is loaded into the Browser and the Browser screen is displayed on the monitor. The user can then view and navigate through the text.

Details and Features

Switch Function

Switch allows the user to view more than one document. When using the Switch (<PF1>S) function in the Browser to select other FileMan word processing fields, it is important to note that browsing is done directly on the actual record text.

BROWSE^DDBR

This procedure enables the user to utilize FileMan's Browser to view and navigate through a document stored in a sequential local or global array.

Format

```
BROWSE^DDBR(SOURCE_ARRAY,FLAGS,TITLE,LINE,TABS,TOP,BOTTOM)
```

Input Parameters

SOURCE_ARRAY (Required) Source array in a closed root format, passed by value which is the location of a sequential local or global array containing text. This array can optionally include the ",0)" subscript nodes which are contained in FileMan word processing structures.

FLAGS (Optional) Flags to control processing.

N **N**o copy of the document is made. The Browser will use the source document. Useful for long static documents.

CAUTION: When the N flag is used, the Browser does not make a copy of the text; instead it uses the actual record array to browse through. Thus, it is best used when documents stored in word processing fields are static and are not likely to be edited by another user during the browse session. This may be preferable if the source array is a scratch global and is very large. Time and resources are saved by not having to copy such a structure into ^TMP("DDB",$J).

R Restrict switching.

See the Switch function in the Details and Features section.

TITLE (Optional) Text centered in screen title.

LINE (Optional) The line in the document that would be at the bottom margin of the opening screen.

TABS (Optional) Closed array root, passed by value; used to scroll horizontally. If not set, the Browser provides default tab stops. Also see "Setting Tab Stops" under the Details and Features section.

TOP (Optional) A number representing the location of the title bar of the Browser screen.

BOTTOM (Optional) A number representing the location of the status bar of the Browser screen.

Output

A successful call enables the user to utilize the Browser to view and navigate throughout a document stored in a sequential local or global array.

Example

```
>K ^TMP("EXAMPLE",$J)
>N I F I=1:1:300 S ^TMP("EXAMPLE",$J,I)="THIS IS LINE "_I

>D BROWSE^DDBR("^TMP(""EXAMPLE"",$J)","N","Example")
```

The Browser screen displays as follows:

```
-------------------------------------------------------------------------------
                                      Example
-------------------------------------------------------------------------------
THIS IS LINE 1
THIS IS LINE 2
THIS IS LINE 3
THIS IS LINE 4
THIS IS LINE 5
THIS IS LINE 6
THIS IS LINE 7
THIS IS LINE 8
THIS IS LINE 9
THIS IS LINE 10
THIS IS LINE 11
THIS IS LINE 12
THIS IS LINE 13
THIS IS LINE 14
THIS IS LINE 15
```

```
THIS IS LINE 16
THIS IS LINE 17
THIS IS LINE 18
THIS IS LINE 19
THIS IS LINE 20
THIS IS LINE 21
THIS IS LINE 22
-------------------------------------------------------------------------------
Col>   1 |<PF1>H=help <PF1>E=Exit| Line>     22 of 300     Screen>    1 of 14
-------------------------------------------------------------------------------
```

Error Codes Returned

200 Invalid field.

202 Invalid parameter.

309 Multiple field. Invalid file and IENS.

401 Data Dictionary reference for file and field not valid.

501 Extended reference invalid.

510 Invalid type in data dictionary.

601 Record entry does not exist.

602 Record unavailable.

842 Device/Terminal type setup issues.

NOTE: For additional information about Browser error messages, see the sections "How Information is Returned" and "Contents of Arrays" in the "Database Server (DBS)" chapter in this manual.

Details and Features

Switch Function

Switch allows the user to view more than one document. When using the Switch (<PF1>S) function in the Browser to select other FileMan word processing fields, it is important to note that browsing is done directly on the actual record text. Users can only access word processing fields in FileMan files for which they have Read access.

Setting Tab Stops

This will set up the TAB with stops at every tenth column.

```
F I=10:10:100 S TAB(I)=""
TAB(10)=""
TAB(20)=""
TAB(30)=""
   .
   .
   .
TAB(90)=""
TAB(100)=""
```

NOTE: Browser always begins at column 1.

WP^DDBR

This procedure displays word processing fields, as well as allowing navigation throughout the text, in a FileMan-compatible database using VA FileMan's Browser facility.

Format

```
WP^DDBR(FILE,IENS,FIELD,FLAGS,TITLE,LINE,TABS,TOP,BOTTOM)
```

Input Parameters

FILE (Required) File or subfile number.

IENS (Required) Standard IENS indicating internal entry number string.

FIELD (Required) Word processing field name or number.

FLAGS (Optional) Flags to control processing.

N **N**o copy of the document is made. The Browser will use the source document. Useful for long static documents.

CAUTION: When the N flag is used, the Browser does not make a copy of the text; instead it uses the actual record array to browse through. Thus it is best used when documents stored in word processing fields are static and are not likely to be edited by another user during the browse session. This may be preferable if the source text is very large. Time and resources are saved by not having to copy such a structure into ^TMP("DDB",$J).

R **R**estrict switching. See the Switch function in the Details and Features section.

TITLE (Optional) Text that is centered in header. Document title.

LINE (Optional) The line in the document that would be at the bottom margin of the opening screen.

TABS (Optional) Closed array root, passed by value, that is used to scroll horizontally. If not set, the Browser provides default tab stops. Also see "Setting Tab Stops" under this section's "Details and Features."

TOP (Optional) A number representing the location of the title bar of the Browser screen.

BOTTOM (Optional) A number representing the location of the status bar of the Browser screen.

NOTE: The TOP and BOTTOM parameters define the boundaries of the scroll region.

Output

A successful call results in the Browser screens being displayed and enables the user to utilize the Browser to view and navigate through word processing fields in a FileMan-compatible database.

Example

```
>D WP^DDBR(999088,"12,","TEXT","N","Programming SAC")
```

Error Codes Returned

200 Invalid field.

202 Invalid parameter.

309 Multiple field. Invalid file and IENS.

401 Data Dictionary reference for file and field not valid.

501 Extended reference invalid.

510 Invalid type in data dictionary.

601 Record entry does not exist.

602 Record unavailable.

842 Device/Terminal type set up issues.

Details and Features

Switch Function

Switch allows the user to view more than one document. When using the Switch (<PF1>S) function in the Browser to select other FileMan word processing fields, it is important to note that browsing is done directly on the actual record text. Users can only access word processing fields in FileMan files for which they have Read access.

Setting Tab Stops

This will set up the TAB with stops at every tenth column.

```
F I=10:10:100 S TAB(I)=""

TAB(10)=""
TAB(20)=""
TAB(30)=""
    .
    .
    .
TAB(90)=""
TAB(100)=""
```

Margin Note

Browser always begins at column 1.

NOTE: For additional information about Browser error messages, see the sections "How Information is Returned" and "Contents of Arrays" in the "Database Server (DBS)" chapter in this manual.

DOCLIST^DDBR

This procedure call allows passing more than one document to the Browser facility. It enables the user to use the Browser to navigate through multiple documents stored in sequential local or global arrays.

A list of documents is passed by value as an array root. The array is subscripted by the document title and must be set equal to the document's location, in a closed root format. The Browser automatically builds the "Current List" and displays the first document to the screen. When you select the "S"witch function to switch to another document, the rest of the documents are presented as a "Current List." A flag is also available to "R"estrict selection to the "Current List" and prevent selecting FileMan word processing fields in other files.

Format

```
DOCLIST^DDBR(SOURCE_ARRAY,FLAGS,TOP,BOTTOM)
```

Input Parameters

SOURCE_ARRAY (Required) Source array in a closed root format, passed by value which is subscripted by document titles and set to the source array of the document in a closed root format.

FLAGS (Optional) Flag(s) to control processing:

R **R**estrict Switching to other documents not in current list. Otherwise, Switch (<PF1>S) function is active and users can look at other FileMan word processing field entries.

See the Switch function in the Details and Features section.

TOP (Optional) A number representing the location of the title bar of the Browser screen.

BOTTOM (Optional) A number representing the location of the status bar of the Browser screen.

Output

A successful call enables the user to employ VA FileMan's Browser to view and navigate through multiple documents stored in a sequential local or global array.

Example

In this example there are three documents.

Document 1, in ^TMP($J,"DOC",1), looks like:

```
^TMP($J,"DOC",1,1)=Line 1 Document 1
^TMP($J,"DOC",1,2)=Line 2 Document 1
  .
  .
  .
^TMP($J,"DOC",2,1)=Line 1 Document 2
^TMP($J,"DOC",2,2)=Line 2 Document 2
  .
      .
      .
^TMP($J,"DOC",3,1)=Line 1 Document 3
^TMP($J,"DOC",3,2)=Line 2 Document 3
```

and so on...

Building the document list array looks like:

```
>S ^TMP($J,"LIST","DOCUMENT 1")="^TMP($J,""DOC"",1)"
>S ^TMP($J,"LIST","DOCUMENT 2")="^TMP($J,""DOC"",2)"
>S ^TMP($J,"LIST","DOCUMENT 3")="^TMP($J,""DOC"",3)"
```

Making a procedure call with Switching restricted to only this list looks like:

```
>D DOCLIST^DDBR("^TMP($J,""LIST"")","R")
```

Error Codes Returned

200 Invalid field.

202 Invalid parameter.

309 Multiple field. Invalid file and IENS.

401 Data Dictionary reference for file and field not valid.

501 Extended reference invalid.

510 Invalid type in data dictionary.

601 Record entry does not exist.

602 Record unavailable.

842 Device/Terminal type set up issues.

NOTE: For additional information about Browser error messages, see the sections "How Information is Returned" and "Contents of Arrays" in the "Database Server (DBS)" chapter in this manual.

Details and Features

Switch Function Switch allows the user to view more than one document. When using the Switch (<PF1>S) function in the Browser to select other FileMan word processing fields, it is important to note that browsing is done directly on the actual record text. Users can only access word processing fields in FileMan files to which they have Read access.

$$TEST^DDBRT

This function call returns a 1 or 0 (true or false) to determine if the CRT being used can support a scroll region and reverse index. (The device must have scroll region and reverse index capabilities in order to use the Browser.) If 1 is returned, the CRT supports the needed functionality to use the Browser.

Format

```
$$TEST^DDBRT
```

Input Parameters

None

Output

Returns a 1 if true or 0 if false.

Example

```
>W $$TEST^DDBRT
1
```

Error Codes Returned

None

CLOSE^DDBRZIS

This procedure executes $$REWIND^%ZISC(), to rewind the file and copies the text from the host file into a scratch global. It is used when setting up the Browser as a device on a system running Kernel version 8.0 or greater. The call is set up in the CLOSE EXECUTE field of the TERMINAL TYPE file (#3.2). Also, refer to the Kernel System Manual for operating system information, if any, to be included with this call.

Format

```
CLOSE^DDBRZIS
```

Input Parameters

None

Output

None

Error Codes Returned

A message is displayed if the rewinding of the file fails.

NOTE: Kernel Version 8.0 or greater is required. See also OPEN^DDBRZIS and POST^DDBRZIS.

OPEN^DDBRZIS

This procedure captures the text used in the Browser's title. It is used when setting up the Browser as a device on a system running Kernel version 8.0 or greater. The call is set up in the OPEN EXECUTE field of the TERMONAL TYPE file (#3.2). Also, refer to the Kernel System Manual for operating system information, if any, to be included with this call.

Format

```
OPEN^DDBRZIS
```

Input Parameters

None

Output

None

Error Codes Returned

None

NOTE: Kernel Version 8.0 or greater is required. See also CLOSE^DDBRZIS and POST^DDBRZIS.

POST^DDBRZIS

This procedure initializes the Browser to display the text sent to the device. It is used when setting up the Browser as a device on a system running Kernel version 8.0 or greater. The call is set up in the POST-CLOSE EXECUTE field of the DEVICE file (#3.5). Also, refer to the Kernel Systems Manual for operating system information, if any, to be included with this call.

Format

```
POST^DDBRZIS
```

Input Parameters

None

Output

None

Error Codes Returned

None

NOTE: Kernel Version 8.0 or greater is required. See also OPEN^DDBRZIS and CLOSE^DDBRZIS.

7. Import and Export Tools

INTRODUCTION

If you want to use non-M applications (for example, a PC-based application like Microsoft Excel) to manipulate data stored in a VA FileMan file, then you need a way to exchange FileMan data with your application. VA FileMan V. 22.0 provides the interactive Import and Export Tools for these purposes. These tools are made available to users through interactive options.

FileMan V. 22.0 provides entry points for both the Import Tool, FILE^DDMP, and the Export Tool, EXPORT^DDXP, so that you can incorporate their functionality into your applications.

For more information about the Import and Export Tools, see the "Import and Export Tools" chapter of the *VA FileMan Advanced User Manual*.

FILE^DDMP: Data Import

This procedure imports data from ASCII host files into VA FileMan file entries. Each record (line of data) in the host file is stored as a new entry in a specified FileMan file.

For additional information about the Import Tool, see the "Import and Export Tools" chapter of the *VA FileMan Advanced User Manual*.

Format

```
FILE^DDMP([FILE],[[.]FIELDS],[.CONTROL],.SOURCE,[.]FORMAT)
```

Input

FILE (Optional) File number into which imported data will be filed. Do not pass this parameter if the import file specifies the destination VA FileMan file and fields. The file must already exist.

[.]FIELDS (Optional) Array specifying the fields into which imported data will be filed. The array can either:

Name an IMPORT template, or

Directly specify the fields for import.

Do not pass this parameter if the import file specifies the destination FileMan file and fields.

If you have the import fields stored in an IMPORT template, simply set the top-level, unsubscripted node to the name of the template, surrounded by [brackets].

If you are directly specifying fields in this array, set the top-level, unsubscripted node in the FIELDS array to the list of destination field numbers at the top level of the file. Separate each field number with a semicolon. The list of field numbers should match the order of the corresponding data pieces in each import file record.

For any field number that identifies a **multiple**, include the top-level field number of the multiple in the top-level node of the

FIELDS array. Then, set an additional node in the FIELDS array for the multiple, subscripted by the data dictionary subfile number of the multiple. Set this additional node to the list of subfield numbers in the multiple into which to have data filed, separated by semicolons. The order of subfield numbers in this node should match the order of the corresponding data pieces for the multiple in the import file record.

For subfiles within subfiles, repeat this process of identifying the top-level field number of the multiple in the appropriate FIELDS node (one data dictionary level above that of the multiple). Then add an additional node subscripted by data dictionary number identifying the fields in the subfile into which data is to be filed.

You can add more than one subentry for the same subfile (see Importing into Subfiles in Details and Features below).

If the import is based on fixed length (rather than character-delimited) data, follow each field's number with the length of the data for that field enclosed in square brackets. For example, ".01[30];.02[30];.03[10]".

.CONTROL (Optional) Pass this array by reference. You can control the behavior of FILE^DDMP by setting the following nodes in the CONTROL array:

CONTROL("FLAGS") (Optional) Concatenated string of character flags to control processing of the import.

E **E**xternal values are contained in the import file. Convert the values to FileMan internal format and verify during import. If the E flag is not present, data is assumed to be in internal format and is not verified.

F Import **F**ile contains identity of destination FileMan file and fields. If F flag is not present, the FILE and FIELDS parameters are required and must contain file and field information.

CONTROL("MSGS") (Optional) Set to the root of an array (local or global) into which error messages should be returned. If a value is not passed, messages are returned in nodes descendent from ^TMP("DIERR",$J).

CONTROL("MAXERR") (Optional) Set to the number of errors which may be encountered before aborting the import. Default is not to abort.

CONTROL("IOP") (Optional) Set to the name of the device (as stored in the DEVICE file) on which to print the Import Report. This pre-selects the output device. You can also set CONTROL("IOP") to match any of the additional formats for the IOP input variable recognized by ^%ZIS entry point (see the Kernel Systems Manual for more information on ^%ZIS and IOP).

Default is to ask the user for output device.

CONTROL("QTIME")	(Optional) Set to the time for queuing the data filing and subsequent printing of the Import Results report. This pre-selects the time for queuing. The time can be in any format that ^%DT recognizes. Default is to ask the user whether or not to queue and for the queuing time.

.SOURCE (Required) An array that identifies the import file. Pass this array by reference.

SOURCE("FILE")	(Required) Set this node to the import file name.
SOURCE("PATH")	(Optional) Path or directory where the file can be found. If this node is not defined, the default path is used to locate the file.

[.]FORMAT (Required) Specifies the format of the incoming data. You can either:

Pass the name of a FOREIGN FORMAT File entry in the top-level, unsubscripted node of this array, or

Set individual nodes in this array to define the import format (pass by reference).

If you set individual nodes in the array to define the format, you can set:

FORMAT("FDELIM")	Set this node to the field delimiter used for the imported data, if a field delimiter is used.
FORMAT("FIXED")	Set this node to "YES" if the incoming data is in fixed-length format. If not set to "YES", the default format is field-delimited.

FORMAT("QUOTED") Set this node to "YES", if you would like FileMan to ignore the field delimiter in any quoted strings in the incoming data.

Output

Error messages and information supplied via EN^DDIOL are returned in ^TMP or in the array specified by MSG_ROOT. DIERR is defined if there was an error. Error messages are **not** returned for individual records whose import fails, however.

Example

In the example below, the import file is PEOPLE2.CSV. The import file is in Excel (Comma) format, which means the data is comma-delimited. There is a corresponding entry in the FOREIGN FORMAT file called "Excel (Comma)" describing the Excel (Comma) format.

The following code calls FILE^DDMP to import data from PEOPLE2.CSV:

```
S FILE=16100
S CONTROL("MSGS")="MYMSGS"
S CONTROL("FLAGS")="E"
S FIELDS=".01;14;14"
S FIELDS(16100.014)=".01;1"
S SOURCE("FILE")="PEOPLE2.CSV"
S SOURCE("PATH")="VA6$:[SMITH]"
D FILE^DDMP(FILE,.FIELDS,.CONTROL,.SOURCE,"EXCEL(COMMA)")
```

The import data is in external format, so the call to FILE^DDMP uses the E flag. The data in the import file contains records of five comma-delimited values that are to be imported into file #16100, as specified in the FIELDS parameter:

- Data piece 1: File as the .01 field of file #16100.
- Data pieces 2 & 3: File as first entry in subfile #16100.014 (field #s .01 and 1).
- Data pieces 4 & 5: File as second entry in subfile #16100.014 (field #s .01 and 1).

If the data for this import were in fixed length format, the code to set the FIELDS array might look like this:

```
S FIELDS=".01[30];14;14"
S FIELDS(16100.014)=".01[30];1[25]"
```

Note that the field numbers that specify a multiple at the top level have no length associated with them.

Error Codes Returned

202 Incorrect parameter was passed.

312 Required identifier is missing.

409 File does not exist.

501 Field does not exist.

520 A word-processing field was specified.

525 Multiple specified, but no fields in subfile chosen.

1810 Data could not be moved into M environment.

1812 No data found in host file.

1820 Format could not be found in the Foreign Format file.

1821 Inconsistencies in the format chosen.

1822 Incorrect data length for a fixed length format.

1833 Inconsistency involving the "F" flag.

1850 Error in device selection or queuing setup.

1870 The IMPORT template does not exist for the file.

Details and Features

Data Formats

Data fields in the import file can be either character-delimited or fixed length. The method used should match the method described in the FOREIGN FORMAT file entry whose name is passed to FILE^DDMP (alternatively, you can specify these values directly in the FORMAT parameter and not reference a FOREIGN FORMAT File entry.) The only fields from the FOREIGN FORMAT file entry used during import are:

- FIELD DELIMITER
- RECORD LENGTH FIXED?
- QUOTE NON-NUMERIC FIELDS

Required fields

All required VA FileMan identifier fields for the destination file must have data filed into them from the import record:

- If a field defined as a required identifier is not a destination field, the import is not performed.
- If a record being filed has a null value for a required identifier, that record will not be filed.

Identifying Destination File and Fields in Import File

You can store the destination VA FileMan file and fields in the import file, rather than passing them to FILE^DDMP in the FILE and FIELDS parameters. Use the "F" flag to indicate that file and field information is being sent in the import file.

To specify the file and fields in the import file, the first line of data in the import file must be:

```
FILE=filename
```

Don't leave any spaces between the literal tag "FILE=" and the name of the file involved. You can identify the file by file number rather than name, also.

The second line in the import file must contain a list of destination field names, in the order of the data pieces in each import file record. You can use field numbers rather than field names to identify the fields (for instance, you

might want to specify a field by number if its name contains punctuation characters).

If the import is delimited, the names should be separated by whatever the specified delimiter is:

```
NAME,ADDRESS
```

If the import is fixed length, the field names should be followed by the field length in [brackets], and then separated by a comma:

```
NAME[25],ADDRESS 1[20]
```

To specify a field in a subfile, show the complete path to the field using the format:

```
multiple fieldname:fieldname
```

Specify as many multiple field names as necessary (separated by colons) to indicate a complete path to the field being imported.

The third and subsequent lines of the import file should contain the data records to be filed.

Here is a listing of an example import file containing destination field information:

```
FILE=DA RETURN CODES
DA RETURN STRING,TERMINAL TYPE STRING
[=7c,C-QVT103
[?1;0c,C-WYSE 75
[?1;2c,C-VT100
[?1;6c,C-VT100
```

Importing into Subfiles

Each record (line of data) from an import file is always stored as a new record at the top level of the destination VA FileMan file. However, you can populate more than one entry in a subfile descendent from the new entry, from a single import record.

To file more than one entry in a subfile, repeat the subfile's multiple field number in the field string of the higher level file or subfile. Each import record must add the **same set of fields** to the subfile in question,

however, as specified by the set of fields you list in the subfile's FIELDS(subfile#) node.

Also, new subentries need to be added to every subfile on a path to the lowest level subfile. Because of this, you must include fields for the .01 field and all the required identifiers for every subfile as well as at the top level of the file.

EXPORT^DDXP: Data Export

This procedure exports data from VA FileMan files into ASCII host files. Each entry in a specified FileMan file is stored as a line of data in the host file.

For additional information about the Export Tool, see the "Import and Export Tools" chapter of the *VA FileMan Advanced User Manual*.

Format

```
D EXPORT^DDXP(FILE,EXPORT_TEMPLATE,DELETE_FLAG,SORT_TEMPLATE,[.]FR,
[.]TO,.DIS,[.]DISTOP,IOP,DQTIME)
```

Input Parameters

FILE

(Required) File number from the file where the data to be exported is located.

NOTE: A special case occurs when exporting data from file number 1.1, the AUDIT file. In this case, FILE then becomes "1.1^<file number of the audited file>". For example, if the audited data is associated with the PATIENT file, then the string would look like: "1.1^2".

EXPORT_TEMPLATE

(Required) The name of the export template, **without** the surrounding brackets "[]", that was created when the developer used the option: CREATE EXPORT TEMPLATE.

DELETE_FLAG

(Optional) Indicates whether or not the export template should be deleted when exporting of the data is finished.

It has two possible values:

0 (zero) Do NOT delete the export template when the export has finished. Default.

1 DELETE the export template when the export has finished.

SORT_TEMPLATE (Optional) The name of the sort template, **without** the surrounding brackets "[]", that will be used for file sorting. If this parameter is Null, then the user will see the standard FileMan Sort dialog.

[.]FR (Optional) The START WITH: values of the SORT BY fields. If FR is undefined, the user will be asked the START WITH: question for each SORT BY field. If FR is defined, it consists of one or more comma pieces, where the piece position corresponds to the order of the sort field in the BY variable.

Passed by reference.

The details of this parameter are identical to those of the FR input variable of the Classic FileMan print routine EN1^DIP. For additional information, see that description.

[.]TO (Optional) The GO TO: values of the SORT BY fields. Its characteristics correspond to the FR variable. If undefined, the user will be asked the GO TO: questions for each SORT BY field. If TO is defined, it consists of one or more comma pieces.

Passed by reference.

The details of this parameter are identical to those of the TO input variable of the Classic FileMan print routine EN1^DIP. For additional information, see that description.

.DIS (Optional) You can screen out certain entries so that they do not appear on the output by setting the optional array DIS. The first subscript in this array can be 0 (zero). This variable (as well as all the others) contains an executable line of M code which includes an IF-statement.

Passed by reference.

The details of this parameter are identical to those of the DIS (0) and DIS(n) input variables of the Classic FileMan print routine EN1^DIP. For additional information, see that description.

[.]DISTOP

(Optional) If Kernel is present, by default prints queued through the EN1^DIP call can be stopped by the user with a TaskMan option. However, if this variable is set to 0, users will not be able to stop their queued prints.

Passed by reference.

The details of this parameter are identical to those of the DISTOP input variable of the Classic FileMan print routine EN1^DIP. For additional information, see those descriptions.

IOP

(Optional) EXPORT^DDXP calls the ^%ZIS entry point to determine which device output should go to. This requires user interaction unless you pre-answer the DEVICE prompt. You can do this by setting IOP equal to the name of the device (as it is stored in the DEVICE file) to which the output should be directed.

Passed by reference.

The details of this parameter are identical to those of the IOP input variable of the Classic FileMan print routine EN1^DIP. For additional information, see that description.

DQTIME

(Optional) If output is queued, this variable contains the time for printing. You can set it equal to any value that %DT recognizes.

Passed by reference.

The details of this parameter are identical to those of the DQTIME input variable of the Classic

FileMan print routine EN1^DIP. For additional information, see that description.

Output Parameters

None

Examples

See examples below for ways to use EXPORT^DDMP.

Note that in all examples, the DELETE_FLAG is null, i.e., 0 (zero).

Example 1

In this example, no sort template is provided and the user is asked sort dialog:

```
D EXPORT^DDXP(2,"ZZS0 SKIP TEST")
SORT BY:  NAME//
START WITH NAME:  FIRST//
DEVICE:
```

Example 2

In this example, a sort template is provided:

```
D EXPORT^DDXP(2,"ZZS0 SKIP TEST",,"ZZS0 TEXPORT #1")

*Previous selection: DATE ENTERED INTO FILE from Jan 1,1997 to Jun 4,1999
START WITH DATE ENTERED INTO FILE: FIRST// 1/1/97  (JAN 01, 1997)
GO TO DATE ENTERED INTO FILE: LAST// T  (JUN 07, 1999)
DEVICE:
```

Example 3

In this example, a sort template is provided and the FROM and TO values are supplied:

```
S FR="1/1/97"
S TO=DT
D EXPORT^DDXP(2,"ZZS0 SKIP TEST",,"ZZS0 TEXPORT #1",FR,TO) DEVICE:
```

Example 4

This example shows the special case of the AUDIT file.

Because users may want to export information from the AUDIT file (1.1), a special case has been created. All parameters that are to be passed remain the same as above, EXCEPT for the FILE parameter. In this special case, the format is as follows:

```
FILE "1.1^<file number of audited file>"
```

Here is an example:

```
D EXPORT^DDXP("1.1^16200","ZZSO",,"ZZS0 AUDIT")

Previous selection: DATE/TIME RECORDED from Jan 1,1997 to Dec 31,1997@24:00
START WITH DATE/TIME RECORDED: FIRST// 1/1/97   (JAN 01, 1997)
GO TO DATE/TIME RECORDED: LAST// 12/31/97  (DEC 31, 1997@24:00)
DEVICE:
```

Example 5

This example shows a sample sort template, export template, and routine.

In this example, we want use Microsoft Word Mail Merge to send a brochure to the new patients who visited the Medical Center in the previous month. For purposes of illustration we are going to assume the month in question was March of 2000.

Sort Template Used:

```
NAME: ZZSO NEW PATIENTS//
READ ACCESS: @//
WRITE ACCESS: @//
SORT BY: ]NAME//
 * Previous selection: NAME not null
 START WITH NAME: FIRST//
  WITHIN NAME, SORT BY: DATE ENTERED INTO FILE  Replace
 * Previous selection: DATE ENTERED INTO FILE from Feb 1, 2000 to Feb 29,
2000
  START WITH DATE ENTERED INTO FILE: FIRST// 3/1/00  (MAR 01, 2000)
  GO TO DATE ENTERED INTO FILE: LAST// 3/31/00  (MAR 31, 2000)
    WITHIN DATE ENTERED INTO FILE, SORT BY:
 STORE IN 'SORT' TEMPLATE:    ZZSO NEW PATIENTS
          (Jun 17, 1999@05:14) User #9152 File #2    SORT
```

```
DATA ALREADY STORED THERE....OK TO PURGE? NO// YES
 DESCRIPTION:
  1>Get previous month's New Patients for mass marketing mailing.
 EDIT Option:

 SHOULD TEMPLATE USER BE ASKED 'FROM'-'TO' RANGE FOR 'DATE ENTERED INTO
 FILE'? NO// YES
```

Export Template Used:

```
NAME: ZZSO PATIENT ADDRESS X
 DATE CREATED: JUN 17, 1999@08:26
 READ ACCESS: @                        FILE: PATIENT
 USER #: 9152                          WRITE ACCESS: @
 DATE LAST USED: MAR 01, 2000          TEMPLATE TYPE: EXPORT
FIELD ORDER: 1                          DATA TYPE: FREE TEXT
FIELD ORDER: 2                          DATA TYPE: FREE TEXT
FIELD ORDER: 3                          DATA TYPE: FREE TEXT
FIELD ORDER: 4                          DATA TYPE: FREE TEXT
FIELD ORDER: 5                          DATA TYPE: FREE TEXT
 EXPORT FORMAT: EXCEL (COMMA)          SUB-HEADER SUPPRESSED: YES
HEADER (c): @@
FIRST PRINT FIELD: W $C(34)//
THEN PRINT FIELD: NAME;X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: W $C(44);X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: STREET ADDRESS [LINE 1];X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: W $C(44);X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: CITY;X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: W $C(44);X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: STATE;X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: W $C(44);X//
HEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: ZIP CODE;X//
THEN PRINT FIELD: W $C(34);X//
THEN PRINT FIELD: W $C(44);X//
COMPILED (c): NO
```

Example Routine and Output:

```
ZZSONPAD --
        ;SFISC/SO-Sample Export API Usage ;7:18 AM  1 APR 2000
       ;;1.0
       N %DT
       S %DT="AEPX"
       S %DT("A")="Enter Beginning of previous Month: "
       D ^%DT
       I Y<1 Q
       S FR=","_$P(Y,".")
       S %DT="AEPX"
```

```
S %DT("A")="Enter End of previous Month: "
D ^%DT
I Y<1 Q
S TO=","_$P(Y,".")
K %DT
D EXPORT^DDXP(2,"ZZSO PATIENT ADDRESS X",,"ZZSO NEW PATIENTS",FR,T
 O)
Q FM22 >D ^ZZSONPAD
```

```
Enter Beginning of previous Month: 3/1/00  (MAR 01, 2000) Enter End of
previous Month: 3/31/00  (MAR 31, 2000) DEVICE:   Telnet terminal
"BIRD,TWEETY","123 TREE ST.","SAN FRANCISCO","CALIFORNIA","94521",
"BUNNY,BUGS","123 CARROT ST","SAN FRANCISCO","CALIFORNIA","90041",
"CAT,SYLVESTER","132 ANY ST","SAN FRANCISCO","CALIFORNIA","98765",
"DUCK,DAFFY","301 Howard St.","San Francisco","CALIFORNIA","94105",
"LASVEGAS,LEACH","111 LAS VEGAS BLVD.","LAS VEGAS","NEVADA","89101",
"RUNNER,ROAD","234 ROAD ST.","SAN FRANCISCO","CALIFORNIA","94077",
"SAM,YOSEMITE","234 YOSEMITE","SAN DIEGO","CALIFORNIA","98765",
```

8. Extract Tool

INTRODUCTION

The Extract Tool lets you move or copy records from one VA FileMan file to another; a typical use is to archive records. Two entry points are provided, with EXTRACT^DIAXU being the preferred entry point to use for extracting data records.

The extract tool can also be used interactively from a set of options; this is described in the "Extract Tool" chapter of the *VA FileMan Advanced User Manual*.

EN^DIAXU: Extract Data

The extract tool has been enhanced and this has resulted in a number of changes to the output variables of the EN^DIAXU entry point.

This entry point extracts data specified in the EXTRACT template for a single entry and moves that data to a destination file. The source entry may be deleted after the extract process is completed.

If you need to extract in batches (more than one entry), you should use the EXTRACT^DIAXU entry point instead.

Input Variables

DIAXF	(Required) The number of the file that contains the source entry.
DIAXT	(Required) The EXTRACT template name enclosed in brackets in the source file that contains specifications of data to be extracted.
DIAXFE	(Required) Internal entry number of the source entry from which data will be extracted.
DIAXDEL	(Optional) This variable, if defined, tells the program to delete the source entry. If not defined, the source entry is unchanged.

Output Variables (Successful Extracts)

If the extract process was completed and the data was **successfully** moved to the destination file, the variables are returned as follows:

DIAXDA Internal entry number of entry created in the destination file.

In addition to DIAXDA,

```
^TMP("DIAXU",$J,"RESULT",DIAXF,DIAXFE)=DIAXDA
```

is returned.

DIAXNTC No longer returned. For batch processing of extracts, you should use the EXTRACT^DIAXU entry point instead of this one.

DIAXFE No longer killed upon exit.

DIAXF Not killed.

DIAXT Not killed.

DIAXDEL Not killed.

Output Variables (If an Error Occurs)

If an **error** occurs during the extract process, the following variable and global array are returned instead:

DIERR Contains the following two ^-pieces of information:

1. Number of errors generated during the call, and
2. Total number of lines of the error messages

In addition, the following "RESULT","ERR" node is returned:

```
^TMP("DIAXU",$J,"RESULT","ERR",file#,ien)
```

For example,

```
^TMP("DIAXU",$J,"RESULT","ERR",662001,5)
```

	No longer indicates the total number of errors encountered during the extract process.
(array in ^TMP)	Error information is returned in ^TMP("DIERR",$J), in the same format that error information is returned for DBS calls. Please refer to the How to Use the Database Server section for a complete description of this array.
DIAXDA	Not defined.
All input variables	Left defined.

Error Codes Returned

This entry point calls $$FIND1^DIC, LIST^DIC, UPDATE^DIE, $$GET1^DIQ, and GETS^DIQ; any errors returned by these entry points can also be returned by EN^DIAXU.

In addition, the following errors may be returned:

201 An input variable is missing or invalid.

601 The entry does not exist.

EXTRACT^DIAXU: Extract Data

EXTRACT^DIAXU is a new entry point. It is the preferred entry point for extracting data records. The principal features introduced with this entry point for extracting data are:

- More than one record can be extracted in a call.
- Subrecords can be extracted as individual transactions. Previously, an entire record including all subrecords had to be extracted as a single entity.
- DBS-style error reporting is used.

Like EN^DIAXU, this entry point extracts data from the fields specified in the EXTRACT template and places that data in an entry in a destination file. You can optionally delete the source entry after the extract process is completed.

Format

```
D EXTRACT^DIAXU(FILE,SOURCE,EXTRACT_TEMPLATE,FLAGS,.SCREENS,
  .FILING_LEVEL,TARGET_ROOT,MSG_ROOT)
```

Input

FILE (Required) File number of source file.

SOURCE (Required) Can be 1 of 2 formats:

- IEN: The record number of a single record, at the top level of the file, to extract.
- SEARCH template name: The name of a SEARCH template, in brackets, that contains SEARCH results (a list of record numbers). For example,

  ```
  S SOURCE="[TEMPLATE_NAME]"
  ```

EXTRACT_TEMPLATE (Required) The name of the EXTRACT template, in brackets, containing what fields to move.

FLAGS

(Optional) A string of characters to modify the behavior of the Extract tool. Permissible characters in the string are:

D Tells the extract tool to **D**elete source records if they were moved successfully. Note that deletion is only done for entire (top-level) records. Subrecords are not individually deleted, even if they are individually extracted.

NOTE: If the SOURCE parameter is set to a SEARCH template, and you include a D in the FLAGS parameter, the record numbers of any successfully extracted records are removed from the list of record numbers in the SEARCH template. But if an error is encountered, the source record is not deleted and the record number is not removed from the list of record numbers in the SEARCH template.

.SCREENS

(Optional) Local array containing screen(s) to apply to subrecords at various subrecord levels. The screens determine whether to move individual subrecords at a given level or not. The screens can be any valid M code that sets $TEST to 1 if the subrecord at a given level should be moved, or 0 if not.

Set up nodes in this array subscripted by subfile# for each subrecord level you want to screen.

For a list of the variables you can reference and change in the screen, please refer to the SCREEN parameter description in the LIST^DIC DBS call.

Example:

```
S MYARRAY(999.01)="I $P(^(0),U,2)=""Y"""
```

.FILING_LEVEL

(Optional) Local array you can use to tell the Extract tool to file subrecords as **individual** transactions, at one or more subfile levels. The

default filing mode is to file an **entire** record—including all subrecords—as a single transaction.

You should consider using the FILING_LEVEL feature when extracting records with **many** subrecords at a given subfile level. This lets you restrict the scope of an extract transaction (every part of the transaction must succeed or the entire transaction fails) to individual subrecords rather than to a record **and** all of its subrecords.

For example, suppose the records you are extracting have one multiple field in particular in which there may be a thousand or more subrecords for every record. The subfile level of this multiple is a very good candidate to be filed individually:

- **Without** filing individually, failure to successfully extract any **one** of a record's thousand subrecords will abort the extract for the top level record and **all** of its subrecords (no changes will be filed).
- **With** filing individually, if any data in the subrecord causes an error, the subrecord will not be extracted, but the extract for the top level record and its other subrecords will continue.

Another drawback of filing a record and a large number of subrecords as a single transaction is that a very large FDA array can be created; this can be resource intensive and could exhaust scratch storage space in ^TEMP.

To file subrecords at any given subfile level individually, set up an array with a node subscripted by subfile# and pass the array by reference as this parameter. You can set more than one subfile level to file individually, by passing one node for each subfile level in the array.

Example:

```
S F_ARRAY(999.01)=""
```

TARGET_ROOT (Optional) Array that should receive the results generated during the extract tool process. This must be a closed array reference and can be either local or global. If you specify your own array for results, make sure it's empty before calling EXTRACT^DIAXU.

If you do not pass this parameter, the results are returned in ^TMP("DIAXU",$J).

MSG_ROOT (Optional) Array that should receive error messages generated during the extract tool process. This must be a closed array reference and can be either local or global. If you specify your own array for error messages, be sure it's empty before calling EXTRACT^DIAXU.

If you do not pass this parameter, error messages are returned in ^TMP("DIERR",$J).

Output

DIERR This variable is returned if an error condition occurred. It contains two ^-pieces of information:

1. Number of errors generated during the call
2. Total number of lines of the error messages

Associated error messages are stored, DBS-style, in MSG_ROOT.

TARGET_ROOT One "RESULT" node is returned for each record extracted (or attempted to be extracted).

The format of the "RESULT" node(s) for a successful extract is:

- TARGET_ROOT parameter passed

```
TARGET_ROOT("RESULT", source_file, source_ien) =
destination_file_ien
```

- No TARGET_ROOT parameter passed

  ```
  ^TMP("DIAXU", $J, "RESULT", source_file,
  source_ien) = destination_file_ien
  ```

The format of the "RESULT" node(s) for an unsuccessful extract is:

- TARGET_ROOT parameter passed

  ```
  TARGET_ROOT("RESULT", "ERR", source_file,
  source_ien) = error_list
  ```

- No TARGET_ROOT parameter passed

  ```
  ^TMP("DIAXU", $J, "RESULT", "ERR", source_file,
  source_ien) = error_list
  ```

The error list for an unsuccessful extract contains the error numbers, each followed by a semicolon. For example, if a "RESULT" node is:

```
TARGET_ROOT("RESULT", "ERR", 16151, 6)=1;2;
```

This means that errors 1 and 2 are caused by the extract of record 6. Errors one and two are returned in the MSG_ROOT array.

If the FILING_LEVEL parameter is being used such that subrecords are being filed individually at some subfile levels, results (successful or unsuccessful) are returned for each individual subrecord extracted, in the same format as above, except that:

- "source_file" will be the subfile number
- "source_ien" will be the IENS string for the subfile entry
- "destination_file_ien" will be the IENS string for the destination subfile entry

If one or more subrecords extracted **unsuccessfully** using the FILING_LEVEL parameter, a single error (1300) is returned for the top-level record in a "RESULT","ERR" node, but this does not abort the extract. So in this case a top-level extracted record can have both a "RESULT" node (indicating

success at the top level and the destination file ien) and a "RESULT","ERR" node (indicating error[s] during subfile filing).

If the extract fails for any subrecord at some subfile level **not** filed individually via the FILING_LEVEL parameter, a "RESULT","ERR" node is returned for the top-level record, and the extract for the top-level record aborts.

MSG_ROOT

Error messages are returned in MSG_ROOT("DIERR") (if the MSG_ROOT input parameter is passed) or ^TMP("DIERR",$J) (if no array is specified). Errors are returned in DBS-style format.

For more information on the format of DBS-style error arrays, see the "DIERR" section of "Contents of Arrays" in the "How to Use the Database Server" section in this manual.

Examples

Example 1

In this example, EXTRACT^DIAXU is called with a SEARCH template containing a list of three record numbers to extract. Two records (#7 and #32) are moved successfully and one record (#34) fails to be moved. As a result of the error, the variable DIERR would be returned (set to "1^1"). The call might look like:

```
D EXTRACT^DIAXU(16151,"[EXTRACT SEARCH]","[EXTRACT TEMPLATE]")
```

The results messages would be returned as follows:

```
^TMP("DIAXU",627068728,"RESULT",16151,7) = 1
^TMP("DIAXU",627068728,"RESULT",16151,32) = 13
^TMP("DIAXU",627068728,"RESULT","ERR",16151,34) = 1;
```

The error messages would be returned as follows:

```
^TMP("DIERR",627068728,1) = 701
^TMP("DIERR",627068728,1,"PARAM",0) = 3
^TMP("DIERR",627068728,1,"PARAM",3) = NEWONE
^TMP("DIERR",627068728,1,"PARAM","FIELD") = .01
^TMP("DIERR",627068728,1,"PARAM","FILE") = 16299
^TMP("DIERR",627068728,1,"TEXT",1) = The value 'NEWONE' for
   field NAME in file FTEXT EXTRACT is not valid.
^TMP("DIERR",627068728,"E",701,1) =
```

Example 2

Suppose that the call to EXTRACT^DIAXU is made using the FILING_LEVEL array. This means that subrecords at some subfile levels are extracted individually. Let's suppose only one record is being extracted (ien #5), and two subrecords are extracted individually with the FILING_LEVEL array. Subrecord #1 extracts successfully, and subrecord #2 fails. The results and error messages would be returned as follows:

```
^TMP("DIAXU",541074770,"RESULT",662001,5) = 75              (record #5,
                                                             success)
^TMP("DIAXU",541074770,"RESULT",662001.1,"1,5,") = 1,75,    (subrecord #1,
                                                             success)
^TMP("DIAXU",541074770,"RESULT","ERR",662001,5) = 2         (record #5,
                                                             error 2 from
                                                             subrecord
                                                             failure)
^TMP("DIAXU",541074770,"RESULT","ERR",662001.1,"2,5,") = 1; (subrecord #2,
                                                             error 1)
^TMP("DIERR",541074770,1) = 330                             (error 1)
^TMP("DIERR",541074770,1,"PARAM",0) = 2
^TMP("DIERR",541074770,1,"PARAM",1) = 99
^TMP("DIERR",541074770,1,"PARAM",2) = pointer to File #200
^TMP("DIERR",541074770,1,"TEXT",1) = The value '99' is not
  a valid pointer to File #200.
^TMP("DIERR",541074770,2) = 1300                             (error 2)
^TMP("DIERR",541074770,2,"PARAM",0) = 1
^TMP("DIERR",541074770,2,"PARAM","IEN") = 5
^TMP("DIERR",541074770,2,"TEXT",1) = "The entry encountered an err
  or during subfile filing.
^TMP("DIERR",541074770,"E",330,1) =
^TMP("DIERR",541074770,"E",1300,2) =
```

Error Codes Returned

This entry point calls $$FIND1^DIC, LIST^DIC, UPDATE^DIE, $$GET1^DIQ, and GETS^DIQ; any errors returned by these entry points can also be returned by EXTRACT^DIAXU.

In addition, the following errors may be returned:

202 An input parameter is missing or not valid.

601 The entry does not exist.

1300 The entry encountered an error during subfile filing.

9. Filegrams API

INTRODUCTION

Filegrams are a feature in VA FileMan intended for use by system managers, package developers, and programmers.

A Filegram is a process that moves a record (also called an entry) from a file on one computer system to a duplicate file on another **independent** computer system. An independent computer system is defined as a system having its own database. Sending data from the "live" account at a medical center to the "test" account at the same medical center is an example of moving a Filegram locally. Sending data from a computer in the San Francisco Medical Center to a computer in the Salt Lake City Medical Center is an example of moving a Filegram remotely.

The Filegram process consists of the following three components:

1. Filegram generator (the DIFGG routines)
2. Filegram installer (the DIFG routines)
3. FILEGRAM template (stored in the PRINT template file)

Although there is a set of options to work with Filegrams, you, as a programmer, will find that the only routines necessary to process a Filegram are the installer and the generator routines which are described in this chapter as ^DIFG and EN^DIFGG, respectively.

For more information about Filegrams, see the "Filegrams" chapter of *VA FileMan Advanced User Manual*.

^DIFG: Installer

The Filegram process consists of the following three components:

1. Filegram generator (the DIFGG routines)
2. Filegram installer (the DIFG routines)
3. FILEGRAM template (stored in the PRINT template file)

You, as a programmer, will find that the only routines necessary to process a Filegram are the installer and the generator routines.

The key variables DUZ, DUZ(0), and DT must be present in addition to the required variables described below.

Use the ^DIFG entry point to install Filegrams. The installer part of the Filegram requires the DIFGLO variable in addition to the VA FileMan key variables mentioned just above. The other input variables are optional.

D ^DIFG will install the Filegram.

Input Variables

DIFGLO (Required) This variable must be the global root of the Filegram to be installed.

DIADD (Optional) If this variable is defined, a new entry will be created in the base file.

DINUM (Optional) Entry number in base file at which new file entry, if added, will be created.

Output Variables

DIFGER This output variable is defined if an error has occurred.

NOTE: It will be defined even if the install fails after the base file has been processed. Thus, it could exist even if DIFGY is not equal -1. See below for a list of error codes that will be found in DIFGER.

DIFGY ^DIFG always returns DIFGY. DIFGY can have one of the following values:

DIFGY=-1 Indicates that the lookup on the initial file processed (the base file) was unsuccessful.

DIFGY=N^F Where N is the internal number of the entry in the base file and F is the base file's number.

DIFGY=N^F^1 Where N and F are defined as above and 1 indicates that a new entry has been added to the base file.

Error Codes Returned in DIFGER

If a soft error occurs, the variable DIFGER is defined when the Filegram routines are exited. This variable contains information about the problem encountered. It consists of two ^-pieces. The first piece indicates the error number. The second piece usually contains a line number in the Filegram that indicates where the Filegram process failed.

Here is a list of the codes found in DIFGER along with their specific meanings:

1^0 The Filegram global root was not passed in DIFGLO.

1.25^0 The Filegram global root format is invalid.

1.5^0 The Filegram global root is passed but the global does not exist.

2^1 The first line of the Filegram does not contain a $DAT.

3^# A line other than the first line has a $DAT as its first colon-piece.

4^# The field does not exist within this file.

5^# ^%DT was called and Y was returned equal to -1.

6^# Line after a context switch, subfile; and any field that required a lookup was not a BEGIN condition.

7^# DINUM variable exists, the mode is A or L, and the INPUT transform contains the word DINUM (files or subfiles only).

8^# DINUM or DIADD variables exist and the mode is neither A nor L (files or subfiles only).

9^# File or subfile lookup failed and mode type will not permit addition of an entry to this file. In other words, the mode type was either D or M.

10^# Lookup failed during a context or subfile shift, the .01 field of the file or subfile is a pointer, and LAYGO to the pointed-to file is not allowed. This code is also generated if lookup failed and LAYGO is not allowed for a pointer that is an identifier or specifier.

11^# A lookup for a single valued pointer field fails and LAYGO is not allowed.

12^# A lookup failed for a file or subfile and the mode is M.

13^# There is a key for a given entry and the internal entry number was not found in the cross-reference or the cross-reference did not exist.

14^# ^DIE called for a MODIFY or DELETE Filegram and Y was returned defined.

15^# ^DIE called for an entry which was an ADD and Y was returned as defined.

16^# Call to ^DIC or FILE^DICN and Y was returned equal to -1. Error occurred during installation.

17^# Entry of a word processing field failed.

18^# Lookup failed when a "B" index lookup was specified and the B cross-reference did not exist.

19^# DINUM was passed to DIFG, the mode of the base line file was M or D, and the entry did not exist in the base line file.

20^# File does not exist.

21^# A field has an "@link" value which is unresolved and will not be LAYGOed to the pointed-to file during installation.

EN^DIFGG: Generator

The Filegram process consists of the following three components:

1. Filegram generator (the DIFGG routines)
2. Filegram installer (the DIFG routines)
3. FILEGRAM template (stored in the PRINT template file)

You, as a programmer, will find that the only routines necessary to process a Filegram are the installer and the generator routines.

The key variables DUZ, DUZ(0), and DT must be present in addition to the required variables described below.

In order to create (or generate) a Filegram, D EN^DIFGG with the key variables just above and the required input variables listed below. DUZ should refer to a valid user. The optional input variables can be used to customize the Filegram.

Input Variables

DIFGT (Required) This variable must equal the internal entry number in the PRINT template file of the FILEGRAM template that defines the data to be sent.

DIFG("FE") (Required) This variable must equal the internal number in the base file of the entry to be sent.

DIFG("FUNC") (Required) This variable must equal A, M, L, or D. The meanings of these codes, which indicate the mode of the Filegram, are:

A	**A**DD (force an add)
M	**M**ODIFY
L	**L**AYGO
D	**D**ELETE

DIFG("FGR") (Optional) Set this variable equal to the root of the global or local array in which the Filegram will be built. The default is ^UTILITY("DIFG",$J, if this variable is not defined.

DILC (Optional) One fewer than the first subscript to generate. Default=0.

DITAB (Optional) Initial indentation level for Filegram text.

Output Variables

DIFGER This output variable is defined if an error has occurred. Its possible values are:

- A required variable was not passed.
- A variable's format is invalid.
- A variable's content is invalid.

Part IV: Developer Tools

10. ^DI: Programmer Access

Often, VA FileMan's options are accessed through a menu system that calls up the main FileMan menu. For example, if Kernel is installed, FileMan can be entered from Kernel's menu system if a user has been granted access.

However, the main menu can also be displayed directly from the M command prompt. When you call VA FileMan directly, you are using "programmer mode."

There are four entry points in the DI routine that you can use to enter VA FileMan. Each way of calling up the main menu has a different effect upon the local M variables that are defined when you begin your FileMan session. They are described below:

P^DI This entry point cleans the symbol table; that is, it kills all local variables except those that are required for FileMan's operation. (The variables DUZ and DTIME are unchanged.) In addition, the variable DUZ(0) is set equal to @. The @-sign gives you complete programmer access to all of FileMan's files and functionality.

NOTE: Included in the variables killed are the IO variables.

Q^DI Like P^DI, this entry point sets DUZ(0)="@". However, the remaining variables in the local symbol table are unchanged.

C^DI Like P^DI, this entry point cleans the symbol table. However, it leaves DUZ(0) unchanged; whatever Access Code string was in DUZ(0) before the call remains to control access within FileMan.

D^DI This entry point leaves all local variables alone. It neither cleans the symbol table nor resets DUZ(0).

In addition, other necessary variables are set to default values if they are undefined when you start VA FileMan from programmer mode.

11. Global File Structure

INTRODUCTION

This chapter describes the storage of VA FileMan files, including the file structure and the actual file data.

Throughout this chapter, these basic components of a FileMan file are described by way of an example: how the rudiments of an EMPLOYEE file would be mapped into a global called ^EMP using FileMan. File number 3 is assigned to this file in the examples.

DATA STORAGE CONVENTIONS

VA FileMan stores the data of every file descendent from a single M global array (or from a node of a global array). When the routines, internally and externally, make reference to a file in a global notation, FileMan expects the following format:

`^GLOBAL(`	for an entire global
`^GLOBAL(X,Y,`	for a subtree of a global

NOTE: A global notation must terminate with an open parenthesis (() or a comma (,). Indirection (@) is always used by FileMan routines when referring to data files.

For the most part, FileMan packs data into subscripts using the up-arrow (^) character as the $PIECE delimiter. You refer to a data element as being stored in the nth ^-piece of a global node.

FILE'S ENTRY IN THE DICTIONARY OF FILES

All VA FileMan files, regardless of the global used for data storage, have an entry in the Dictionary of Files—the ^DIC global descendent from the file's DD number.

The **zero subscript** contains the file name and file number.

The **global location (GL)** node descendent from subscript zero is set to the root of the global used to store data for this file. So, the EMPLOYEE file example may have the following:

```
^DIC(3,0)      = "EMPLOYEE^3"
^DIC(3,0,"GL") = "^EMP("
```

The ^DIC global also contains the file's **security protection** codes, if any, descendent from the zero subscript in the following nodes:

```
^DIC(filenumber,0,"AUDIT")      --  Audit Access
^DIC(filenumber,0,"DD") --  Data Dictionary Access
^DIC(filenumber,0,"DEL")        --  Delete Access
^DIC(filenumber,0,"LAYGO")      --  LAYGO Access
^DIC(filenumber,0,"RD") --  Read Access
^DIC(filenumber,0,"WR") --  Write Access
```

The rest of the ^DIC global descriptors for a file are:

```
^DIC(filenumber,"%",    --  At lower subscript levels, contains the
          application groups.
^DIC(filenumber,"%A")   --  Creator's DUZ^file creation date.
          DIFROM does not send this node.
^DIC(filenumber,"%D",   --  At lower subscript levels, contains the
          text of the file's DESCRIPTION.
```

FILE HEADER

A descriptor string is stored in the zero subscript of the file's global root-^EMP(in our example. This is simply an ^-piece-delimited string containing the following:

piece 1 file name

piece 2 file number with file characteristics codes

piece 3 most recently assigned internal entry number

piece 4 current total number of entries

NOTE: The most recently assigned number is not necessarily the largest entry number. The file number is the record number of the file in the attribute (or data) dictionary that describes the data fields for this file. Thus, if a file has three employees and if the file's most recently added employee was assigned entry number 9, we have:

```
^EMP(0)="EMPLOYEE^3I^9^3"
```

The data dictionary number (second ^-piece) may also be followed by a string of alphabetic characters that are used by VA FileMan as flags to indicate various characteristics of the file. This string may contain the following:

D .01 field of the file is a **D**ate/Time.

P .01 field of the file is a **P**ointer to another file.

S .01 field of the file is a **S**et of Codes.

V .01 field of the file is a **V**ariable Pointer.

A **A**utomatically adds entries without asking: "ARE YOU ADDING A NEW ENTRY?"

I File has **I**dentifiers.

O The user will be asked "...**O**K?" whenever a matching entry is found during lookup.

s (lowercase s) File has a **s**creen defined in ^DD(filenumber,0, "SCR").

ATTRIBUTE DICTIONARY

File Characteristics Nodes

Certain file characteristics are kept in the subtree descendent from ^DD(filenumber,0,. These characteristics with their subscripted location and brief explanation are:

Global Node	Meaning
^DD(filenumber,0,"ACT")	Post-Action
^DD(filenumber,0,"DDA")	Data Dictionary Audit
^DD(filenumber,0,"DIC")	Special Lookup
^DD(filenumber,0,"ID",field)	Field identifiers
^DD(filenumber,0,"ID","WRITE")	Write identifiers
^DD(filenumber,0,"IX",x-ref name,(sub)filenumber,field)	Cross-references
^DD(filenumber,0,"SCR")	Screens
^DD(filenumber,0,"VR")	Version Number
^DD(filenumber,0,"VRPK")	Distribution Package
^DD(filenumber,0,"VRRV")	Package Revision Data

Post-Action

```
^DD(filenumber,0,"ACT")
```

After an entry has been selected, some action can be taken to examine or verify the selection. This executable code is stored at this global location. If you decide that the

entry should not be selected, set Y=-1. See the "Advanced File Definition" chapter in this manual.

Data Dictionary Audit

```
^DD(filenumber,0,"DDA")
```

This node is set to "Y" if auditing is turned on for the data dictionary. The node is either nonexistent, null, or set to "N" if data dictionary auditing is not on.

Special Lookup

```
^DD(filenumber,0,"DIC")
```

A special lookup program can be written to facilitate selection from a particular file. If such a program is to be used, its name is stored at this location.

NOTE: The lookup program reference cannot be a labeled entry point to a routine. The routine's name is stored without the up-arrow (^); it cannot begin with DI. See the "Advanced File Definition" chapter in this manual.

Field Identifiers

```
^DD(filenumber,0,"ID",field)
```

Field Identifiers are defined using the FileMan Identifier option. The value at the node is a WRITE statement. If the identifier is to be used only to ask fields when a new entry is added, then the statement will only write null; otherwise, it will contain the code to write the external value of the field. An "I" is added to the second piece of the File Header when you add a field identifier, (as described above).

NOTE: WRITE statements in the FileMan-generated field identifier nodes are **not** executed when VA FileMan is in silent mode. Since FileMan generates the Field Identifier nodes, it knows their format. So in silent mode, FileMan places what would have been Written into an array instead for use by DBS calls. So Field Identifier nodes, although they contain WRITE statements, are compatible with GUI applications (in contrast to Write Identifier nodes).

Write Identifiers

```
^DD(filenumber,0,"ID","ASTRING")
^DD(filenumber,0,"ID","W1")
^DD(filenumber,0,"ID","W2")
^DD(filenumber,0,"ID","WRITE1")
```

You can use M code to define addtional custom identifier text to be displayed along with field identifiers. To do this, add "write identifier" node(s) one level descendent from ^DD(filenumber,0,"ID"). The write identifier nodes you add must be subscripted with strings that begin with an **uppercase alphabetic** character.

Set the value of each write identifier node to the M code that will produce the desired output. Write out your output using either the EN^DDIOL entry point (preferred) or the M WRITE command (not compatible with access to your file by GUI applications). In your M code for each write identifier node, you can refer to the following values which will be defined at the time the node is executed:

Y	Current record number
Naked Reference	Set to the 0-node of the entry

Write identifiers are displayed after any field identifiers are displayed. If there is more than one write identifier, they are displayed in the collating order of the write identifier subscripts.

Since you must **hard-set** any "WRITE" nodes, you must also add an "I" (if one isn't already there) to the second piece of the File Header.

Cross-references

```
^DD(filenumber,0,"IX",x-ref name,(sub)filenumber,field)
```

For cross-references, this node is set equal to null.

In Version 22 of FileMan, a new INDEX file has been introduced as an alternate way to define indexes on a file. The information is descendent from ^DD("IX". See a description below under the "INDEX FILE" section.

Screens

```
^DD(filenumber,0,"SCR")
```

If you want to screen access to entries in a file, set the screen code into this node. The screen should be written like a screen put into the local variable DIC("S") for an ^DIC call. The code in this node is executed for each entry in the screened file. If $T=0 is returned when the node is executed, the entry being screened is unavailable for lookups, prints, inquiries, searches, or other actions.

In order for the screen in this global to be used, you must put a lower-case "s" into the second piece of the file's header following the file number (as described above).

Version Number

```
^DD(filenumber,0,"VR")
```

This node is created during an INIT built by the VA FileMan package distribution routine (DIFROM) or an installation using the Kernel Installation and Distribution System (KIDS). It contains the current version number for the package that distributes this file. This node and the Distribution Package node are updated for any file sent by a KIDS installation. The only time these nodes are not updated is when a partial DD is sent. (For additional information, see the KIDS section in the Kernel Systems Manual, beginning with V. 8.0.)

Distribution Package

```
^DD(filenumber,0,"VRPK")
```

This node is created during an installation using the Kernel Installation and Distribution System (KIDS). It contains the name of the package that distributes this file. The only time this is not updated is when a partial DD is sent. (For additional information, see the KIDS section in the Kernel Systems Manual, beginning with V. 8.0.)

Package Revision Data

```
^DD(filenumber,0,"VRRV")
```

This optional node, if present, is created during an installation using the Kernel Installation and Distribution System (KIDS). The node is defined by the developer who distributes the package. It may contain patch or other package revision

information used to designate the version of the file that is installed at the site. Updating this node is done in the KIDS Post Install Routine (formerly the POST-INIT with DIFROM/INITS) using PRD^DILFD( ). (For additional information, see the KIDS section in the Kernel Systems Manual, beginning with V. 8.0.

Field Definition 0-Node

Each entry in the attribute dictionary is a descriptor of one of the data fields in the file. VA FileMan always assigns the internal number .01 to the NAME field and lets you assign numbers to the other fields. The attribute dictionary stores the definition of each field descendent from the node ^DD(filenumber,fieldnumber). Crucial information about the field is stored in:

```
^DD(filenumber,fieldnumber,0)
```

Every field has this 0-node defined in the attribute dictionary.

In our example, the EMPLOYEE file has four fields in addition to the NAME field: SEX, BIRTHDATE, DEPARTMENT, and SKILL. SKILL is multiple-valued. Let us suppose that the attribute dictionary for this file is stored in:

```
^DD(3)
```

Piece 1 The field's label is always found as the first ^-piece in subscript zero. Thus, for our example, you would have:

```
^DD(3,.01,0)="NAME^"
^DD(3,1,0)="SEX^"
^DD(3,2,0)="DOB^"
^DD(3,3,0)="DEPARTMENT^"
^DD(3,4,0)="SKILL^"
```

Piece 2 A string containing any of the following letters and symbols:

Character	Meaning
a	The field has been marked for **a**uditing all the time.
e	The auditing is only on **e**dit or delete.
A	For multiples, a user entering a new subentry is not **A**sked for verification.

Character	Meaning
BC	The data is **B**oolean **C**omputed (true or false).
C	The data is **C**omputed.
Cm	The data is **m**ultiline **C**omputed.
D	The data is **D**ate-valued.
DC	The data is **D**ate-valued, **C**omputed.
F	The data is **F**ree text.
I	The data is uneditable.
Jn	To specify a print length of **n** characters.
Jn,d	To specify printing **n** characters with **d**ecimals.
K	The data is M code.
M	For **M**ultiples, after selecting or adding a subentry, the user is asked for another subentry.
N	The data is **N**umeric-valued.
O	The field has an **O**UTPUT transform.
Pn	The data is a **P**ointer reference to file "**n**".
Pn'	LAYGO to the **P**ointed-to file is **n**ot allowed.
R	Entry of data is **R**equired.
S	The data is from a discrete **S**et of codes.
V	The data is a **V**ariable pointer.
W	The data is **W**ord processing.

Character	Meaning
WL	The **W**ord processing data is normally printed in **L**ine mode (i.e., without word wrap).
X	Editing is not allowed under the Modify File Attributes option because the INPUT transform has been modified by the Input Transform option on the Utility Functions submenu.
*****	If there is a screen associated with a pointer or set of codes data type.

NOTE: The second ^-piece begins with the subfile number if the field is a multiple.

Piece 3 Only contains data for Pointer and Set of Codes data types. In those cases, the data is:

Pointer	The global root of the pointed-to file.
Set of Codes	The set of codes of allowed responses and their meanings.

Piece 4 One of the following, based on the kind of data storage:

- subscript location and ^-piece, separated by a semicolon (;)
- subscript location and character-positions, also separated by a semicolon (;), where Em,n designates character-positions m through n
- subscript location, followed by a semicolon (;) followed by 0 (zero), to designate multiple-valued data, and
- semicolon preceded and followed by a space (' ; ') to indicate no data storage, i.e., computed fields

Piece 5 M code to check an input in the variable X. If the input is invalid, the variable X is killed by the code. This is the field's INPUT transform. In the case of a computed field, the code creating the variable X is stored here. (Pieces following the fifth piece are part of this M code.)

Other Field Definition Nodes

Every field must have a zero node. All other nodes describing a field are presented below, but none are mandatory. Each subscript listed is at least the third level—that is, the global reference appears in the following format: ^DD(File#,Field#,Subscript).

Subscript	Definition
.1	Contains the full-length title of the field.
1	Contains, at lower subscript levels, executable M code to set and kill cross-references based on the value of the field (in the variable X).
2	Contains the OUTPUT transform: M code to display the field value in a format that differs from the format in which it is stored. (See the "Advanced File Definition" chapter in this manual.)
3	Contains the help prompt message that is displayed when the user types a question mark.
4	Contains M code that will be executed when the user types one or two question marks. (Other help messages are also displayed.)
5	Contains, at lower subscript levels, pointers to trigger cross-references to this field.
7.5	Is valid only on .01 fields. It contains M code that will be executed to check the user input (in the variable X). This code is executed at the start of the ^DIC routine before the lookup on X has begun. If X is killed, the lookup will terminate. Special lookup programs naturally have a way to look at X.
8	Read access for the field.
8.5	Delete access for the field.
9	Write access for the field.

9.01 The fields used if the field is a computed field.

9.1 The expression entered by the user to create the computed field.

9.2 to 9.9 The overflow executable M code that may be part of the specification of a field definition, INPUT transform, or cross-reference.

10 Contains the source of the data.

11 Contains the destination of the data.

12 Contains the explanation of the screen on node 12.1.

12.1 Contains the code which sets DIC("S") if a screen has been written for a pointer or a set of codes.

20 A multiple that lists the fields that belong to certain groups.

21 A word processing field that holds the field description.

22 The name of a help frame presented to a user who entered two question marks. **NOTE:** This subscript is being phased out.

23 A word processing field that holds the technical description of the field.

AUDIT Contains a code defining the status of an audit trail for changes to the data in the field. Possible codes are: y (always audited), e (changes and deletions only audited), n (no audit recorded).

AX Contains the executable code that determines if a field should be audited.

DEL In this example, a string of executable M code that determines if the field can be deleted. This code must contain an M IF statement to set the value of $T. If $T is set to 1, the field cannot be deleted. Normally, the ^DD format is:

```
^DD(File#,Field#,"DEL",#,0)="executable MUMPS code"
```

where **#** is an arbitrary number to distinguish each condition. If the condition was based on a particular field, then the field

number was traditionally used. If "DEL" nodes are on the .01 field of a file, deletion of the entire entry can be blocked.

If an entry is being deleted by a direct call to ^DIK, the "DEL" nodes are not checked.

DT Contains the date the field was last edited.

LAYGO A string of executable M code that determines if an entry can be added. This code must contain an M IF statement to set the value of $T. If $T is set to 0, the entry cannot be added. Normally, the ^DD format is:

```
^DD(File#,.01,"LAYGO",#,0)="executable MUMPS code"
```

where **#** is an arbitrary number to distinguish each condition. LAYGO nodes only apply to .01 fields.

V Descendent from these nodes is information regarding variable pointers including pointed-to file, message, order, prefix, screen, and LAYGO status.

```
^DD(File#,Field#,"V",n,0)
```

Where "**n**" is a sequential number representing a different pointed-to file. The pieces within this 0 node are:

^-Piece	**Contents**
Piece 1	File number of the pointed-to file.
Piece 2	Message defined for the pointed-to file.
Piece 3	Order defined for the pointed-to file.
Piece 4	Prefix defined for the pointed-to file.
Piece 5	y/n indicating if a screen is set up for the pointed-to file.
Piece 6	y/n indicating if the user can add new entries to the pointed to file.

^DD(File#,Field#,"V",n,1) Contains the M code defined as a screen on the pointer to the file defined in the 0 node above.

^DD(File#,Field#,"V",n,2) contains a description of the screen.

How to Read the Attribute Dictionary: An Example

Each attribute dictionary is stored descendent from a positive-valued, first-level subscript of this global. Each attribute dictionary, in itself, is also in the form of a file and thus consists of entries, cross-references, descriptor, and a reference to the data dictionary of the attribute of attributes (^DD(0)).

Following are the ^DD nodes associated with our sample EMPLOYEE file:

```
^DD(3,.01,0)="NAME^FR^^0;1^I X'?1A.AP1",".AP K X"
^DD(3,.01,.1)="EMPLOYEE NAME"
^DD(3,.01,1,0)="^.1^1^1"
^DD(3,.01,1,1,0)="3^B"
^DD(3,.01,1,1,1)="S ^EMP("B",$E(X,1,30),DA)="""
^DD(3,.01,1,1,2)="K ^EMP("B",$E(X,1,30),DA)"
^DD(3,.01,3)="NAME MUST BE 3-30 CHARACTERS, IN THE FORMAT LAST,FIRST"
^DD(3,1,0)="SEX^RS^M:MALE;F:FEMALE^0;2^Q"
^DD(3,2,0)="DOB^D^^0;3^S %DT="EX" D ^%DT S X=Y I X<1400000 K X"
^DD(3,2,.1)="DATE OF BIRTH"
^DD(3,3,0)="DEPARTMENT^P13'^DIZ(13,^0;4^Q"
```

Their meaning can be translated to:

The first field is NAME (full title: EMPLOYEE NAME). It is free text data which must start with at least one alpha followed by other alpha and punctuation characters and contains a comma. It is always required from the user and is stored in subscript 0, ^-piece 1 of each employee's entry. If the user types a question (?) when asked for the NAME, the user will see:

```
NAME MUST BE 3-30 CHARACTERS, IN THE FORMAT LAST,FIRST
```

The EMPLOYEE file is cross-referenced by NAME so every time a name is changed, the corresponding subscript under ^EMP("B") is also changed. DA will always be the internal number of the employee when the cross-referencing code is executed. If a second cross-reference for NAME existed (for example, a trigger), it would be descendent from:

```
^DD(3,.01,1,2
```

The second field is SEX. It is stored as either M or F in the second ^-piece position of subscript 0 of each EMPLOYEE file entry. The user is required to respond and can type MALE instead of M, and FEMALE instead of F. The user will see the two choices displayed if a ? is typed when asked for SEX.

The third field is DOB (full title: DATE OF BIRTH). It is not required. If entered, it must be in the format of a date after 1840. It is stored in the third ^- piece of subscript 0 of the EMPLOYEE file entry.

The fourth field is DEPARTMENT; it is not required. It is a pointer to file number 13 and adding new entries (LAYGO) to the DEPARTMENT file from the EMPLOYEE file is NOT allowed as indicated by the apostrophe (') after the number 13. It is stored in the fourth ^-piece of subscript zero of the EMPLOYEE file entry. The internal value of the Employee's department in the DEPARTMENT file is stored in this location. The data of the DEPARTMENT file can be found in ^DIZ(13,.

Suppose there is also a multi-valued field, SKILL. A multiple-valued field is described by a separate data dictionary. FileMan creates this new data dictionary descendent from a non-integer subscript of ^DD. In the case of the EMPLOYEE file described by ^DD(3), it would store subsidiary data dictionaries in ^DD(3.01), ^DD(3.02), etc. The subsidiary data dictionary for the multiple-valued SKILL field could look like this:

```
^DD(3.01,0)="SKILL subfield^^1^2"
^DD(3.01,.01,0)="SKILL^MF^^0;1^K:$L(X)>30!($L(X)<3) X"
^DD(3.01,.01,3)="ANSWER MUST BE FROM 3 TO 30 CHARACTERS IN LENGTH"
```

The only new element here is the M in the second ^-piece of ^DD(3.01,.01,0). This is the flag corresponding to the 'YES' answer to the question:

```
HAVING ENTERED OR EDITED ONE SKILL, SHOULD USER BE ASKED
  ANOTHER?
```

If you answer 'YES' to this question, each time the user enters data, the "Select SKILL:" prompt will be repeated until the user enters a null response. There will also be an entry corresponding to SKILL in the principal EMPLOYEE file's data dictionary as follows:

```
^DD(3,4,0)="SKILL^3.01A^^SX;0"
```

The 3.01 points to the subsidiary data dictionary of that number; it says that, to find the data descriptors of SKILL (and all fields pertaining to SKILL), we must look in ^DD(3.01). The A indicates that every time the user enters a new SKILL, it will be automatically added to the file and the user will not be asked:

```
ARE YOU ADDING A NEW SKILL?
```

The SX;0 in the fourth ^-piece tells us the entire SKILL subfile will be stored descendent from the SX subscript in each employee's record.

INDEX File

Version 22 of VA FileMan introduces a new way to define an index (cross-reference) on a file. The option to Cross-Reference a File within the Utilities option will now ask whether the developer wishes to add/edit a Traditional index or a New-style index. Use of the INDEX file allows for design of more sophisticated indexes, including compound indexes (i.e., with more than one data field subscript), indexes where transforms are done on fields, indexes with computed subscripts, indexes whose normal collation sequence is backward, and indexes whose set/kill logic is executed once per record rather than once per field. These indexes can then be used by the FileMan code for such things as looking up a record on the file. The INDEX file stores all information describing the new indexes. Data is stored descendent from ^DD("IX"). The INDEX file is #.11 (stored in global ^DD("IX",). See also KEY File.

KEY File

Version 22 of VA FileMan introduces a way to uniquely identify a record on a file. The developer defines a field or fields as belonging to a KEY. The developer must also build an index for those fields. Fields in the Primary KEY are displayed during classic FileMan lookup ^DIC. KEY fields are used to decide whether a record already exists on the target file during Transfer or during data Installation using the Kernel Installation and Distribution System (KIDS). The KEY file is #.31 (stored in global ^DD("KEY",). See also INDEX File.

File Entries (Data Storage)

Each entry in a file corresponds to a positive-valued key subscript, the internal entry number, of the file global. All data pertaining to an entry will be stored in global nodes descendent from that subscript. The value of the .01 field of an entry is always stored in the first ^-piece of subscript zero, descendent from the internal entry number subscript. Thus, for entry #1, an employee named John Jones, you would have:

```
^EMP(1,0)="JONES,JOHN^"
```

Suppose you want to store the employee's sex in the second ^-piece of subscript zero, and date of birth in the third ^-piece, and department in the fourth ^-piece. You would have:

```
^EMP(1,0)="JONES,JOHN^M^2341225^3"
```

Notice that the entry for the employee's department in this file is a number. This means that the employee's department is internal entry number 3 in the Department file; and to find the employee's department, you would have to consult that file. The 7-digit number representing the employee's date of birth is FileMan's way of internally representing 12/25/1934.

How is multiple-valued data, such as skill, stored? There can be one or five or ten skills on file for a given employee and they obviously cannot all be stored (in the general case) in a single subscript. VA FileMan's answer is to make the skills list a subfile within the employee entry. This requires adding subscripts beyond the first internal key subscript which are different in value from the zero subscript that stores each employee's name, sex, and birth date. For example, if John Jones currently has two (free-text) skills on file, you can consider those to be entries #1 and #2 in a two-entry file, which can extend at a lower level from any unused subscript, say from SX as shown below:

```
^EMP(0)="EMPLOYEE^3I^9^3"
^EMP(1,0)="JONES,JOHN^M^2341225^3"
^EMP(1,"SX",0)="^3.01A^2^2"
^EMP(1,"SX",1,0)="TYPING"
^EMP(1,"SX",2,0)="STENOGRAPHY"
```

Notice that the data global ^EMP has ^EMP(1,"SX",0) for the skill multiple. The zero node, except for the first ^-piece, has the same structure as ^EMP(0). The second ^-piece is the subfile ^DD number. This tells FileMan which subsidiary dictionary to use for the data stored in this node. The actual data (the employee's skills in our example) are stored in the next lower level of subscripting. In the same manner that entries in the EMPLOYEE file have internal entry numbers, entries in the multiple field also have internal entry numbers in the subfile. In the example above, TYPING is the first entry and STENOGRAPHY the second.

Cross-references

The M capabilities of string-valued array subscripting offer a simple, general way to cross-reference VA FileMan files. To minimize the number of global names used by the system, FileMan stores each cross-reference set as a descendent of an alphanumeric subscript of the file's global. A file, such as an EMPLOYEE file, that should be accessible by name, is set up by the system so that there is a subscript "B", which in turn is subscripted by strings corresponding to the first 30 characters

from the .01 field for every entry in the file. For each such string-valued subscript, the next level of subscripting contains the internal entry numbers of the entries that contain the name.

Let's add to our previous example a second employee, internal entry number 9, also named John Jones, and a third employee, internal number 7, whose name is Sam Smith. Then you would have:

```
^EMP(1,0)="JONES,JOHN^M^2341225^3"
^EMP(7,0)="SMITH,SAM^M^2231109^2"
^EMP(9,0)="JONES,JOHN^M^2500803^18"
^EMP("B","JONES,JOHN",1)=""
^EMP("B","JONES,JOHN",9)=""
^EMP("B","SMITH,SAM",7)=""
```

Notice that all the data is in the subscripting and the global nodes under ^EMP("B") are simply null strings. FileMan allows for these strings to be non-null in the case where a mnemonic cross-reference is set up for the name. Multiple cross-references (C, D, etc.) are also allowed.

In Version 22.0 of VA FileMan, cross-references (indexes) may be defined that have more than one data field subscript before the record number. These cross-references can then be used for a lookup and the user will be prompted for more than one lookup value, one for each data subscript on the index. Such compound indexes must be defined as a new-style index on the INDEX file (described above).

Example: An entry from a cross-reference with the name and the date-of-birth on the data above might look like:

```
^EMP("C","JONES,JOHN",2341225,1)=""
```

12. Advanced File Definition

INTRODUCTION

When FileMan routines are invoked with the local variable DUZ(0) set to the @-sign, the user is understood by FileMan to be an M-proficient programmer who has "programmer access." Those working with programmer access can control certain file-definition options that are otherwise handled invisibly by FileMan. These features are described in this chapter.

See the "Creating Files and Fields" chapter of the *VA FileMan Advanced User Manual* for a description of the file and field definition options available to everyone.

FILE GLOBAL STORAGE

Storing Data in a Global other than ^DIZ

When setting up a new file, (Modify File Attributes option), you can instruct VA FileMan to:

- store the new file's data in the default ^DIZ global array, descendent from the file number just assigned

OR:

- store the new file in another global array

The dialog looks like this:

```
MODIFY WHAT FILE: TEST
ARE YOU ADDING 'TEST' AS A NEW FILE? Y <RET>  (YES)
FILE NUMBER: 24000//  <RET>
INTERNAL GLOBAL REFERENCE: ^DIZ(24000,//
```

At this prompt, you either press the Enter/Return key to choose the default or you type an explicit global reference. This reference is in the following format:

```
^GLOBAL( or ^GLOBAL(subscript1,subscript2,…
```

The ^ preceding GLOBAL(need not be entered. Extended global reference ([UCI]) may be entered ahead of the global name. If the subscripted global already exists with data in it, a warning message is displayed.

If the subscripted global is a descendent of a global that stores the data for another file, an error message is displayed. For example, if a file's data is stored at:

```
^GLOBAL(662001,
```

you cannot define another file that stores its data at:

```
^GLOBAL(662001,"A",
```

FIELD GLOBAL STORAGE

Assigning a Location for Fields Stored within a Global

When creating a new Field, (Modify File Attributes option), press the Enter/Return key at the "IS THIS FIELD MULTIPLE" window. If you are a programmer, you are asked in a "pop-up" window for the global subscript and ^-piece position to specify where in each file entry to store the data element being defined. If, for example, you were creating a field that you wanted to be stored in the first ^-piece position of the global subscript DEMOG for every entry, you would enter the following:

```
                              Field #77 in File #100
FIELD LABEL: EMAIL ADDRESS                    DATA TYPE... FREE TEXT

            SUBSCRIPT: DEMOG
AUDIT C   PIECE-POSITION: 1
    REA
  DELETE ACCESS:
   WRITE ACCESS:
         SOURCE:
 DESCRIPTION...        TECHNICAL DESCRIPTION...

                      IS THIS FIELD MULTIPLE... NO

     MANDATORY: NO
   HELP-PROMPT: Answer must be 1-11 characters in length
XECUTABLE HELP:

Enter name of MUMPS Global subscript where this Field's data will be stored.
Already assigned:
        0        1        2        3        4      4.5        5        6
        8        9

                                          Press <PF1>H for help     Insert
```

To aid in the process, VA FileMan prompts you with the highest subscript previously used for the file, and then, when the subscript has been entered, it prompts the ^-piece position one past the highest previously assigned for that subscript. FileMan ensures that no more than 250 characters of data will be stored in any single global node and that no two fields are assigned to the same subscript and ^-piece position.

Note that at the bottom of the screen, a list of the Global subscripts already in use is displayed.

Storing Data by Position within a Node

You may occasionally wish to store a field's data by character position within the global node, rather than by ^-piece position. This is called extract storage instead of ^-piece storage. To accomplish this, after specifying a subscript, respond to the ^-piece prompt with Em,n where m is the first character position for data storage and n is the last. For example, to store data in character positions 1 to 3 of subscript 20, do the following:

```
SUBSCRIPT: 20
^-PIECE POSITION: E1,3
```

One advantage of specifying your field data location using the Em,n format is that ^ can be part of the stored data. It is recommended that you do not mix extract and ^-piece storage on the same global node.

ASSIGNING SUB-DICTIONARY NUMBERS

The "Global File Structure" chapter in this manual points out that data specifications for subfields of a multiple are kept in a subsidiary data dictionary. Such a sub-dictionary is stored in the global ^DD(sub-dictionary_number), where sub-dictionary_number is a number with a fractional portion. For example, the specifications for the RESPONSES multiple of File 100, the ORDER file, are stored in ^DD(100.045). Normally, when a new multiple-valued field is created, VA FileMan automatically assigns the fractional sub-dictionary number. The programmer, however, is allowed to choose the desired number.

When creating a new Multiple Field, (Modify File Attributes option), if you are a programmer, you are asked in a "pop-up" window for the global subscript at which to store the data element being defined. Under this question is the SUB-DICTIONARY question. The RESPONSES multiple in File #100 would have been defined like this:

```
                          Field #4.5 in File #100
FIELD LABEL: RESPONSES                        DATA TYPE... NUMERIC

          TITLE:
          AUDIT:
AUDIT CONDITION:
    READ ACCESS:
  DELETE ACCESS:
    W
                 SUBSCRIPT: 4.5
 DES  SUB-DICTIONARY NUMBER: 100.045

                      IS THIS FIELD MULTIPLE... YES

     MANDATORY: NO
   HELP-PROMPT: Type a number between 1 and 9999999, 0 Decimal Digits
XECUTABLE HELP:

Enter name of MUMPS Global subscript where this Field's data will be stored.
Already assigned:
        0
^DD number must be between 100 and 101 and not already used

                                          Press <PF1>H for help     Insert
```

COMPUTED EXPRESSIONS

A programmer can enter an executable line of M code at any point where one would normally be allowed to use the computed expression syntax. (See the "Computed Expressions" chapter in the *VA FileMan Advanced User Manual*.) The code should create a variable X, which will be understood to be the value of its computation.

NOTE: Because of concatenation, IF, FOR, and QUIT statements are **not** recommended in M computed expressions.

MUMPS DATA TYPE

A data type called MUMPS is available to those with programmer access. The input to this field will be executable M code. Each field of this type is stored on its own global node using the extract format (Em,n).

When an M type field is created, write protection of "@" is automatically given to it. Unless this is modified, only those with programmer access will be able to enter data into an M field.

Programmers are allowed to change the data type of an M field to, for example, free text. However, the values will still be stored in extract format on the subscripted node.

SCREENED POINTERS AND SET OF CODES

A programmer modifying a **pointer** data type field will be asked:

```
SHOULD POINTER ENTRIES BE SCREENED? NO// Y <RET>  (YES)
```

Answering YES allows entry of a line of M code. The variable DIC("S") is set equal to this code. The code is used in the DIC lookup routine to screen out certain entries in the pointed-to file. (See the description of ^DIC call in the "Classic FileMan" chapter in this manual for details about the use of DIC("S"); especially in regard to the naked indicator.) For example, the following trick could be used to make sure that all providers being pointed to from a Surgery file had an S code in some auxiliary field:

```
SCREEN:  S DIC("S")="I $D(^(1)),$P(^(1),U,5)[""S"""
```

Each pointed-to file defined for a variable pointer field can be screened in a similar way.

Also, the programmer can put a screen on a **set of codes** type data field. After the set values have been described, the user is asked:

```
SHOULD SET ENTRIES BE SCREENED?  NO//
```

Again, answering YES allows entry of a line of M code. This code should set the variable DIC("S") which is applied to the selection of the member of the set. When this DIC("S") is executed, the variable, Y, contains the internal value of the member of the set.

INPUT TRANSFORM

An INPUT transform is M code for a particular field that is executed to determine if the data for that field is valid.

The M code for some field types' INPUT transforms is automatically generated when you create the field (this is the case for Free Text, Numeric, Date/time, Computed, MUMPS, and screened Pointer field types).

The Input Transform option of the Utility Functions submenu allows those with programmer access to customize the M code in automatically generated INPUT transforms. It also lets you create input tranforms for other field types. In the Input Transform option, when you select the field, you see an M statement that validates the variable X and kills it if it is invalid. Here, X usually contains the user's response that is being validated. If the field is a variable pointer data type, X contains the value in internally stored format—that is, 'record_number;storage_root'.

You can rewrite this line of code to meet individual requirements. If desired, the code can transform X by resetting it to another value to be filed. An example would be a name transform that deletes an extraneous space character following a comma as shown below:

```
INPUT TRANSFORM: K:$L(X)>30!($L(X)<3) X    Replace K
With S:X[", " X=$P(X,", ")_","_$P(X,",",2) K
Replace <RET>
S:X[", " X=$P(X,", ")_","_$P(X,", ",2) K:$L(X)>30!($L(X)<3) X
```

Unlike the M code for OUTPUT transforms, you **can** use the IF, FOR, and QUIT commands in the M code for INPUT transforms.

Once an INPUT transform has been created for a field, the syntax checking that the field performs can no longer be modified using the Modify File Attributes option. A data dictionary listing will show XXXX for such a field.

For a computed field, the INPUT transform is simply the M code that is executed whenever the field is computed. Hence, a computed field calculation can be edited by a programmer using this option.

INPUT Transforms and the Verify Fields Option

INPUT transforms are ordinarily executed before data is filed (in which case the INPUT transform expects data in **external** form, **not yet filed**). But the INPUT transform is also executed by VA FileMan's Verify Fields option (in which case the data being checked is in **internal** form, and **already filed**). Some parts of your INPUT transform may not be compatible with data in its internal form or when the data is already filed. For example, you may check to make sure a field's value is not stored in a cross-reference before you file it; once you file the entry, however, the field value does exist in the cross-reference and Verify Fields would report the entry as invalid.

To help the Verify Fields option report fewer invalid values in this situation, the Verify Fields option sets the variable DIUTIL to "VERIFY FIELDS" when it is running. You can then check for this variable in your custom INPUT transform and skip any checks that would not be compatible with data that is in its internal form or already filed.

For example:

```
I $G(DIUTIL)'="VERIFY FIELDS"
```

The Verify Fields option does not execute the INPUT transform for the following field types:

- Screened Pointers
- Screened Set of Codes

OUTPUT TRANSFORM

The programmer can write an M OUTPUT transform to convert internal data values to a different external form. Use the variable Y (not X, as used with INPUT transforms).

NOTE: Due to concatenation, do **not** use IF, FOR or QUIT statements when defining OUTPUT transforms. Also, any variables you introduce within an OUTPUT transform (but not Y) should be NEWed.

To reverse the above example, suppose you wanted always to display the name field with a space character following the comma, even though the space is not stored. You could do something like this:

```
OUTPUT TRANSFORM:  S Y=$P(Y,",")_", "_$P(Y,",",2,9)
```

In addition to containing M code setting Y, OUTPUT transforms can consist of a computed expression. For example, if you wanted always to display the month and year from a date/time field called FOLLOW-UP, you could write:

```
OUTPUT TRANSFORM:  MONTH(FOLLOW-UP)
```

SPECIAL LOOKUP PROGRAMS

At times you may need to write a lookup program to respond to unique characteristics of a file. The Edit File option on the Utility Functions submenu allows you to tell VA FileMan what this program is. The information is stored at ^DD(filenumber,0,"DIC"). The routine's name cannot begin with DI. These programs must respond to all the variables that ^DIC does (see the description of ^DIC for additional information). The calls to DO^DIC1, DQ^DICQ, and FILE^DICN may be quite useful to maintain FileMan compatibility. You can tell FileMan to ignore these special programs by including an I in DIC(0).

NOTE: Only the ^DIC call honors the special lookup routines. Those calls that allow the user to specify the indexes (IX^DIC and MIX^DIC1), and the Data Base Server calls (FIND^DIC, $$FIND1^DIC, and UPDATE^DIE) all ignore the Special Lookup Program.

For assistance with special lookups, it is suggested that you contact the VA FileMan developers.

POST-SELECTION ACTION

When it is necessary to examine an entry after it has been selected by DIC, the post-selection action can be invoked. The Edit File option on the Utility Functions submenu allows you to tell VA FileMan what code to execute upon selection. This is stored at ^DD(filenumber,0,"ACT") and can be any standard line of M code. If you decide that the entry should not be selected, the variable Y should be set to -1.

NOTE: The Data Base Server calls (FIND^DIC, $$FIND1^DIC, UPDATE^DIE) all ignore the Post-Selection Action node.

AUDIT CONDITION

You can make a data audit conditional when you define a field as being audited. An audit condition is a line of M code with the characteristics that follow:

- The condition must contain an IF-statement or in some way set $T.
- The audit will take place only if $T=1.
- The variables available to a programmer are:

DA Internal number of the entry being audited. The DA-array will exist if the audit is in a subfile.

DIE The global root of the file or subfile being audited.

DIIX A two-piece variable described below:

- **piece 1** 3 if this audit is taking place during a set; and 2 if this audit is taking place during a kill.
- **piece 2** Field number being edited.

X The internal representation of a field's value, i.e., the actual stored value. X is always present, but its value will vary based on the first piece of DIIX. If $P(DIIX,U,1)=3, then X equals the new value in the field. If $P(DIIX,U,1)=2, then X equals the old value in the field.

If the data type of the field being audited is a Pointer, Variable Pointer, or Set of Codes, then the internal value of the field and its data type will be stored. The old value is stored on node 2.1 of the entry in the Audit file (#1.1) and the new value is stored on node 3.1.

EDITING A CROSS-REFERENCE

A programmer can edit the SET and KILL statements in a MUMPS cross-reference. The logic for other types of cross-reference cannot be edited. After selecting a cross-referenced field in the Cross Reference a Field option on the Utility Functions submenu, choose the Edit option and you will be prompted with the MUMPS cross-reference's current SET and KILL statements for editing. After you have edited the MUMPS cross-reference, you will be given the option of running the **old** kill logic and of cross-referencing existing data (that is, of running the set logic).

For **all** types of cross-references, you can describe the cross-reference in the DESCRIPTION field and enter a free text message in the NO-DELETION field. A message entered in the NO-DELETION field should be a don't-delete-me type of warning since the message entered is displayed under the type of cross-reference prompt presented to someone inquiring about deleting or attempting to delete the cross-reference. For example, PLEASE DON'T DELETE THIS would be a possible message.

The NO-DELETION field must be null before the cross-reference can be deleted.

EXECUTABLE HELP

In addition to placing online help in a field's HELP PROMPT and DESCRIPTION attributes, you can enter EXECUTABLE HELP if you have programmer access. When defining a field's attributes using the Modify File Attributes option, you will receive the "XECUTABLE 'HELP':" prompt. Here you can enter M code that will be executed when the user requests help while editing data in the field. If the user enters one question mark, the code is executed after the help prompt is displayed. With two question marks, it is executed before the field's description is displayed.

13. Trigger Cross-references

INTRODUCTION

A trigger causes something else to happen. In VA FileMan, you can set up a trigger so that the entry of data in one field automatically updates a second field value. Since a trigger is considered a type of cross-reference on the field for which data is entered, a trigger is logically created under the Cross-Reference a Field or File option in the Utility Functions menu.

To understand how a trigger is set up, you must first understand that every cross-reference specification describes both:

- what happens when a new value is entered, either initially or when an existing value is changed (set logic)
- what happens when an old value is changed or deleted (kill logic)

In other words, when patient SMITH,JACK is first entered into a file of patients, a SMITH,JACK regular cross-reference on the name is built (and nothing is deleted). Then, when this name is edited (changed) to be SMITH,JOHN Q—two things happen:

- the SMITH,JACK regular cross-reference is deleted
- a SMITH,JOHN Q regular cross-reference is created

Finally, when this patient is deleted from the file, the SMITH,JOHN Q cross-reference is deleted (and none is created).

When you are using the Cross-Reference a Field or File option and you specify a trigger, you must identify both what happens when a new field value is entered (either initially or through an edit on an existing value) and when an old value is changed or deleted.

You must be careful in setting up any trigger cross-reference since unexpected effects can sometimes result. Note that at the moment when the trigger actually occurs:

- No validity check is made on the value being forced into the field (in other words, the value doesn't go through the triggered field's INPUT transform).
- Cross-references (if any) do occur on the triggered field (e.g., a triggered field can in turn trigger other fields in a chain reaction!).

A TRIGGER ON THE SAME FILE

Adding a time and date stamp to the file whenever a particular field is updated is a simple example of a trigger. Suppose the Patient file has a date-valued field called DATE NAME CHANGED. Here is how you could put the current date and time into this field whenever the patient's NAME is entered or changed:

```
Select OPTION:  UTILITY FUNCTIONS
Select UTILITY OPTION:  CROSS-REFERENCE A FIELD

MODIFY WHAT FILE:  PATIENT <RET>              (1890 entries)
Select FIELD:  NAME

CURRENT CROSS-REFERENCE IS REGULAR 'B' INDEX OF FILE

Choose E (Edit)/D (Delete)/C (Create):  CREATE
WANT TO CREATE A NEW CROSS-REFERENCE FOR THIS FIELD?
NO// Y <RET>  (YES) CROSS-REFERENCE NUMBER:  2// <RET>
Select TYPE OF INDEXING: REGULAR// TRIGGER

WHEN THE NAME field (#.01) of the PATIENT File (#2)
IS CHANGED, WHAT FIELD SHOULD BE 'TRIGGERED':  DATE NAME CHANGED <RET>  ..OK
```

The field to be triggered must already exist.

```
---- SET LOGIC ----

IN ANSWERING THE FOLLOWING QUESTION, 'DATE NAME CHANGED'
   CAN BE USED TO REFER TO THE EXISTING TRIGGERED FIELD VALUE.
PLEASE ENTER AN EXPRESSION WHICH WILL BECOME THE VALUE OF THE DATE
NAME CHANGED field (#2) OF THE 'PATIENT' File (#2)
 WHENEVER 'NAME' FIELD IS ENTERED OR CHANGED:  NOW

DO YOU WANT TO MAKE THE SETTING OF 'DATE NAME CHANGED' CONDITIONAL?
  NO// <RET>  (NO)
```

If you answered YES, you can set conditions for the trigger. You will get the prompt:

```
ENTER AN EXPRESSION FOR THE CONDITION:  <RET>

--- KILL LOGIC ---

IN ANSWERING THE FOLLOWING QUESTION, 'DATE NAME CHANGED'
 CAN BE USED TO REFER TO THE EXISTING TRIGGERED
 FIELD VALUE.  NOTE: 'OLD NAME' CAN BE USED TO REFER TO THE VALUE
 OF THE NAME FIELD BEFORE ITS CHANGE OR DELETION.
PLEASE ENTER AN EXPRESSION WHICH WILL BECOME THE VALUE OF
THE 'DATE NAME CHANGED' field (#2) OF THE 'PATIENT' File (#2)
  WHENEVER 'NAME' IS CHANGED OR DELETED: <RET> NO EFFECT
```

You have specified that the NAME field will trigger the DATE NAME CHANGED field (noting that the NAME field is already cross-referenced in the usual way). You have requested that the current date/time (NOW) be stuffed into the triggered field.

Since this triggering will occur whenever NAME is changed, you don't have to specify anything else that depends on the pre-existing value of NAME. When the entire patient entry is deleted, the DATE NAME CHANGED will be deleted along with the name. Thus, no KILL LOGIC is needed. The response to pressing the Enter/Return key at that prompt is "NO EFFECT".

Since you always want the trigger to take place when NAME is changed, no condition is placed on the trigger. A trigger can be setup that will only occur under specified circumstances.

```
WANT TO PROTECT THE 'DATE NAME CHANGED' FIELD, SO THAT
IT CAN'T BE CHANGED BY THE 'ENTER & EDIT' ROUTINE? NO//  YES
```

You specify that the only way you want the DATE NAME CHANGED field to be updated is via this trigger. No Enter or Edit File Entries option user (not even one with an @-sign) will be able to change a patient's DATE NAME CHANGED field directly.

```
NO-DELETION MESSAGE: <RET>
```

If you enter a free text message at this prompt, this cross-reference cannot be deleted.

```
DESCRIPTION:
1>The DATE NAME CHANGED field will be triggered whenever the
2>NAME field is entered or updated.  The triggered value will be
3>NOW.  This field cannot be edited.
4><RET>
```

The description will appear in a standard DD listing.

```
...CROSS-REFERENCE IS SET

DO YOU WANT TO RUN THE CROSS-REFERENCE FOR EXISTING
ENTRIES NOW? NO//  <RET>
```

Finally, you have the option of using the new trigger to update the file. In this case, it would not be useful to put the current date and time into the DATE NAME CHANGED field for every existing entry. Thus, the NO default is accepted.

TRIGGERS FOR DIFFERENT FILES

A trigger can also update a field in a file different than the one in which the edited field exists. To illustrate this, the previous example is extended to show how a separate Monitor file could be updated whenever a patient name is added or changed.

First of all, define this Monitor file using the Modify File Attributes option. The Monitor file's NAME field will contain the same value as the NAME field in the Patient file. A TIME field should be defined as a date/time data type; this field will contain the time the NAME field in the Patient file was added or changed. Use the Trigger option on the Utility Functions submenu to set up the trigger:

```
Select UTILITY OPTION:  CROSS-REFERENCE A FIELD

MODIFY WHAT FILE:  PATIENT
Select FIELD:  NAME

CURRENT CROSS-REFERENCES:
       1    REGULAR 'B' INDEX OF FILE
       2    TRIGGER OF THE 'DATE NAME CHANGED' FIELD OF THE PATIENT
                    FILE
Choose E (Edit)/D (Delete)/C (Create):   CREATE
WANT TO CREATE A NEW CROSS-REFERENCE FOR THIS FIELD? NO//  YES <RET> CROSS-
REFERENCE NUMBER: 3// <RET>
Select TYPE OF INDEXING: REGULAR//  TRIGGER

WHEN THE 'NAME' field (#.01) OF THE 'PATIENT' File (#2) IS CHANGED,
WHAT FIELD SHOULD BE 'TRIGGERED:   NAME:MONITOR:TIME
DO YOU WANT TO PERMIT ADDING A NEW 'MONITOR' ENTRY? NO//  Y <RET>   (YES)
WELL THEN, DO YOU WANT TO **FORCE** ADDING A NEW ENTRY
      EVERY TIME? NO//  Y <RET>   (YES)
  ...OK

--- SET LOGIC ---

IN ANSWERING THE FOLLOWING QUESTION, 'TIME'
 CAN BE USED TO REFER TO THE EXISTING TRIGGERED FIELD VALUE.
PLEASE ENTER AN EXPRESSION WHICH WILL BECOME THE
 VALUE OF THE 'TIME' field (#1 ) OF THE 'MONITOR' File (#16001)
 WHENEVER 'NAME' IS ENTERED OR CHANGED:  NOW

DO YOU WANT TO MAKE THE SETTING OF 'TIME' CONDITIONAL? NO// <RET> (NO)

--- KILL LOGIC ---

IN ANSWERING THE FOLLOWING QUESTION, 'TIME'
  CAN BE USED TO REFER TO THE EXISTING TRIGGERED FIELD VALUE.
  NOTE: 'OLD NAME' CAN BE USED TO REFER TO THE VALUE OF
  THE NAME FIELD BEFORE ITS CHANGE OR DELETION.
```

```
PLEASE ENTER AN EXPRESSION WHICH WILL BECOME THE VALUE OF
  THE 'TIME' field (#1) OF THE 'MONITOR' File (#16001)
  WHENEVER 'NAME' IS CHANGED OR DELETED:  @
ARE YOU SURE YOU WANT TO 'ADD A NEW ENTRY' WHEN THIS
  KILL LOGIC OCCURS? NO//  Y <RET>  (YES)

DO YOU WANT TO MAKE THE DELETING OF 'TIME' CONDITIONAL? NO// <RET> (NO)

WANT TO PROTECT THE 'TIME' FIELD, SO THAT
IT CAN'T BE CHANGED BY THE 'ENTER & EDIT' ROUTINE? NO// <RET> (NO)
NO-DELETION MESSAGE:  <RET>
DESCRIPTION:

1>The TIME field of the Monitor file will be triggered whenever
2>the NAME field of the Patient file is entered or changed.  The
3>new value=NOW.  A new entry in the Monitor file will be created
4>at the same time.  If the NAME field in the Patient file is
5>deleted, TIME will be deleted.
6><RET>

...CROSS-REFERENCE IS SET
DO YOU WANT TO RUN THE CROSS-REFERENCE FOR EXISTING ENTRIES NOW?
   NO//  <RET> (NO)
```

This example shows the **extended pointer syntax** used to specify a field in another file. The patient's NAME is used as a lookup value in the Monitor file. A new Monitor entry is created by the trigger. In a sense, this trigger really updates two fields in Monitor file, NAME and TIME.

An alternative extended pointer syntax is NAME IN MONITOR FILE:TIME. This syntax is exactly equivalent to NAME:MONITOR:TIME, and may better express the meaning of the extended syntax if you are a new user.

NOTE: @ indicates that a field is to be deleted by the trigger.

14. DIALOG File

DIALOG FILE: USER MESSAGES

Introduction

The VA FileMan DIALOG file is used to store dialog that would normally appear on a screen during interaction with a user. This dialog may include error messages, user help, and other types of prompts. FileMan distributes a set of entries in the DIALOG file.

The VA FileMan calls, BLD^DIALOG or $$EZBLD^DIALOG, are used to move text from the DIALOG file into arrays. The text can then be displayed using the display mode of choice.

Developers may add entries to the DIALOG file. Entries such as error messages, help messages and other general prompts can be placed in the file. The DIALOG file should not be used for storing alternate synonyms either for data or for fields in the data dictionary such as field labels or descriptions.

Note: If you wish to add entries to the DIALOG file, you must apply to the DataBase Administrator for a numberspace.

Advantages of the DIALOG file for user interaction are:

- User interaction can be easily separated from the other program functionality, a necessary step in creating alternate interfaces to roll-and-scroll, such as GUI.
- Text stored in the DIALOG File can be re-used.
- Package error lists can be identified and listed by error number in documentation.
- Text can be returned in multiple languages without changes to developers' code. (See "Internationalization" section of the "DIALOG File" chapter in this manual.

Use of the DIALOG File

VA FileMan controls and distributes entries in the DIALOG file in the number range 0 through 10000. These entries should not be edited by other package developers, with the exception of adding foreign language equivalents for text (see the "Internationalization and the Dialog File" section of the "DIALOG File" chapter in this manual for details). Some of the FileMan error messages are available for retrieval by other package developers, using the FileMan program calls. These messages are listed in the "Error Codes" appendix in this manual. Entries within

the FileMan number range that are not in the Error Codes listing should not be used as they are subject to change.

Other packages may make entries in the DIALOG file for their own use. The VHA Database Administrator will assign number ranges to a subscribing package.

If your package or site already has a file numberspace assigned by the DBA, you can use that number (or numbers) multiplied by 10000 (plus any decimal value between .001 and .999) for adding entries to the DIALOG file (e.g., Kernel owns file 200, so they can use numbers 2000000 through 2000000.999. If I'm site 665, I own numberspace 665000 for files, so I can use 6,650,000,000 through 6,650,000,000.999 in the DIALOG file).

If developers do not follow these guidelines, their DIALOG entries may be overwritten when new packages are installed. Note that an entry number does not have to be an integer--up to 3 decimal places can be used to identify an entry.9

Creating DIALOG File Entries

Developers may enter or edit entries to the DIALOG file using VA FileMan Enter/Edit. The only required fields are the DIALOG NUMBER which uniquely identifies the entry, the TYPE (Error, Help or General Message), and the dialog TEXT.

Dialog text can contain parameter windows delimited by vertical bars. Within a pair of vertical bars, the developer puts a value that will correspond to a subscript in a parameter list. This subscript need not be numeric, but may be meaningful alpha characters such as "FIELD". When the dialog text with windows is retrieved using a call to either BLD^DIALOG or $$EZBLD^DIALOG, a subscripted parameter list is input to the call. The parameters are matched by subscript to the windows in the text, and the values from the parameter list are inserted into the corresponding windows in the text. If parameters are included in the text, the INTERNAL PARAMETERS NEEDED field should be set to YES. A multiple field called PARAMETER is used in documenting these parameters.

For error messages only, a list of output parameters can also be passed to BLD^DIALOG or $$EZBLD^DIALOG. This list is returned by the routine in a standard format. Output parameters might be, for example, file or field numbers which the calling routine may then use to make a decision. Output parameters should also be documented in the PARAMETER multiple described immediately above.

Another important optional field is the POST-MESSAGE ACTION field. If the developer wishes to perform some special action whenever a message is retrieved, M

code is simply inserted into this field. The code will then be executed whenever the associated message is retrieved with a call to BLD^DIALOG or $$EZBLD^DIALOG.

The TRANSLATION (LANGUAGE) multiple in the DIALOG file allows a developer to enter text in a language other than English. See the "Internationalization and the DIALOG File" section of the "DIALOG File" chapter in this manual for additional information on this feature.

Finally, there is a place to enter documentation for the ROUTINE names and LINE TAGs which use the dialog entries. This is optional internal documentation for the use by developers only.

Following is an example creating a new entry in the DIALOG file.

```
    Select DIALOG: 10001
      Are you adding '10001' as a new DIALOG (the 239TH)? Y <RET>  (YES)
    TYPE: ?
         Enter code that reflects how this dialogue is used when talking to
the users.
         Choose from:
           1        ERROR
           2        GENERAL MESSAGE
           3        HELP
    TYPE: 3 <RET>  HELP
    PACKAGE: VA FILEMAN <RET>          DI
    DESCRIPTION:
      1>Here we enter a description of the help message itself.  This
      2>description is for our own documentation.
      3>
    EDIT Option: <RET>
    INTERNAL PARAMETERS NEEDED: Y <RET>  YES
    TEXT:
      1>Here we enter the actual text of the help messages, with
      2>parameters designated by vertical bars |1| as shown.
      3>
    EDIT Option: <RET>
    Select PARAMETER SUBSCRIPT: 1
      Are you adding '1' as a new PARAMETER SUBSCRIPT (the 1ST for this
DIALOG)? Y <RET> (YES)
      PARAMETER DESCRIPTION: Brief description of parameter 1 goes here. For
      documentation only.
    Select PARAMETER SUBSCRIPT: <RET>
    POST MESSAGE ACTION: ? <RET>        This is Standard MUMPS code.  This
code will be
    executed whenever this message is retrieved through a call to BLD^DIALOG
    or $$EZBLD^DIALOG
    POST MESSAGE ACTION: S MYVAR="HELP #10001 WAS REQUESTED"
    Select LANGUAGE: <RET>
    Select ROUTINE NAME: DIKZ// <RET>
      ROUTINE NAME: DIKZ// <RET>
      LINE TAG:  // <RET>
```

INTERNATIONALIZATION AND THE DIALOG FILE

Role of the VA FileMan DIALOG File in Internationalization

The VA FileMan DIALOG file is used to store dialog that would normally appear on a screen during interaction with a user. The DIALOG file becomes especially important in assisting developer support for non-English speaking users because it allows easy entry and retrieval of non-English dialog without making any changes to code that is already using the DIALOG file.

Use of the DIALOG File in Internationalization

A system variable, DUZ(LANG), identifies to VA FileMan the language currently in use. This system variable is set equal to a number that corresponds to the ID NUMBER of an entry in the LANGUAGE file (see discussion of the VA FileMan LANGUAGE file). This number is also used as a subscript for the TRANSLATION (LANGUAGE) multiple in which non-English text can be stored. For users running Kernel V. 8.0 or later, this variable is set automatically during signon.

For every entry needing translation in the DIALOG file, the developer should populate the FOREIGN TEXT field for the desired language. When either of the text retrieval routines, BLD^DIALOG or $$EZBLD^DIALOG, is called, if DUZ("LANG") is greater than one (1), FileMan will look at the language location specified by DUZ("LANG") to find the text. If text cannot be found at that location, FileMan defaults to use the English equivalent from the TEXT field. As with English text, parameters to be inserted into the text can be passed to the call.

See also the programmer calls BLD^DIALOG and $$EZBLD^DIALOG.

Creating Non-English Text in the DIALOG File

Once an entry exists in the DIALOG file, developers may enter or edit non-English equivalents for the TEXT field, using FileMan Enter/Edit.

Example

```
Select DIALOG: 10001 <RET>  This is English text for a test message.
.
.
.

Select LANGUAGE: ?
 Answer with TRANSLATION LANGUAGE
     You may enter a new TRANSLATION, if you wish
     Enter the number or name for a non-English language.
     English language cannot be selected.
 Answer with LANGUAGE ID NUMBER, or NAME
Choose from:
   2      GERMAN
   3      SPANISH
   4      FRENCH
   5      FINNISH
   6      ITALIAN
   10     ARABIC
   11     RUSSIAN

Select LANGUAGE: 2 <RET>       GERMAN
  Are you adding '2' as a new TRANSLATION (the 1ST for this DIALOG)? Y <RET>
(Yes)
  FOREIGN TEXT:
 1>Here is where we enter the non-English text.
```

VA FILEMAN LANGUAGE FILE

Introduction

Certain types of data such as dates and numbers, should be formatted differently for display depending on the language of the end user. The VA FileMan LANGUAGE file has been designed to help solve this problem for users of interactive VA FileMan. The LANGUAGE file stores M code used to perform language-specific conversions on such data. A system variable identifies to FileMan the language currently in use.

At this time, VA FileMan distributes in the LANGUAGE file only the English equivalent of language-specific data conversions specified below.

Use of the LANGUAGE File

A system variable, DUZ("LANG"), identifies to VA FileMan the language currently in use. This system variable is set equal to a number that corresponds to the ID NUMBER of an entry in the LANGUAGE file. It tells VA FileMan where to find the appropriate data conversion code from the LANGUAGE file at the time the code needs to be executed (for example, when printing a date). For users running Kernel V. 8.0 or later, this variable is set automatically during signon.

Developers may enter or create their own entries in the LANGUAGE file. The VHA Database Administrator will assign an ID NUMBER for each unique language entry in the LANGUAGE file. If developers do not follow these guidelines, their language entry may be overwritten when VA FileMan is installed.

The following Language file entries have been assigned and are distributed with VA FileMan:

1	English
2	German
3	Spanish
4	French
5	Finnish

6	Italian
10	Arabic
11	Russian

Creating LANGUAGE File Entries

Developers may enter or edit entries in the LANGUAGE file using VA FileMan Enter/Edit. The only required fields are the ID NUMBER that uniquely identifies a language and the language NAME. If M code is not found within the current language for a specific conversion, VA FileMan will default to use the English equivalent.

The other fields that can be entered for any LANGUAGE file entry are described below. At the time the code in any of these fields is executed, the data to be converted will be in the local variable Y. The M code in the field should put the transformed output back into Y, without altering any other local variables. More detail can be found in the field description for each field. Looking at the English equivalent in entry number 1 may also be helpful.

ORDINAL NUMBER FORMAT	Changes 1 to 1ST, 2 to 2ND, etc.
CARDINAL NUMBER FORMAT	Changes 1234567 to 1,234,567
UPPERCASE CONVERSION	Converts text to uppercase
LOWERCASE CONVERSION	Converts text to lowercase
DATE/TIME FORMAT	Converts date in internal VA FileMan format to MMM,DD,YYYY@HH:MM:SS
DATE/TIME FORMAT	Does other date conversions from date in internal VA FileMan format. This call has an additional input flag that

(FMTE) indicates the conversion to be done.

The flags are:

1	MMM DD, YYYY@HH:MM:SS	Space before year
2	MM/DD/YY@HH:MM:SS	No leading zeros on month,day
3	DD/MM/YY@HH:MM:SS	No leading zeros on month,day
4	YY/MM/DD@HH:MM:SS	--
5	MMM DD,YYYY@HH:MM:SS	No space before year
6	MM-DD-YYYY @ HH:MM:SS	Special spacing for time
7	MM-DD-YYYY@HH:MM:SS	--
S	Always return seconds	
U	Return uppercase month (use only with 1 or 5)	
P	Return time with am,pm	
D	Return only date without time	

15. VA FileMan Functions (Creating)

INTRODUCTION

As mentioned in the "VA FileMan Functions" chapter of the *VA FileMan Advanced User Manual*, as a programmer in FileMan you can create your own computed-expression functions. In some ways, a function can be thought of as an OUTPUT transform that can work on any field. For example, you may have a preference for seeing many dates displayed as 20-7-69, rather than the JUL 20,1969 format that FileMan typically produces. Since this date is internally stored in the form 2690720 (see the description of %DT), you could write a line of code that took the internally stored format in the variable X and transformed it using:

```
+$E(X,6,7)_"-"_+$E(X,4,5)_"-"_$E(X,2,3)
```

FUNCTION FILE ENTRIES

This is exactly what you are allowed to do when you edit the FUNCTION file (#.5) using the Enter or Edit File Entries option.

To continue the above example, you could create a DASHDATE function which could then be used by any user to display date-valued fields and expressions in the DAY-MONTH-YEAR format as follows:

```
    Select OPTION: ENTER AND EDIT FILES
    INPUT TO WHAT FILE: FUNCTION
    EDIT WHICH ATTRIBUTE: ALL// <RET>
    Select COMPUTED-FIELD FUNCTION: DASHDATE
      ARE YOU ADDING 'DASHDATE' AS A NEW COMPUTED-FIELD FUNCTION? Y <RET>
(YES)
    MUMPS CODE: S X=+$E(X,6,7)_"-"_+$E(X,4,5)_"-"_$E(X,2,3)
    EXPLANATION: PRINTS DATE IN "DD-MM-YY" FORMAT
    DATE-VALUED: NO
    NUMBER OF ARGUMENTS: 1
    WORD-PROCESSING: <RET>
```

Notice that the MUMPS CODE field contains code to transform the variable X (the argument of the function) into a different X. If two arguments were required for the function, the first would be found in the variable X1 and the second in X. Although the new function being created here takes a date-valued argument, it is not itself considered to be date-valued since it doesn't produce values that look like the standard FileMan internal representation of a date. If this function was only

meaningful in a word processing context, you would put a W at the "WORD-PROCESSING:" prompt.

NOTE: If there is an output transform on a field, the function code is applied to the field after it has been transformed. In most cases, if a field has an output transform, you should therefore use the syntax FUNCTION_NAME(INTERNAL(FIELD_NAME)), rather than FUNCTION_NAME(FIELD_NAME).

A function can also be defined as taking no arguments. This is very similar to the special variables in M like $I and $H. For example, you could define a function like BELL as follows:

```
    Select COMPUTED-FIELD FUNCTION: BELL
       ARE YOU ADDING A NEW COMPUTED-FIELD FUNCTION? Y <RET>  (YES)
    MUMPS CODE: SET X=$C(7) <RET>    EXPLANATION: CAUSES A 'BEEP' TO OCCUR
ON OUTPUT <RET>   DATE-VALUED: NO
    NUMBER OF ARGUMENTS: 0
    WORD-PROCESSING: <RET>
```

Users could then embed "beeps" in output templates by entering:

```
    FIRST PRINT FIELD: BELL
```

NOTE: No parentheses are shown for a function with no arguments.

You can delete a function in the usual way by deleting the NAME of the function. Such deletions do not harm any computed fields that already have been created using the function. However, you may not edit the computed field unless you remove reference to the deleted function.

WARNING: Due to concatenation, do not use IF, FOR or QUIT statements when defining functions. Also, any variables you introduce within a function's code (but not X, X1, etc.) should be NEWed.

The Function file already contains several functions. Consult the "VA FileMan Functions" chapter of the *VA FileMan Advanced User Manual* for a description of the functions exported with VA FileMan.

16. DIFROM

Introduction

DIFROM is the mechanism that was used **in the past** to transfer software packages from one VA FileMan environment to another.

NOTE: DIFROM has been superceded by KIDS (Kernel Installation and Distribution System) for this function, starting with Kernel V. 8.0. DIFROM can still be used, for the time being, for the purpose of package export between FileMan systems where Kernel has not been installed.

However, the VA FileMan developers are no longer enhancing DIFROM, so **in VA FileMan V. 22, any new-style indexes or keys that are added to a file will NOT be transported by DIFROM.**

Package transfer is a two-stage process. First, DIFROM is run on the source system. It is a nondestructive process that uses the ^UTILITY global to build data structures and store information about the package. Then, DIFROM creates init routines. Later, on the target system, the init routines are run to recreate each component of the package and put them into place according to the installer's instructions.

Another component of the package export process is the Package file. A PACKAGE file entry contains information about the components of a package. It also indicates how the installation will proceed on the target system. The PACKAGE file also has fields that document the package production and installation process. PACKAGE file entries can be created using the standard VA FileMan editing option.

EXPORTING DATA

Preparing To Run DIFROM

DIFROM simply creates routines, "init routines." DIFROM names routines by appending INI* or I### to the package namespace (for example, nmspI005 or nmspINI1). It will overwrite any like-named routines. Except for replacing routines with the same name, DIFROM is nondestructive and, unlike the init installation process, neither changes nor destroys data. The DIFROM user should ensure that there is sufficient disk space to hold the init routines DIFROM creates.

CAUTION: Remember that, beginning with Kernel V. 8.0, DIFROM has been superceded by KIDS (Kernel Installation and Distribution System) for the function of transferring software packages from one VA FileMan environment to another. DIFROM can still be used for the time being for the purpose of package export between FileMan systems where Kernel has not been installed. VA FileMan developers are no longer enhancing DIFROM and, in FileMan V. 22.0, any new-style indexes and keys added to a file will NOT be transported by DIFROM.

PACKAGE File and DIFROM

The PACKAGE file (#9.4) is used both to document a software package and to aid in exporting the package. A PACKAGE file entry is not required to build inits; inits can be built on the fly. Some of the fields are used for documentation only and some for both the export process and documentation. Whenever you build an init using an entry in the PACKAGE file, that entry is also put into the PACKAGE file of the target system when the init is run. Thus, a copy of the documentation for the package will be on both the source and target systems.

The fields that DIFROM uses during the package export process are described below. All fields not noted below are used for documentation only:

1. NAME
2. PREFIX
3. Template Multiples
4. EXCLUDED NAME SPACE
5. ENVIRONMENT CHECK ROUTINE
6. PRE-INIT AFTER USER COMMIT
7. POST-INITIALIZATION ROUTINE
8. FILE

9. Other PACKAGE File Fields

NAME

This is a brief (4-30 characters) name describing the package. It is used to identify the package and does not affect the init directly. However, it is the key field used when installing the PACKAGE file entry on the target system. If you change the name and install a package on a system where it already exists under a different name, a new entry will be created in the PACKAGE file on the target system. The unchanged old entry will remain, too.

PREFIX

This is the 2-4 character namespace of the package. It is the unique identifier for the package. The PREFIX controls which Templates, Options, Bulletins, etc., are included in the init routines for export. Those components with names beginning with the package's PREFIX are automatically exported, except for those beginning with the letters in the EXCLUDED NAME SPACE multiple.

Template Multiples

There is a multiple field for each of the following template types: INPUT, SORT, PRINT, and SCREEN (FORM). The developer uses these multiples to have the init include templates in addition to those within the PREFIX namespace. Each of these multiples contains the free-text name of a template and the number of the file associated with that template (a pointer to the FILE of Files).

Note that for SCREEN (FORM) templates, all blocks pointed-to by exported forms are automatically included in the init regardless of their namespace. The blocks need not be specified by the developer.

EXCLUDED NAME SPACE

The developer can use the EXCLUDED NAME SPACE multiple to exclude templates, options, bulletins, etc., that are a subset of the package's namespace. For example, if the namespace of a package were PRC and the EXCLUDED NAME SPACE multiple contained the entry PRCZ, then any of the components of the package with names beginning with "PRCZ" would not be exported.

ENVIRONMENT CHECK ROUTINE

When the installer starts the init, the routine identified in the ENVIRONMENT CHECK ROUTINE field is run before any of the init routines DIFROM created and before any questions are asked. The installer cannot interrupt the init process until this routine has completed. Thus, this pre-init should be used to simply examine the environment; it should not change any data.

PRE-INIT AFTER USER COMMIT

The routine named in the PRE-INIT AFTER USER COMMIT field runs after the installer has committed to proceeding with the install but before any data is updated. This routine may edit or delete data. The developer uses this routine to make any data conversions, etc., that need to be performed before the init brings in new data.

POST-INITIALIZATION ROUTINE

The routine named in the POST-INITIALIZATION ROUTINE field runs after the inits put everything in place. Here, the developer makes any data conversions, etc., that need to be performed after the new data is installed.

FILE

This is a multiple field used to describe how the data dictionaries (DDs) and data from the exported files are to be handled in the inits. The following fields are included within the FILE multiple:

- **FILE**

  This field contains the number of the file to be exported. It is a pointer to the FILE of Files (#1).

- **FIELD**

  This optional multiple within the FILE multiple allows the developer to send a subset of the fields from a file. If only some of the fields are being exported, a "partial" file is being sent. If no entries are made in the FIELD multiple, all of the fields from the file are exported. Only the names of fields at the top level of a file can be entered. Thus, single fields at the top level and entire multiples with all the subfields and subfiles descendent from those multiples can be sent.

The .01 field will automatically be sent, whether or not it appears in this multiple, unless the file being exported is File #200. If a partial of File #200 is being sent, the .01 is sent only if it is included in this multiple or if the PREFIX is XU (Kernel's namespace).

NOTE: This list only applies to the information about fields found in the data dictionary. It is not possible at this time to send a subset of the actual data.

- **ASSIGN A VERSION NUMBER**

  If this set of codes field is YES, the version number entered by the developer while running DIFROM to build the init will be used to create the following node when the init is run on the target system:

  ```
  ^DD(File#,0,"VR")=Version Number
  ```

  The version number is that of the package being installed, not the VA FileMan version number.

  If this field is NO or left null, a "VR" node will not be built by the init. Thus, whatever was present in this node on the target system will remain. Once a "VR" node has been set, the developer should continue to update it with each version. Otherwise, the node will contain the wrong version.

- **UPDATE THE DATA DICTIONARY**

  This set of codes field controls whether or not a pre-existing DD on the target system will be updated during the init. The DD is included in the init routines regardless of how this question is answered. If a DD for the file does not exist on the target system, it is always installed.

  If this field is YES or left null, the DD in the init will overlay an existing DD on the target system.

  NOTE: The existing DD on the target system is not deleted first. For example, if a field is changed from one type to another, it is possible that the DD information from the previous definition of the field will be left behind. This situation may cause problems for FileMan. If this might happen, the developer is urged to clean up the field from the DD in a pre-init, using a call to ^DIK.

  If this field is NO, the DD currently on the target system will not be changed. The developer can still send data for the file.

- **MAY USER OVERRIDE DD UPDATE**

  If this set of codes field is YES, the installer decides if a pre-existing DD will be overwritten. When the init routine runs, the question, "Shall I write over the existing Data Definition?" is asked if there is a pre-existing DD on the target system. If the installer answers this question NO, the existing DD will not be changed. This feature is useful when a package contains some DDs that are unchanged from the previous version. If the DD is not found on the target system, it will be brought in by the init regardless of this field's contents.

  If answered NO or left null, the installer cannot choose whether or not to not bring in the DD.

  NOTE: If there is a screen on the DD, the question is not asked regardless of the contents of this field. The result of the screen's test determines if the new DD is installed or not.

- **SCREEN TO DETERMINE DD UPDATE**

  The developer can enter M code in this field to examine the target environment to determine whether or not to bring in a DD. The code should set the value of $T. If $T is true, the new DD is installed; if $T=0, it is not. If the developer enters a screen, the installer is not given the option of installing the DD. The screen alone determines whether or not the DD is installed.

  NOTE: If the DD does not exist on the target system, the screen is ignored and the incoming DD is installed.

- **DATA COMES WITH FILE**

  If this set of codes field is YES, DIFROM picks up ALL of the data for the file from the system on which the developer builds the init. This data is included in the init routines. Data from all fields is sent even if the developer is only sending selected fields from the DD. Pointers are not resolved to their external values. Thus, data with pointers should not be sent if the pointed-to entries may be in different locations on the target system.

  If this field is NO or left null, the init does not pick up data. The contents of the following two fields are ignored.

- **MERGE OR OVERWRITE SITE'S DATA**

  This set of codes field controls how exported entries are combined with existing ones on the target site. The possible values are "m" (MERGE) and "o" (OVERWRITE). The default is MERGE.

  When an init is installed, incoming entries and subentries are checked to see if they match ones on the target system. (A detailed description of this process is given in the Running an Init section below.) If a match is not found, the entry or subentry is added. The contents of this field determine what happens to entries that do match.

  If incoming entries are to be merged with existing ones, fields with non-null values are left unchanged on the target system. Data from the init is placed into fields with null values.

  If incoming entries are to overwrite existing ones, fields with non-null values in the init overwrite values currently on the target system. If the field is null in the init and the field on the target system contains data, the current value is not overwritten with a null value.

- **MAY USER OVERRIDE DATA UPDATE**

  If this set of codes field is YES, the installer can decide whether or not to install the data from the init. The installer can choose to bring in the data or not to bring it in. However, the merge/overwrite flag cannot be changed; merge cannot be switched to overwrite, and vice versa.

  If the field is NO or left null, the installer cannot choose if the target system's data is updated or not.

Other PACKAGE File Fields

Other PACKAGE file fields are used only for documentation and do not affect the DIFROM procedure. One of the documentation fields, SHORT DESCRIPTION, is required. It is a free text field of up to 60 characters. Other documentation fields include: ROUTINE, GLOBAL, VERSION, DEVELOPMENT ISC, and KEY VARIABLE. There are fields to document the development, verification, site installation, and patch history. This data describing the package is bundled and exported with the rest of the package. It is put into the recipient's PACKAGE file.

Some of the documentation fields are updated on the target system when the init is run. For example, the date/time that the pre- and post-inits are run is automatically recorded in the PACKAGE file entry as is the version number.

ORDER ENTRY AND DIFROM

DIFROM for VA FileMan versions 18 and later has been customized to support Order Entry. Order Entry inits must export records from the PROTOCOL file (#101). This file contains pointers back to itself, similar to the OPTION file (#19). Since DIFROM does not currently resolve these pointers, a joint effort was made by Order Entry and FileMan developers to support Order Entry inits that correctly install the protocols. (See Order Entry documentation for details.)

Basically, the process involves the creation of a second set of routines, similar to init routines, to export the Order Entry protocols and to resolve the pointers in the PROTOCOL file. An Order Entry routine, ORVOM, is run to create these routines. The resulting routines are named nmspONI*, the ONIT routines. These routine are run at the target site after the init routines to install the protocols.

The following considerations pertain to the creation of Order Entry inits:

- Like regular inits, Order Entry inits can be created either based on an entry in the PACKAGE file (#9.4) or on the fly.
- Order Entry's file Order Parameters contains a multiple called PACKAGE PARAMETERS. This multiple controls the export and installation of entries from the PROTOCOL file. Thus, it is used like the PACKAGE file. To export entries from the PROTOCOL file in a sophisticated way, use the PACKAGE PARAMETERS instead of building the export on the fly.

  If the developer is going to use the PACKAGE PARAMETERS, there must be a PACKAGE file entry. Then, create an entry in the PACKAGE PARAMETERS multiple within the ORDER PARAMETERS file (#100.99). The .01 field of this multiple is a DINUMed pointer to the PACKAGE file entry. This implies that the namespace must be the same as that used for the init.
- Whether the Order Entry init is built from the PACKAGE PARAMETERS or on the fly, next run the ORVOM routine. The ORVOM routines look at the PACKAGE PARAMETERS (if they exist) or prompt the developer for the names of protocols to be sent. They build ONIT routines that are similar to inits but contain only PROTOCOL file entries. The code generated in these routines installs and resolves pointers on the PROTOCOL file entries. (The ORVOM routines are part of the Order Entry package and are maintained by the Order Entry developers. They were reviewed by the VA FileMan developers.)
- If the init is being built from an entry in the PACKAGE file, enter the namespaced ONIT routine into the POST-INITIALIZATION ROUTINE field in the PACKAGE file entry. It will be run automatically as a post-init.

If the init is not being built from the PACKAGE file, the developer must tell the installers to run the ONIT routines manually AFTER they run the init routines.

- Now, build the actual init using DIFROM in a normal way as described above.

DIFROM has been modified for Order Entry to automatically pick up the entry from the PACKAGE PARAMETERS multiple of the ORDER PARAMETERS file. When the init is run any pointers back to the PROTOCOL file that are contained in this entry are resolved. See the documentation on running the init for more information about resolving pointers during package installation.

RUNNING DIFROM (STEPS 1-17)

Running DIFROM is an interactive process. Prompts are presented to which the developer responds. The dialog is described below. In addition, the internal workings of the DIFROM process are detailed. The different parts of running DIFROM are shown in the order in which they occur.

1. Starting DIFROM
2. Preliminary Validations
3. Package Identification
4. Identifying the Init Routines
5. Specifications for Exported Files
6. Entering Current Version Information
7. Including Templates (No Package File Entry)
8. Including Other Package Components
9. Exporting File Security
10. Specifying Routine Size
11. DIFROM Gathers Miscellaneous Package Components
12. DIFROM Builds Routines Containing Data Dictionaries
13. DIFROM Builds Routines Containing Data Values
14. DIFROM Builds Routines Containing Security Access Codes
15. DIFROM Gathers Templates and Forms
16. DIFROM Completes Building Routines of Package Components
17. DIFROM Completes the Code that Runs the Init

WARNING: DIFROM is not to be used by VA developers.

DIFROM has been replaced by the Kernel Installation and Distribution system (KIDS). Note that DIFROM does not support new VA FileMan V. 22.0 data dictionary structures! If new style indexes or keys are added to any file, they will not be transported by DIFROM.

1. Starting DIFROM

In order to run DIFROM, the developer must have programmer access (i.e., DUZ(0) contains @). There is no menu option for running DIFROM. It must be run by using the M command D ^DIFROM from programmer mode.

2. Preliminary Validations

DIFROM compares the version number from the second line of the DIFROM routine with the VA FileMan version node from the M Operating System file. This node is ^DD("VERSION"). If the version numbers do not match, an error message is displayed and the program exits.

DIFROM then makes sure DUZ(0)["@". If not, the developer will see an error message and the program exits.

3. Package Identification

Next, DIFROM prompts the developer for the 2-4 character Package name. DIFROM uses these characters to do a lookup for a matching PREFIX on the Package file. If a match is found, DIFROM uses information from the Package file entry when building the init.

Even if no matching entry is found, the process continues. In that situation, DIFROM will prompt the developer for the necessary information that is otherwise stored in the Package file. In this way, an init can be built on the fly.

4. Identifying the Init Routines

DIFROM creates a routine name by appending the suffix INIT to the package's namespace. The developer is informed of the name. DIFROM determines whether a routine called nmspINIT is already on the system. If one exists, DIFROM prints a warning. The developer decides whether or not to continue.

Note that the INI* routines that DIFROM creates overlay any INI* routines with the same name that exists on the system. This situation will not cause problems when running inits. However, to avoid confusion for the user, it is suggested that previous init routines under the same namespace be deleted before rebuilding the init.

5. Specifications for Exported Files

DIFROM next asks the developer whether any data dictionaries will be included with the init. The developer must answer YES in order to include either DDs or file data.
If the init is being built from a PACKAGE file entry, the developer is given the option to display online the information in the FILE multiple from the relevant PACKAGE file entry. If the FILE multiple has no entries or if the developer is building an init without the PACKAGE file, the developer is prompted for a list of files to be included in the init.

If the developer does not want to accept the PACKAGE file information as shown or if the init is being built on the fly, DIFROM allows the developer to enter or edit the FILE multiple's data. This data specifies how to include the files in the init and what installation options the installer will have at the time the init is run on the target system. The documentation describing the FILE multiple of the PACKAGE file details the questions the developer will see and the significance of the answers.

A developer can send only some of the fields from a file, that is, send a partial file. The FIELDS multiple contains a list of exported fields when a partial file is being exported. Normally, the .01 field of a file is automatically exported even if it is not specified in the FIELDS multiple. However, the developer has the option of sending or not sending the .01 field of File #200 (the NEW PERSON file). If a partial of File #200 is being sent and the package does not have Kernel's namespace (XU), the .01 field will be sent only if it is specified for export.

If the init is being built from a PACKAGE file entry, DIFROM next loops through each of the template multiples. It builds a list of the templates to be included in the init. There are multiples for INPUT, PRINT, SORT and SCREEN (FORM) templates. The developer uses these multiples to send templates that are not within the package's namespace.

6. Entering Current Version Information

DIFROM next prompts for information that is required by the VA Programming Standards and Conventions (SAC) to appear on the second line of all routines. The developer must enter the package name (if it cannot be picked up from the Package file), the version number, and the date distributed. The existing Package file entry is updated with this information. The version number entered at this step will be used to build target system nodes that look like the following line of code:

```
^DD(File#,0,"VR")=Version Number
```

7. Including Templates (No Package File Entry)

Next, if the init is not being built from a Package file entry, the developer is asked "Do you want to include all the templates?" If the question is answered YES, the init will include ALL templates associated with the files being sent, regardless of their namespace. If the question is answered NO, only namespaced templates are included.

NOTE: If there is a Package file entry, namespaced templates and templates in the template multiples are automatically sent. See discussion below for details.

8. Including Other Package Components

DIFROM asks the developer if OPTIONS, BULLETINS, SECURITY KEYS, FUNCTIONS, and HELP FRAMES should be included in the init. Whereas templates are always sent with the init, the developer must specifically ask that these other components be included. The developer's choices are saved in a list. Only components in the package's namespace are included. There is currently no way to send ones that are not namespaced. Also, if their namespace appears in the EXCLUDED NAME SPACE multiple in the Package file, they are not sent.

9. Exporting File Security

DIFROM next asks the developer whether security codes (e.g., READ, WRITE, and LAYGO access) should be sent with the DDs.

10. Specifying Routine Size

Then DIFROM prompts the developer for maximum routine size. This size determines how large the init routines that contain the data will be. The routines that contain the code that is executed to install the data are of fixed size. DIFROM obtains the default value for maximum routine size from ^DD("ROU"). The size of the init routines cannot be less than 2000 characters. The upper limit should be set in accordance with current portability standards.

11. DIFROM Gathers Miscellaneous Package Components

At this point, the interactive part of building the init is complete. DIFROM now uses the information provided by the developer along with data stored in the Package file entry (if one exists) to build the init routines.

DIFROM first checks the developer's answers to the questions about sending OPTIONS, BULLETINS, KEYS, FUNCTIONS, and HELP FRAMES. For each one that the developer elected to send, DIFROM reads through entries in the associated file and picks up those entries in the package's namespace. For each one, DIFROM makes sure that the name is not included in one of the entries from the EXCLUDED NAME SPACE multiple. For example, if an OPTION's name is PRCZ TEST OPTION and the namespace of the init is PRC, the OPTION is a candidate for export. However, if PRCZ is entered in the EXCLUDED NAME SPACE multiple of the Package file entry, the OPTION is not sent. The data for each component to be included in the init routines is loaded into the ^UTILITY global.

Namespaced bulletins can be sent. However, data in the MAIL GROUP multiple, a pointer to the Mail Group file, is not sent. On the target system, the installer is reminded that mail groups may need to be added to bulletins.

12. DIFROM Builds Routines Containing Data Dictionaries

Next, DIFROM builds the routines containing the DDs, data, and file security. To save space in the init routines, cross-references are not sent. They are rebuilt when the inits are run on the target system. DDs are turned into routines that hold parts of each data dictionary node on separate lines. The global reference is listed on one line; its value is recorded on the next line. Thus, for each node in the data dictionary there are two corresponding lines in the init routine.

The routine lines that hold data dictionary information begin with two semicolons. This format conforms to the VA programming standard for using $TEXT to reference routine lines. When the data dictionaries are put into place during the init process, the lines are referenced using indirection as follows:

```
^DD(442,0)="value"    becomes      ;;^DD(442,0)
                                   ;;="value"
```

If the "value" is too long to fit on a single line, it is divided between two lines. The first "value" line starts with a tilde (~) and the second with an equal sign (=).

If the installer chooses to update the data dictionaries, data dictionary nodes on the target system are overwritten. This will bring in newly-defined fields, including specifications for cross-references or triggers. It will also replace existing field definitions (data dictionary nodes) with incoming definitions. Thus, revisions of existing fields may occur. However, the process will not alter nodes that exist on the target system but that are not in the incoming data dictionary. For example, if a field has been deleted from the source system's data dictionary, that field will not be

deleted on the recipient's system. Instead, a pre-init program can be used to delete obsolete fields and obsolete data dictionary nodes.

If auditing is turned on at the sending site, the DD node indicating that auditing should occur will be sent. In this situation, auditing will be turned on at the installing site if the data dictionaries are updated.

13. DIFROM Builds Routines Containing Data Values

DIFROM stores data values differently than it stores data dictionary information. The recipient's data dictionaries may be updated directly, node by node, but data must first be evaluated for a match of entries. As described in the Running an Init section below, updating of the target system's data is done only after checking for matches. For this reason, the init routines first store data values in a ^UTILITY global structure that is rebuilt on disk on the target system. This allows the existing and incoming values to be compared.

The routines that DIFROM creates to transport data are similar in structure to the ones created to transport data dictionaries. The nodal address and associated values are maintained on separate program lines. The structure as it appears on the target system and as it is contained in the init routines is:

```
^UTILITY(U,$J,file#,entry#,node)="value"

            transported as     ;;^UTILITY(U,$J,file#,entry#,node)
                               ;;="value"
```

14. DIFROM Builds Routines Containing Security Access Codes

DIFROM creates a separate global array for storage of security Access Codes if the developer indicates that they should be sent with the package. Security codes are extracted from the data dictionaries and saved in another routine. The nodes containing security information such as write protection on a field are not in the same routine as the data definition of the field.

When the package is installed, the recipient is asked whether security codes should be updated. A positive response invokes a special program that puts the nodes containing security information back in the DD structures. For example:

```
^DIC(442,0)                is always installed
^DIC(442,0,"DD")="@"       is only installed upon user request
```

DIFROM sends most file security codes only if the developer has answered YES to the question about sending security. However, the following two kinds of field level security codes are always sent:

- **Write Access** If set to the "^" (a write protected field) or the "@" (programmer access required), or if the field is a MUMPS type field.
- **Delete Access** If set to the "@" (programmer access required) or if the field is a MUMPS-type field.

15. DIFROM Gathers Templates and Forms

Next, DIFROM puts INPUT, PRINT and SORT templates into ^UTILITY. It then puts FORMS (SCREEN TEMPLATES) into ^UTILITY along with any BLOCKS that are pointed-to by the FORMS being included. DIFROM uses the list compiled during the interactive dialog with the developer to select templates. Namespaced templates, with the exception of any in the EXCLUDED NAME SPACE multiple of the Package file entry, are always included. In addition, any templates in the template multiples are also included. If the init being built does not have a corresponding Package file entry and the developer asked to send ALL templates, all templates associated with the files being sent in the init are selected regardless of their namespace.

FILEGRAM and EXTRACT templates are sent along with the other entries in the PRINT template file. However, the templates used by the Export Tool (Selected Fields for Export and Export) are never included by DIFROM when a package's components are assembled. These templates must be created at the local site.

16. DIFROM Completes Building Routines of Package Components

DIFROM reads through everything it stored in the ^UTILITY global and builds init routines containing the information. This information includes the TEMPLATES, OPTIONS, BULLETINS, KEYS, FUNCTIONS, and HELP FRAMES.

NOTE: Except for TEMPLATES, only those components in the package's namespace can be sent.

The Package file entry, if any, is automatically included with the init. This entry will be added to the target system when the init is run. It will completely replace an entry with the same name at the target site. This entry is a record of what was included with the init.

17. DIFROM Completes the Code that Runs the Init

DIFROM's final step is to build those routines that contain the code that is executed when the init is run. The code retrieves and installs all of the data components that are being sent. The code that goes into the nmspINI0, nmspINI1, nmspINI2, nmspINI3, nmspINI4 and nmspINIT routines is nearly identical for all regular inits. (If the package's namespace is less than 4 characters, the routines are named nmsINIT0 to nmsINIT4.)

IMPORTING DATA

DIFROM: RUNNING AN INIT (STEPS 1-16)

A package is installed on the target system by "running the init" for the package. Here, the process for installing a package from inits is described in the order in which it occurs.

1. Preliminary Steps
2. Check of Version Number
3. Running Environment Check Routine (DIFROM and DIFQ Variables)
4. Determining Install Status of DDs and Data
5. Determining Install Status of Security Codes
6. Determining Install Status of other Package Components
7. Starting the Update
8. Running the Pre-init After User Commit Routine
9. Installing Data Dictionaries
10. Installing Data
11. Reindexing the Files
12. Installing Other Package Components
13. General Processing
14. Special Processing
15. Running the Post-Initialization
16. Recording the Install on the Target System

1. Preliminary Steps

As a safeguard, the target system should always be backed up before running an init. This will allow the system to be restored should an error, possibly corrupting the database, occur when the init is run.

To ensure that the installer has complete access to all files being installed during an init, the installer should have programmer access when running the init.

Init routines must be run from programmer mode after the routines have been loaded onto the target system. For example, to run an init with the package namespace of ZZTK, do the following:

```
D ^ZZTKINIT
```

2. Check of Version Number

When an init is built, the VA FileMan version number of the source system is put into the init routine. When the init is run, that version number is compared to the version number of the target system that is stored in the MUMPS OPERATING SYSTEM file node, ^DD("VERSION"). If the init was built using a version of FileMan later than the one on the target system, an error message is displayed and the installer is not allowed to continue running the init.

This precaution is necessary because a newer version of VA FileMan may contain features and DD structures that are not recognized by previous versions. Trying to use the new features or to install the new structures on an older system could cause the installation to fail or to produce undesirable results.

3. Running Environment Check Routine (DIFROM and DIFQ Variables)

The ENVIRONMENT CHECK ROUTINE is a field in the Package file that may indicate a routine to run as part of the init process. If the developer has included a routine name in the ENVIRONMENT CHECK ROUTINE field, this routine is run next. The routine is written by package developers to provide capabilities not possible from the init routines alone.

The developer's Environment Check routine may be used to explore the current system and halt the init process under certain conditions. For example, if a prior version of the package must be initialized before this one, a warning message might be displayed and the process halted.

The DIFQ variable is used to stop the init process. Within the Environment Check routine, the developer may kill DIFQ if conditions warrant the stopping of the init process.

The DIFROM variable is defined throughout the init process. It contains the version number of the incoming package. The developer can use it for checking in any pre- or post-init routines.

4. Determining Install Status of DDs and Data

Next, the init determines which file's data dictionaries and data values will be installed on the target system. Based on the parameters the developer included in the init in combination with the environment encountered at the target site, the installer is asked a series of questions for each file.

NOTE: With the one exception mentioned below, no changes are made on the target system at this time. The answers obtained are saved to be used later in the installation process when the target system is updated.

The exported files are checked one-by-one. What happens to each file is described in the list that follows:

- The name of the file is displayed to the installer whether or not a partial DD is being sent and whether or not data is coming with the file. If there is not a file with the same file number on the target system, the DD will be installed and the installer is next presented with the questions concerning the installation of the data.
- If there is already a file under that number and the names are the same, the init tells the installer.

  NOTE: You already have the 'file name' File.
- If there is a file with that number, but the file names do not match, the installer is asked if the name should be replaced. The default response is NO. In the event of miss matched file names, the following instructions are provided:
  - If the installer is sure that the files are really the same and that just the name has been changed, this question should be answered YES. In this case, the init does a DIE call to change the name of the file on the target system. (This is the only situation in which the target system is altered during this phase of the install.) The init then continues with the dialog as if the file names had matched in the first place.
  - If the installer determines that the files are not the same and answers NO, then the init asks if the incoming file should replace the file currently on the system. If the installer answers NO to this question, the current file will be left unchanged. However, this choice will result in the installation of an incomplete package. Therefore, if this happens, the installation should probably be stopped and the package developer consulted.
  - If the installer chooses to replace the file on the target system, the init asks if the current file's data and templates should be kept. Based on

the answers to these questions, the current DD (and optionally the data and templates) will be deleted, before the new DD is brought in. A call to DIU is set up to do the DD deletion and also to delete the data and templates if the installer so instructs. See the description of EN^DIU2 for additional information.

- If to this point the DD will be installed, the init checks if the developer defined a screen to determine whether or not to install the DD. The existence of a developer-defined screen overrides the installer's ability to decide if the DD should be installed. If the screen exists and its conditions are not met, the DD will not be installed but the init continues. The package developer should indicate what to do when the screen stops the DD from installing.
- If the developer decided to let the installer determine if the DD will be installed, the init asks if an existing DD should be overwritten. If the installer answers NO, the existing DD will be unchanged. Package developers should indicate when it is okay to answer this question NO.
- If data is being brought with the DD and the package developer decided to ask the question, the installer is asked whether to overwrite the target system data or merge it with the incoming data. The package developer determines whether data merges or overwrites; the installer can decide if the data will be installed, not how it will be installed. The developer should advise the installer on how to answer this question.
- If the developer did not give the installer the option of installing the data or not, the init just indicates whether the data will merge with or overwrite the current data.

5. Determining Install Status of Security Codes

Next, if the developer sent file security access codes with the file, the init asks if security codes present on the target system should be overwritten. In most cases, file security is built into files by the developer. However, if there are local security codes that need to be preserved, the installer should answer this question NO.

NOTE: Even if the installer says not to bring in security codes, the init will install the following field security:

- If Write Access is set to the "^" (a write protected field) or the "@" (programmer access required) or the field is a MUMPS type field, Write Access security is installed.
- If Delete Access is set to an "@" (programmer access required) or the field is a MUMPS type field, Delete Access security is installed.

6. Determining Install Status of other Package Components

Next the installer is notified of the kinds of components included in the init. The init asks whether or not to overwrite existing components with the same name. The possible components are INPUT TEMPLATES, SORT TEMPLATES, PRINT TEMPLATES, SCREEN TEMPLATES (FORMS), OPTIONS, FUNCTIONS, BULLETINS, SECURITY KEYS and HELP FRAMES. The developer should instruct the installer if it is all right not to install any of the components included in the init.

7. Starting the Update

Finally, the init asks "ARE YOU SURE EVERYTHING'S OK?" To this point, there are many chances to stop the init with no changes having been made to the target system. However, if the installer answers YES to this question, the init proceeds to install the package. If the installer answers NO, the init process is safely halted.

8. Pre-init After User Commit Routine

First, the init runs the PRE-INIT AFTER USER COMMIT routine if the developer included a routine name in the PRE-INIT AFTER USER COMMIT field of the Package file.

The developer's PRE-INIT AFTER USER COMMIT routine does things that are not possible with the init routines alone. Often, it cleans up DDs or data on the target system before the init routines bring in any of the new DDs or data. For example, obsolete fields or parts of field definitions can be removed from data dictionaries.

9. Installing Data Dictionaries

Next, the init installs the data dictionaries for files sent with the init. The data dictionaries are then reindexed.

Data dictionaries are set in place node-by-node, integrating with what already exists. In other words, if a node is brought in by the DD that exists on the target system, the existing node will be replaced. However, if a node that is not included in the init exists on the target system, the init will NOT delete that node. This feature allows users to create local fields and cross-references.

However, this does mean that the developer must carefully consider what the target system's data dictionary will look like after installation. For example, if the developer in the account used to build the init changes the definition of a field or removes a cross-reference, the field or cross-reference must be deleted or otherwise cleaned up on the target account by the PRE-INIT AFTER USER COMMIT routine. This cleanup ensures that the data dictionary will not end up with an inconsistent structure after the init.

Further, each line of a word processing field resides on a separate node. Thus, a change in one of the field attributes that is a word processing field (e.g., field description or technical description) may not completely overwrite a pre-existing attribute. If the incoming value has fewer lines than the pre-existing one, the install will not delete the surplus lines automatically.

10 Installing Data

Next, the init brings in data that was sent with the files.

Depending on the developer's specifications, incoming data either overwrites or merges with data existing on the target system. In either case, if an incoming entry or subentry doesn't exist on the current system, one will be added. If an existing entry or subentry is found and if data is to be overwritten, each field's value will be replaced with non-null incoming values. Null values will not overwrite existing values. If data is to be merged, only those fields with null values will be updated with incoming values. Hence, when merging, new values will be added without altering any pre-existing ones.

Since the installation of data is dependent on whether or not an incoming entry or subentry already exists on the target system, the init must determine if they are the same. The process, described as follows, is repeated for each incoming entry or subentry:

- **Checking the B Cross-reference or Zero Node**

  The B cross-reference holds the entry's name (.01 field) along with the internal entry number. If a B cross-reference exists for the file, it is searched for an existing value that matches the incoming one. (The B cross-reference holds the name as a subscript.) The maximum length of subscripts is defined for each operating system and is stored in the MUMPS OPERATING SYSTEM file (#.7). The init uses this length, for example, 63 (default) or 99 as the limit of characters to compare.

  Files occasionally lack a B cross-reference. In this case, the init examines the actual data (first piece of the entry's zero node) for a match of values.

If a match (either of the B cross-reference or of the first piece of the zero node) is not found, the incoming entry is considered new and is added to the file. If a match is found, additional checks (discussed below) are made to determine whether the entries may be associated.

- **Using the Internal Entry Number to Verify a Match**

  Once a match of the .01 fields of the incoming and existing entries is found, the init determines whether the internal entry numbers of the two entries are related. If the file has a defined .001 field, internal entry number is a meaningful attribute of an entry. In this situation, when the name and internal entry numbers match, identifiers are checked to verify the match.

  If the INPUT transform of the .01 field contains DINUM, it operates in the same way as a .001 field. In this case, the .01 field and the internal entry number must match for the entries to be considered the same.

  After a match is established based on the .001 field (or DINUMed .01 field), the identifiers are checked. If the identifiers for the two entries are the same, the entries are considered the same. If the identifiers do not match, the new entry is not installed at all.

- **Using Identifiers to Verify a Match**

  If the file is not referenced by number (i.e., .001 field does not exist) and there are duplicate B cross-references or entries in the file with duplicate .01 fields, the init cannot resolve the ambiguity without identifiers. A well-designed file uses one or more identifiers so that each entry is unique with respect to name and identifiers. If the file lacks identifiers and a .001 field, the init will associate the incoming entry with the first existing entry with a matching name.

  If identifiers exist, the init gets the global location of the identifier (piece position) from the data dictionary and uses indirection to retrieve the identifier's value from the ^UTILITY storage global. This value is then compared with the existing entry's identifier value for a match. Only identifiers that have valid field numbers are used in this process.

  The init matches identifiers in the same way it matches .01 fields. If the values of all the incoming identifiers match the existing ones, the two entries are considered to be the same. If the values don't match, the possibility of identity is rejected and the search continues. If none of the values for existing entries matches the incoming entry, the incoming entry is considered new

and is added to the file. However, as mentioned above, if a .001 field exists or the .01 field is DINUMed, the entry is not installed if the identifiers differ.

Once the internal entry number on the target system for matching entries is found, it is used to place the incoming data, either by merging with or overwriting existing values.

NOTE: No audit trail is kept of data brought in by an init even if the audit flag is on for a field receiving data.

11. Reindexing the Files

Once all the new data has been integrated, the files are reindexed. If any of the files have compiled cross-references, the compiled cross-reference routines are rebuilt. Then, if any data was sent for a file, the init reindexes ALL cross-references for ALL the records in the file. Only the SET logic is executed.

12. Installing Other Package Components

Next, the init brings in the remaining components built into the init. They are installed in the following order:

- Help Frames
- Bulletins
- PACKAGE file entry for the package being installed
- PACKAGE PARAMETER multiple from the ORDER PARAMETER file (an Order Entry file)
- Options
- Security Keys
- Functions
- Print Templates
- Sort Templates
- Input Templates
- Blocks associated with Screen Templates (Forms)
- Screen Templates (Forms) themselves

The init might contain some or all of these components. They consist of entries that are placed into pre-existing files. Many of them are prefixed with the package namespace.

There is special coding in DIFROM to bundle and install data sent from the HELP FRAME, BULLETIN, OPTION, INPUT TEMPLATE, etc., files. For example, DIFROM resolves pointers for these files. (It does not resolve pointers for data sent for other files in an init.) To resolve pointers, DIFROM replaces, in the init routines, a pointer to another file with the pointer's external value. When the data is installed at the target site, the init routines use this external value for a lookup in the B cross-reference of the pointed-to file. When the corresponding entry number is found, the external value is replaced with this entry number as the new pointer value. In this way, the values of pointer fields are correct for the data brought in by the init.

13. General Processing

The general process used for installing each of the package components is described here. Component-specific special processing is described following this section.

The init reads the name of the incoming entry from the ^UTILITY global and searches for a matching name in the relevant file's B cross-reference. The cross-reference for the HELP FRAME file (#9.2), for example, looks like this:

```
^DIC(9.2,"B",entryname,DA)
```

If an exact match is not found, the incoming entry is considered new and is added as a new file entry. If an exact match is found, special processing, described in detail below, is done. Each different type of entry has its own special processing. Unless noted in the special processing, the entire matching old entry is deleted from the target system before the new entry is installed.

For either new or replaced entries, other special processing, such as resolving pointers, is done for each different type of entry. This processing is also described in detail below.

Finally, all cross-references on the new or replaced entry are reindexed (SET logic only).

NOTE: Not all files are reindexed.

14. Special Processing

- HELP FRAMES

  If an exact match is found for a HELP FRAME entry, only the existing word processing field TEXT and the multiple fields RELATED FRAME and INVOKED BY ROUTINE are deleted from the existing entry. Then, the new entry is brought in on top of the old one.

  For all entries brought in by the init, the init loops through the RELATED FRAME multiple and resolves the pointer field RELATED FRAME, which is a pointer back to the Help Frame file.

- BULLETINS

  If a matching entry is found, the old entry in the Bulletin file is deleted. However, entries in the bulletin's MAIL GROUP multiple (which identify recipients of the bulletin) present on the target system before the install will remain associated with the bulletin after the incoming bulletin is installed.

  The init displays each bulletin brought in by the init and reminds the installer to "Remember to add mail groups for new bulletins."

- PACKAGE FILE ENTRIES

  The current date/time is stuffed into the field DATE INSTALLED AT THIS SITE, within the VERSION multiple for the current version of the package.

  The pointer field PRIMARY HELP FRAME is resolved.

- PACKAGE PARAMETERS entry in the ORDER PARAMETERS file (an Order Entry file)

  Pointer fields DISPLAY GROUP DEFAULT, PROTOCOL TO EXPORT, DEFAULT PROTOCOL, and MENU are resolved. If pointers to the Protocol file cannot be resolved because the pointed-to protocol cannot be found, the init routines add a new entry to the Protocol file (with just a .01 field) in order to resolve the pointer. This is done because PROTOCOLS are exported in a special set of routines (called ONIT routines) that are normally executed as a post-init.

- OPTIONS

  If a matching entry is found, the entire old entry is not deleted. Only the DESCRIPTION field (a word processing field) and the ITEMS multiple (containing menu items) are deleted from the old entry before the new one is brought in.

For example, if the site has a local lock on an OPTION, and no lock is brought in by the init, the local lock is preserved.

The pointer fields SERVER BULLETIN, SERVER MAIL GROUP, PACKAGE, HELP FRAME and the .01 field of the ITEMS multiple (which points back to the Option file) are all resolved.

- SECURITY KEYS

  No special processing, except that if a matching entry is found in the target system, it is merged rather than replaced. Note that pointers in the SUBORDINATE KEY multiple are not resolved; so, data should not be exported in that multiple.

- FUNCTION

  No special processing is done for the Function file.

- PRINT, INPUT and SORT TEMPLATES

  The only special processing done for these templates is that after they are all installed, compiled PRINT and INPUT templates are automatically recompiled. The init uses the system's preferred routine size from the MUMPS OPERATING SYSTEM file (#.7) when compiling these templates. It is possible that the recipient of the init could already have routines with the same names that the compiling routine will use. Thus, the developer should warn the installer of the routine names that will be used by incoming compiled templates, especially if the developer is sending templates that are not namespaced.

- SCREEN TEMPLATES (FORMS)

  Any BLOCKS that are pointed-to by FORMS are automatically included in the init routines. The BLOCKS are installed first, with no special processing. Then, the FORMS are installed. Finally, pointers to the Block file from the Form file are resolved.

15. Running the Post-Initialization Routine

At the developer's discretion, there may be a routine identified in the POST-INITIALIZATION ROUTINE field in the Package file. This routine is written by the package developers and provides added capability which is not possible within the init routines alone.

If the developer has included a POST-INITIALIZATION ROUTINE in the init, it is run now.

The POST-INITIALIZATION ROUTINE may be used to do cleanup after all of the other components contained in the init have been installed. For example, it might delete obsolete OPTIONS and update Option file pointers, check the status of such things as file protection, or issue some additional information to the installer. It might also do some sort of data conversion. For example, the routine might move some old data to a new location in a file to match a changed data dictionary.

16. Recording the Install on the Target System

Then, if pre- or post-init routines were included, the Package File fields that track the date and time that those routines were run are updated with the current date and time. If any new files were added to the target system, the record count of the File of Files is updated to reflect the new files. Then, the init routines update any VERSION number nodes on the files that have been specified by the Package developer. Finally, the VERSION number node is set in the Package File entry (if any).

The init is now complete.

Glossary

NOTE: You can also search the Free Online Dictionary of Computing.

.001 Field	A field containing the internal entry number of the record.
.01 Field	The one field that must be present for every file and file entry. It is also called the NAME field. At a file's creation the .01 field is given the label NAME. This label can be changed.
Access Codes	In VA FileMan, a string of codes that determines your security access to files, fields, and templates. In Kernel, you enter an Access Code to identify yourself during signon.
Alternate Editor	One of the text editors available for use from VA FileMan. Editors available vary from site to site. They are entries in the ALTERNATE EDITOR file (#1.2).
At-sign ("@")	A VA FileMan security Access Code that gives the user programmer-level access to files and to VA FileMan's developer features. See Programmer Access. Also, the character "@" (i.e., at-sign) is used at VA FileMan field prompts to delete data.
Audit Trail	The record or log of an ongoing audit.
Auditing	The monitoring and recording of computer use.
Backward Pointer	A pointer to your current file from another file; used in the extended pointer syntax.
Boolean Expression	A logical comparison between values yielding a true or false result. In M, zero means false and non-zero (often one) means true.
Canonic Number	A number with no leading zeros and no trailing zeros after a decimal point.
Caption	In ScreenMan, a label displayed on the screen. Captions often identify fields that are to be edited.

Command Area In ScreenMan, the bottom portion of the screen used to display help information and to accept user commands.

Cross-reference An attribute of a field or a file that identifies an action that should take place when the value of a field is changed. Often, the action is the placement of the field's value into an index. A Traditional cross-reference is defined with a specific field. A New-Style cross-reference is a file attribute and can be composed of one or more fields. New-Style cross-references are stored in the INDEX file (#.11).

Cursor On your display terminal, the line or rectangle identifying where your next input will be placed on the screen.

Data Dictionary A record of a file's structure, its elements (fields and their attributes), and relationships to other files. Often abbreviated as DD.

DATA TYPE The kind of data stored in a field. NUMERIC, COMPUTED, and WORD-PROCESSING are examples of VA FileMan DATA TYPEs.

Database An organized collection of data spanning many files. Often, all the files on a system constitute that system's database.

Decentralized Hospital Computer Program (DHCP) See **V***IST***A**.

Default A computer-provided response to a question or prompt. The default might be a value pre-existing in a file. Often, you can change a default.

Device Prompt A Kernel prompt at which you identify where to send your output.

Edit Window In ScreenMan, the area in which you enter or edit data. It is highlighted with either reverse video or an underline. In Screen Editor, the area in which you enter and edit text; the area between the status bar and the ruler.

Entry A record in a file. "Entry" and "record" are used interchangeably.

Extended Pointers A means to reference fields in files other than your current file.

Field In an entry, a specified area used to hold values. The specifications of each VA FileMan field are documented in the file's data dictionary.

Field Number The unique number used to identify a field in a file. A field can be referenced by "**#**" followed by the field number.

File A set of related records (or entries) treated as a unit.

Form In ScreenMan, a group of one or more pages that comprise a complete transaction. Comparable to an INPUT template.

FREE TEXT A DATA TYPE that can contain any printable characters.

Full-screen Editing The ability to enter data in various locations on the two-dimensional computer display. Compare to scrolling mode.

Histogram A type of bar graph that indicates frequency of occurrence of particular values.

Identifier In VA FileMan, a field that is defined to aid in identifying an entry in conjunction with the NAME field.

Index An ordered list used to speed retrieval of entries from a file based on a value in some field or fields. The term "simple index" refers to an index that stores the data for a single field; the term "compound index" refers to an index that stores the data for more than one field. Indexes are created and maintained via cross-references.

INPUT Template A pre-defined list of fields that together comprise an editing session.

Internal Entry Number The number used to identify an entry within a file. Every record has a unique internal entry number. Often abbreviated as IEN.

Kernel A **V***IST***A** software package that functions as an intermediary between the host operating system and **V***IST***A** application packages. Kernel includes installation, menu, security, and device services.

Key A group of fields that, taken collectively, uniquely identifies a record in a file or subfile. All fields in a key must have values. The term "simple key" refers to keys that are composed of only one field; the term "compound key" refers to keys that are composed of more than one field. Keys are stored in the KEY file (#.31)

LAYGO A user's authorization to create a new entry when editing a computer file. An acronym for **L**earn **A**s **Y**ou **Go**.

Line Editor The VA FileMan editor that lets you input and change text on a line-by-line basis. The Line Editor works in scrolling mode. See Screen Editor.

Lookup To find an entry in a file using a value for one of its fields.

MailMan An electronic mail system (e-mail) that allows you to send messages to and receive them from other users via the computer. It is part of **V***IST***A**.

Menu A list that includes the names of options from which you can select an activity.

Multiple A VA FileMan DATA TYPE that allows more than one value for a single entry. See Subfile.

MUMPS Abbreviated as M. The American National Standards Institute (ANSI) computer language used by VA FileMan and throughout **V***IST***A**. The acronym MUMPS stands for **M**assachusetts General Hospital **U**tility **M**ulti **P**rogramming **S**ystem.

NAME Field The one field that must be present for every file and file entry. It is also called the .01 field. At a file's creation the .01 field is given the label NAME. This label can be changed.

Navigation

1. Navigation can mean switching your reference point from one file to another.
2. Navigation can also mean moving your cursor around a terminal display or a document using cursor keys and other commands.

Non-canonic Number

A number with either leading zeros, or trailing zeros after a decimal point. M treats non-canonic numbers as text instead of as numbers.

Non-null

A value other than null. A space and zero are non-null values.

Null

Empty. A field or variable that has no value associated with it is null.

Null Response

When replying to a prompt, pressing only the Enter/Return key, abbreviated as **<RET>**, to enter nothing.

Numeric Expression

An expression whose value is a number. Compare to string expression.

Operator

One of the processes done to the elements in an expression to create a value.

Option

A computing activity that you can select, usually a choice from a menu.

Paste

Insert text or other data as input into one computer program that has been copied into a clipboard by the same or by another computer program.

Pattern Match

In M, an operator that compares the contents of a variable or literal to a specified pattern of characters or kinds of characters.

PF keys

Keys on a terminal keyboard labeled **PF1**, **PF2**, etc. that are used to perform special functions instead of displaying visible characters.

POINTER TO A FILE

A field DATA TYPE that contains an explicit reference to an entry in a file. POINTER TO A FILE-type fields are used to relate files to each other.

Pop-up Page In ScreenMan, a page that overlays the regular ScreenMan screen in order to present the contents of a selected Multiple.

Preferred Editor The editor always entered when you access a WORD-PROCESSING-type field; your default editor. Kernel must be present to establish a Preferred Editor.

PRINT Template The stored specifications of a printed report, including fields to be printed and formatting instructions.

Programmer Access The ability to use VA FileMan features that are reserved for application developers. Referred to as "having the at-sign ('@')" because the at-sign is the DUZ(0) value that grants programmer access.

Prompt A question or message from the computer requiring your response.

Record A set of data pertaining to a single entity in a file; an entry in a file.

Record Number See Internal Entry Number.

Relational Navigation Changing your current (or primary) file reference to another file. Relational navigation is accomplished by using the extended pointer syntax without specifying a field in the referenced file.

Required Field A field that cannot be left null for an entry.

Scattergram A graph in which occurrences of two fields are displayed on an X-Y coordinate grid to aid in data analysis.

Screen Editor VA FileMan's Screen-oriented text editor. It can be used to enter data into any WORD-PROCESSING field using full-screen editing instead of line-by-line editing. See Line Editor.

Screen-oriented A computer interface in which you see many lines of data at a time and in which you can move your cursor around the display screen using screen navigation commands. Compare to Scrolling Mode.

ScreenMan The set of routines that supports Screen-oriented data editing and data display.

Scrolling Mode The presentation of the interactive dialogue one line at a time. Compare to Screen-oriented.

SDP An area on disk set aside for temporary, sequential storage of data; an abbreviation for **S**equential **D**isk **P**rocessor. It is available on some M implementations (e.g., DSM-11).

SEARCH Template The saved results of a search operation. Usually, the actual entries found are stored in addition to the criteria used to select those entries.

Security The strategies and procedures used to ensure that user access to data and data structures is controlled and appropriate.

SET OF CODES A field DATA TYPE where a short character string is defined to represent a longer value.

Simple Extended Pointers An extended pointer that uses a pre-existing pointer relationship to access entries in another file.

Sort To place items in order, often in alphabetical or numeric sequence.

SORT Template The stored record of sort specifications. It contains sorting order as well as restrictions on the selection of entries. Used to prepare entries for printing.

Stuff To place values directly into a field, usually with no user interaction.

Subentry An entry in a Multiple; also called a Subrecord.

Subfield A field in a Multiple.

Subfile The data structure of a Multiple. In many respects, a Subfile has the same characteristics as a File.

Terminal Emulation Using one kind of terminal or computer display to mimic another kind. Often used with PC remote communication applications.

Terminal Type The designation of the kind of computer peripheral being used (e.g., the kind of video display or printer). Full terminal type functionality is supplied by Kernel.

Truth Test An evaluation of an expression yielding a true or false result. In M, usually a 1 (true) or a 0 (false) is returned from a truth test.

Up-arrow The ^ character (caret); used in VA FileMan for exiting an option or canceling a response. Also used in combination with a field name or prompt to jump to the specified field or prompt.

Upload Send a file from one computer system to another (usually using communications software).

V*IST*A The Veterans Health Information Systems and Technology Architecture (**V***IST***A**), within the Department of Veterans Affairs, is the component of the Veterans Health Administration that develops software and installs, maintains, and updates compatible computer systems in VA medical facilities. (Previously known as the Decentralized Hospital Computer Program [DHCP].)

Appendix A—VA FileMan Error Codes

INTRODUCTION

Descriptions of the error codes returned by VA FileMan's DBS are contained in this section. When an error condition is recognized, an error code, the text of the error, and (when appropriate) one or more parameters are returned to the client application. The "How the DBS Communicates" section of the "Database Server (DBS)" chapter in this manual describes in detail the array structure in which this information about the error is returned.

The following information is ordered by error code number. After the number is a brief DESCRIPTION of the condition that produced the error.

Then the TEXT of the error is shown. Within the text, information that is inserted into the message at the time it is created is represented by a parameter name surrounded by vertical bars (|). For example, in the text of message number 201, you see "|1|". Parameter 1 represents the variable name that is missing or invalid. When the message is created, the name of the variable causing the error is substituted into the text for the |1|.

After the text, the PARAMETERS associated with the error are listed. Each parameter is followed by a short description. The names of the parameters identify both the place within the text of a message into which they are inserted and the subscript in the PARAM array that identifies them. Some parameter names are constant in all appropriate error messages: FILE representing file number, FIELD representing field number, and IENS representing the IENS. If you need to identify in your application code the file, field, or entry that caused an error, check these subscripts of the PARAM array. Of course, if no parameters are listed, this indicates that there are none associated with the particular error condition.

Error 101

DESCRIPTION:

The option or function can only be done if DUZ(0)="@", designating the user as having programmer access.

TEXT:

Only those with programmer's access can perform this function.

PARAMETERS:

None

Error 110

DESCRIPTION:

An attempt to get a lock timed out. The record is locked and the desired action cannot be taken until the lock is released.

TEXT:

The record is currently locked.

PARAMETERS:

- 'FILE' means File or subfile #.
- 'IENS' means IEN string of entry numbers.

Error 111

DESCRIPTION:

An attempt to get a lock timed out. The File Header Node is locked, and the desired action cannot be taken until the lock is released.

TEXT:

The File Header Node is currently locked.

PARAMETERS:

'FILE' means File #.

Error 120

DESCRIPTION:

An error occurred during the Xecution of a FileMan hook (e.g., an INPUT transform, DIC screen). The type of hook in which the error occurred is identified in the text. When relevant, the file, field, and IENS for which the hook was being Xecuted are identified in the PARAM nodes. The substance of the error will usually be identified by a separate error message generated during the Xecution of the hook itself. That error will usually be the one preceding this one in the DIERR array.

TEXT:

The previous error occurred when performing an action specified in a |1|.

PARAMETERS:

- '1' means Type of FileMan Xecutable code.
- 'FILE' means File#
- 'FIELD' means Field#.
- 'IENS' means Internal Entry Number String.

Error 200

DESCRIPTION:

There is an error in one of the variables passed to a FileMan call or in one of the parameters passed in the actual parameter list.

TEXT:

An input variable or parameter is missing or invalid.

PARAMETERS:

None

Error 201

DESCRIPTION:

The specified input variable is either 1) required but not defined or 2) not valid.

TEXT:

The input variable |1| is missing or invalid.

PARAMETERS:

'1' means Variable name.

Error 202

DESCRIPTION:

The specified parameter is either required but missing or invalid.

TEXT:

The input parameter that identifies the |1| is missing or invalid.

PARAMETERS:

'1' means Parameter as identified in the FM documentation.

Error 203

DESCRIPTION:

An incorrect subscript is present in an array that is passed to FileMan. For example, one of the subscripts in the FDA which identifies FILE, IENS, or FIELD is incorrectly formatted.

TEXT:

The subscript that identifies the |1| is missing or invalid.

PARAMETERS:

'1' means The data element incorrectly specified by a subscript.

Error 204

DESCRIPTION

Control characters are not permitted in the database.

TEXT

The input value contains control characters.

PARAMETERS

'1' means INPUT VALUE

Error 205

DESCRIPTION:

Error message output when a file or subfile number and its associated IEN string are not in sync. (i.e., the number of comma pieces represented by the IEN string do not match the file/subfile level according to the "UP" nodes.)

TEXT:

File# |1| and IEN string |IENS| represent different subfile levels.

PARAMETERS:

- '1' means File or subfile number
- 'IENS' means IEN string

Error 206

DESCRIPTION:

FileMan is trying to pack fields onto a single node for a record, and the data will not fit. The application has asked for too many fields back for this record.

TEXT:

The data requested for record |1| is too long to pack together.

PARAMETERS:

'1' means Record Number.

Error 207

DESCRIPTION:

The library function $$HTML^DILF can encode or decode a string to and from HTML, used within FileMan to pack a value containing embedded ^s into a ^-delimited string. Encoding increases the length of the string. If encoding would cause the length to exceed the portable string length limit, $$HTML^DILF instead returns this error.

TEXT:

The value |1| is too long to encode into HTML.

PARAMETERS:

'1' means Value.

Error 299

DESCRIPTION:

A lookup that was restricted to finding a single entry found more than one.

TEXT:

More than one entry matches the value '|1|'.

PARAMETERS:

- '1' means Lookup Value.
- 'FILE' means File #.
- 'IENS' means IEN String.

Error 301

DESCRIPTION:

Flags passed in a variable (like DIC(0)) or in a parameter are incorrect.

TEXT:

The passed flag(s) '|1|' are unknown or inconsistent.

PARAMETERS:

'1' means Letter(s) from flag.

Error 302

DESCRIPTION:

The calling application has asked us to add a new record, and has supplied a record number, but a record already exists at that number.

TEXT:

Entry '|IENS|' already exists.

PARAMETERS:

- 'FILE' means File #.
- 'IENS' means IEN String.

Error 304

DESCRIPTION:

The problem with this IEN string is that it lacks the final ','. This is a common mistake for beginners.

TEXT:

The IENS '|IENS|' lacks a final comma.

PARAMETERS:

'IENS' means IENS.

Error 305

DESCRIPTION:

A root is used to identify an input array, but the array is empty.

TEXT:

The array with a root of '|1|' has no data associated with it.

PARAMETERS:

'1' means Passed root.

Error 306

DESCRIPTION:

When an IENS is used to explicitly identify a subfile, not a subfile entry, then the first comma-piece should be empty. This one wasn't.

TEXT:

The first comma-piece of IENS '|IENS|' should be empty.

PARAMETERS:

'IENS' means IENS.

Error 307

DESCRIPTION:

One of the IENs in the IENS has been left out, leaving an empty comma-piece.

TEXT:

The IENS '|IENS|' has an empty comma-piece.

PARAMETERS:

'IENS' means IENS.

Error 308

DESCRIPTION:

The syntax of this IENS is incorrect. For example, a record number may be illegal, or a subfile may be specified as already existing, but have a parent that is just now being added.

TEXT:

The IENS '|IENS|' is syntactically incorrect.

PARAMETERS:

'IENS' means IENS.

Error 309

DESCRIPTION:

A multiple field is involved. Either the root of the multiple or the necessary entry numbers are missing.

TEXT:

There is insufficient information to identify an entry in a subfile.

PARAMETERS:

None

Error 310

DESCRIPTION:

Some of the IENS subscripts in this FDA conflict with each other. For example, one IENS may use the sequence number ?1 while another uses +1. This would be illegal because the sequence number 1 is being used to represent two different operations.

Consult your documentation for an explanation of the various conflicts possible.

The IENS returned with this error happens to be one of the IENS values in conflict.

TEXT:

The IENS '|IENS|' conflicts with the rest of the FDA.

PARAMETERS:

'IENS' means IENS.

Error 311

DESCRIPTION:

Adding an entry to a file without including all required identifiers violates database integrity. The entry identified by this IENS lacks some of its required identifiers in the passed FDA.

TEXT:

The new record '|IENS|' lacks some required identifiers.

PARAMETERS:

'IENS' means IENS.

Error 312

DESCRIPTION:

All required identifiers must be present for a new entry to be filed. One or more of those fields is missing for the (sub)file.

TEXT:

The list of fields is missing a required identifier for File #|FILE|.

PARAMETERS:

FILE means File or subfile #.

Error 330

DESCRIPTION:

The value passed by the calling application should be a certain data type, but according to our checks it is not.

TEXT:

The value '|1|' is not a valid |2|.

PARAMETERS:

- '1' means Passed Value.
- '2' means Data Type.

Error 348

DESCRIPTION:

The calling application passed us a variable pointer value. That value points to a file that does not exist or that lacks a Header Node.

TEXT:

The passed value '|1|' points to a file that does not exist or lacks a Header Node.

PARAMETERS:

'1' means Passed Value.

Error 351

DESCRIPTION:

When passing an FDA to the Updater, any entries intended as Finding or LAYGO Finding nodes must include a .01 node that has the lookup value. This value need not be a legitimate .01 field value but it must be a valid and unambiguous lookup value for the file.

TEXT:

FDA nodes for lookup '|IENS|' omit a .01 node with a lookup value.

PARAMETERS:

- 'FILE' means File #
- 'IENS' means IENS Subscript for Finding or LAYGO Finding node.

Error 352

DESCRIPTION:

When passing an FDA to the Updater, any entries intended as LAYGO or LAYGO Findings nodes must include .01 node. Every new entry must have a value for the .01 field.

TEXT:

The new record '|IENS|' for file #|FILE| lacks a .01 field.

PARAMETERS:

- 'FILE' means File #
- 'IENS' means IENS Subscript for Finding or LAYGO Finding node.

Error 401

DESCRIPTION:

The specified file or subfile does not exist; it is not present in the data dictionary.

TEXT:

File #|FILE| does not exist.

PARAMETERS:

'FILE' means File number.

Error 402

DESCRIPTION:

The specified file or subfile lacks a valid global root; the global root is missing or is syntactically not valid.

TEXT:

The global root of file #|FILE| is missing or not valid.

PARAMETERS:

- 'FILE' means File number.
- 'ROOT' means File root.
- 'IENS' means IEN String.

Error 403

DESCRIPTION:

The File Header Node, the top level of the data file as described in the Programmer Manual, must be present for FileMan to determine certain kinds of information about a file.

TEXT:

File #|FILE| lacks a Header Node.

PARAMETERS:

'FILE' means File #.

Error 404

DESCRIPTION:

We have identified a file by the global node of its data file and found its Header Node. We needed to use the Header Node to identify the number of the file, but that piece of information is missing from the Header Node.

TEXT:

The File Header node of the file stored at |1| lacks a file number.

PARAMETERS:

'1' means File Root.

Error 405

DESCRIPTION:

The NO EDIT flag is set for the file. No instruction to override that flag is present.

TEXT:

Entries in file |1| cannot be edited.

PARAMETERS:

- '1' means File Name.
- 'FILE' means File number.

Error 406

DESCRIPTION:

The data definition for a .01 field for the specified file is missing. This file is therefore not valid for most database operations.

TEXT:

File #|FILE| has no .01 field definition.

PARAMETERS:

'FILE' means File #.

Error 407

DESCRIPTION:

The subfile number of a word processing field has been passed in the place of a file parameter. This is not acceptable. Although we implement word processing fields as independent files, we do not allow them to be treated as files for purposes of most database activities.

TEXT:

A word-processing field is not a file.

PARAMETERS:

'FILE' means Subfile # of word-processing field.

Error 408

DESCRIPTION:

The file lacks a name. For subfiles, $P(^DD(file#,0),U) is null. For root files, $O(^DD(file#,0,"NM",""))="".

TEXT:

File# |FILE| lacks a name.

PARAMETERS:

'FILE' means File #.

Error 409

DESCRIPTION:

The indicated file does not exist in the FileMan database.

TEXT:

File '|1|' could not be found.

PARAMETERS:

1 means File name or number.

Error 420

DESCRIPTION:

A cross-reference was specified for a lookup, but that cross-reference does not exist on the file. The file has entries, but the index does not. This error implies nothing about whether the index is defined in the file's DD.

TEXT:

There is no |1| index for File #|FILE|.

PARAMETERS:

- '1' means Cross-reference name.
- 'FILE' means File number.

Error 501

DESCRIPTION:

A search of the data dictionary reveals that the field name or number passed does not exist in the specified file.

TEXT:

File #|FILE| does not contain a field |1|.

PARAMETERS:

- '1' means Field name or number.
- 'FILE' means File number.
- 'FIELD' means Field number.

Error 502

DESCRIPTION:

The field has been identified, but some key part of its definition is missing or corrupted. ^DD(file#,field#,0) may not be defined. Some key piece of that node may be missing.

TEXT:

Field# |FIELD| in file# |FILE| has a corrupted definition.

PARAMETERS:

- 'FILE' means File #.
- 'FIELD' means Field #.

Error 505

DESCRIPTION:

The field name passed is ambiguous. It cannot be determined to which field in the file it refers.

TEXT:

There is more than one field named '|1|' in File #|FILE|.

PARAMETERS:

- '1' means Field name.
- 'FILE' means File #.

Error 510

DESCRIPTION:

For some reason, the data type for the specified field cannot be determined. This may mean that the data dictionary is corrupted.

TEXT:

The data type for Field #|FIELD| in File #|FILE| cannot be determined.

PARAMETERS:

- 'FIELD' means Field number.
- 'FILE' means File number.

Error 520

DESCRIPTION:

An incorrect kind of field is being processed. For example, filing is being attempted for a computed field or validation for a word processing field.

TEXT:

A |1| field cannot be processed by this utility.

PARAMETERS:

'1' means Data type or other field characteristic (e.g., .001, DINUMed).

- 'FILE' means File #.
- 'FIELD' means Field #.

Error 525

DESCRIPTION:

It is indicated that a subfile is involved (for example, by choosing a multiple field's field number), but no fields from the subfile are chosen.

TEXT:

No fields are specified for subfile #|FILE|.

PARAMETERS:

FILE means Subfile #.

Error 537

DESCRIPTION:

This error means that a certain field in a certain file has a data type of pointer, but something is wrong with the rest of the DD information needed to make that pointer work. For example, perhaps the number of the pointed-to file, which should follow the P in the second ^-piece of the field descriptor node, is missing.

Another problem would be if the global root of the pointed to file were missing from the field's definition; that should be found in the third ^-piece of the field descriptor.

TEXT:

Field #|FIELD| in File #|FILE| has a corrupted pointer definition.

PARAMETERS:

- 'FILE' means File #.
- 'FIELD' means Field #.

Error 601

DESCRIPTION:

The entry identified by FILE and IENS does not exist in the database.

TEXT:

The entry does not exist.

PARAMETERS:

- 'FILE' means File or subfile #. (external only)
- 'IENS' means IEN string (external only)

Error 602

DESCRIPTION:

There is a -9 node for the entry; therefore, the entry cannot be accessed.

TEXT:

The entry is not available for editing.

PARAMETERS:

- 'FILE' means File or subfile #. (external only)
- 'IENS' means IEN string. (external only)

Error 603

DESCRIPTION:

A specific entry in a specific file lacks a value for a required field. This error message returns the name of the field which is missing.

TEXT:

Entry #|1| in File #|FILE| lacks the required Field #|FIELD|.

PARAMETERS:

- '1' means Entry #.
- 'FILE' means File #.
- 'FIELD' means Field #.

Error 630

DESCRIPTION:

The database is corrupted. The value for a specific field in one entry should be a certain data type, but it is not.

TEXT:

In Entry # |1| of File #|FILE|, the value '|2|' for Field #|FIELD| is not a valid '|3|'.

PARAMETERS:

- '1' means Entry #.
- '2' means Field Value.
- '3' means Data Type.
- 'FILE' means File #.
- 'FIELD' means Field #.

Error 648

DESCRIPTION:

The database is corrupted. In a specific variable pointer field of a certain entry, the field's value points to a file that either does not exist or that lacks a Header Node.

TEXT:

In Entry #|1| of File #|FILE|, the value '|2|' for Field #|FIELD| points to a file that does not exist or lacks a Header Node.

PARAMETERS:

- '1' means Entry #.
- '2' means Field Value.
- 'FILE' means File #.
- 'FIELD' means Field #.

Error 701

DESCRIPTION:

The value is invalid. Possible causes include: value did not pass INPUT transform, value for a pointer or variable pointer field cannot be found in the pointed-to file, a screen was not passed.

TEXT:

The value '|3|' for field |1| in file |2| is not valid.

PARAMETERS:

- '1' means Field name.
- '2' means File name.
- '3' means Value that was found to be invalid.
- 'FIELD' means Field number. (external only)
- 'FILE' means File number. (external only)
- 'IENS' means IEN string identifying entry with invalid value. (external only, sometimes returned)

Error 703

DESCRIPTION:

The value passed cannot be found in the indicated file using $$FIND1^DIC.

TEXT:

The value '|1|' cannot be found in file #|FILE|.

PARAMETERS:

- 'FILE' means File #.
- 'IENS' means IEN String.
- '1' means Lookup Value.

Error 710

DESCRIPTION:

The data dictionary specifies that the field is uneditable. Data already exists in the field. It cannot be changed.

TEXT:

Data in Field #|FIELD| in File #|FILE| cannot be edited.

PARAMETERS:

- 'FIELD' means Field number.
- 'FILE' means File number.

Error 712

DESCRIPTION:

The value of a field cannot be deleted either because it is a required field, because it is the .01 of a file, or because the test in the "DEL" node was not passed.

TEXT:

The value of field |1| in file |2| cannot be deleted.

PARAMETERS:

- '1' means Field name.
- '2' means File name.
- 'FIELD' means Field number. (external only)
- 'FILE' means File number. (external only)

Error 714

DESCRIPTION:

The field uses $Piece storage and the data contains an '^'. The data cannot be filed.

TEXT:

Data for Field |1| in File |2| contains an '^'.

PARAMETERS:

- '1' means Field name.
- '2' means File name.
- 'FILE' means File number. (external only)
- 'FIELD' means Field number. (external only)

Error 716

DESCRIPTION:

Data being filed is too long for the field. Specifically, this occurs when data of the wrong length is being filed in a $Extract (Em,n) field.

TEXT:

Data for field |1| in file |2| is too long.

PARAMETERS:

- '1' means Field name.
- '2' means File name.
- 'FIELD' means Field number. (external only)
- 'FILE' means File number. (external only)

Error 720

DESCRIPTION:

The lookup for a pointer fails. This is an error only when LAYGO is not allowed.

TEXT:

The value cannot be found in the pointed-to file.

PARAMETERS:

- 'FILE' means File number—the number of the file in which the pointer field exists.
- 'FIELD' means Field number of the pointer field.

Error 726

DESCRIPTION:

There is an attempt to take an action with word processing data, but the specified field is not a word processing field.

TEXT:

Field #|FIELD| in File #|FILE| is not a word processing field.

PARAMETERS:

- 'FIELD' means Field number.
- 'FILE' means File number.

Error 730

DESCRIPTION:

Based on how the data type is defined by a specific field in a specific file, the passed value is not valid.

TEXT:

The value '|1|' is not a valid |2| according to the definition in Field #|FIELD| of File #|FILE|.

PARAMETERS:

- '1' means Passed Value.
- '2' means Data Type.
- 'FIELD' means Field #.
- 'FILE' means File #.

Error 740

DESCRIPTION:

When one or more fields are declared as a key for a file, there cannot be duplicate values in those field(s) for entries in the file. The values being passed for validation, when combined with values for unchanging fields in the entry if necessary, create a duplicate key. The changes destroy the integrity of the key. Therefore, they are invalid.

TEXT:

New values are invalid because they create a duplicate Key '|1|' for the |2| file.

PARAMETERS:

- '1' means Name of Key.
- '2' means Name of affected file.

Error 742

DESCRIPTION:

Every field in a key must have a value. The incoming data cannot delete the value for any field in a key.

TEXT:

The value of field |1| in the |2| file cannot be deleted because that field is part of the '|3|' key.

PARAMETERS:

- '1' means Field name
- '2' means File name
- '3' means Key name
- 'FILE' meansFile number
- 'FIELD' means Field number

Error 744

DESCRIPTION:

Every field that is in a key must have a value. No value for this field exists.

TEXT:

Field |1| is part of Key '|2|', but the field has not been assigned a value.

PARAMETERS:

- '1' means Field name.
- '2' means Key name.
- 'FIELD' means Field number.
- 'FILE' means File number.

Error 746

DESCRIPTION:

A lookup node is present in the FDA, but no Primary Key fields are provided.

The K flag was used, but no primary key fields were provided in the FDA for Finding and LAYGO Finding nodes.

TEXT:

No fields in Primary Key '|1|' have been provided in the FDA to look up '|IENS|' in the |2| file.

PARAMETERS:

- '1' means Key name
- '2' means File name
- "IENS' means IEN string of lookup node. (external only)
- 'KEY' means Key number. (external only)
- 'FILE' means File number. (external only)

Error 810

DESCRIPTION:

A %ZOSF node required to perform a function does not exist. The VA FileMan Programmer's Manual contains a complete list of %ZOSF nodes.

TEXT:

A necessary %ZOSF node does not exist on your system.

PARAMETERS:

None

Error 820

DESCRIPTION:

The ZSAVE CODE field (#2619) in the MUMPS OPERATING SYSTEM file (#.7) is empty for the operating system being used. It is impossible to perform functions such as compiling templates or cross-references.

TEXT:

There is no way to save routines on the system.

PARAMETERS:

None

Error 840

DESCRIPTION:

The Terminal Type file does not have an entry that matches IOST(0).

TEXT:

Terminal type '|1|' cannot be found in the Terminal Type file.

PARAMETERS:

'1' means Terminal type as identified by IOST(0).

Error 842

DESCRIPTION:

The field in the Terminal Type field that contains the specified characteristic of the terminal is null.

TEXT:

|1| cannot be found for Terminal Type |2|.

PARAMETERS:

- '1' means Terminal Type characteristic.
- '2' means Terminal type.

Error 845

DESCRIPTION:

A %ZIS call with IOP set to "HOME" returns POP.

TEXT:

The characteristics for the HOME device cannot be obtained.

PARAMETERS:

None

Error 1300

DESCRIPTION:

The entry encountered an error during subfile filing.

TEXT:

The entry encountered an error during subfile filing.

PARAMETERS:

'IEN' means Entry number

Error 1500

DESCRIPTION:

Error given for unsuccessful lookup of SEARCH template in BY(0) input variable.

TEXT:

SEARCH template |1| in BY(0) variable cannot be found, is for the wrong file, or has no list of search results.

PARAMETERS:

'1' means Name of SEARCH template in input variable BY(0).

Error 1501

DESCRIPTION:

Error message shown to user when no code was generated during compilation of SORT TEMPLATES.

TEXT:

There is no code to save for this compiled Sort Template routine.

PARAMETERS:

None

Error 1502

DESCRIPTION:

Error message notifying the user that there are no more available routine numbers for compiled Sort Template routines. This should never happen, since routine numbers are re-used.

TEXT:

All available routine numbers for compilation are in use.

IRM needs to run ENRLS^DIOZ() to release the routine numbers.

PARAMETERS:

None

Error 1503

DESCRIPTION:

Warn user to shorten compiled cross-reference routine name.

TEXT:

Routine name is too long. Compilation has been aborted.

PARAMETERS:

None

Error 1504

DESCRIPTION:

If doing Transfer/Merge of a single record from one file to another and the .01 field names do not match, we cannot do the transfer/merge.

TEXT:

No matching .01 field names found. Transfer/Merge cannot be done.

PARAMETERS:

None

Error 1610

DESCRIPTION:

A question mark or, in the case of a variable pointer field, a <something>.? was passed to the Validator. The Validator does not process help requests.

TEXT:

Help is being requested from the Validator utility.

PARAMETERS:

- 'FILE' means File number.
- 'FIELD' means Field number.

Error 1700

DESCRIPTION:

Generic message for Silent DIFROM

TEXT:

Error: |1|.

PARAMETERS:

'1' means Generic message

Error 1701

DESCRIPTION:

Transport structure does not contain SPECIFIC ELEMENT.

TEXT:

Transport structure does not contain |1|.

PARAMETERS:

'1' means Describes missing element in transport structure.

Error 1805

DESCRIPTION:

For some reason a record or a field in a record could not be filed. The cause of the error should be present in another message.

TEXT:

An error occurred during the actual filing of data into the FileMan database.

PARAMETERS:

None

Error 1810

DESCRIPTION:

The attempt to move data from a host file into the M environment failed. A possible cause is that the host file does not exist in the path specified.

TEXT:

The data from host file '|1|' could not be moved into a FileMan file.

PARAMETERS:

'1' means Host file name.

Error 1812

DESCRIPTION:

A host file was located; however, no data was present in it. This error will also occur if the only "data" is the designation of file and fields with no actual data present to file.

TEXT:

The host file, |1|, contains no data to import.

PARAMETERS:

1 means Host file name.

Error 1820

DESCRIPTION:

The foreign format name that was passed could not be found in the Foreign Format file.

TEXT:

There is no Foreign Format named '|1|'.

PARAMETERS:

'1' means Foreign format.

Error 1821

DESCRIPTION:

The format of the imported data must either be delimited by a specified character or be fixed length. The format being specified is neither.

TEXT:

If no record delimeter is specified, the foreign format must be fixed length.

PARAMETERS:

None

Error 1822

DESCRIPTION:

For a fixed length import, the length data for a field is impossible. For example, the length is zero or the start position is larger than the end position.

TEXT:

The length of a field is incorrectly specified.

PARAMETERS:

None

Error 1833

DESCRIPTION:

The F flag for the Import call means that the file and field information is in the host file. However, the file and/or fields parameter contained data. This conflicts with the F flag.

TEXT:

The F flag conflicts with the File or Fields parameter.

PARAMETERS:

None

Error 1850

DESCRIPTION:

The device for printing the Import report was not properly specified. This could be caused either by a user's response or by the device specifications passed to the FILE^DDMP call. The problem could involve either device or queuing instructions.

TEXT:

There is an error in device selection or queuing setup.

PARAMETERS:

None

Error 1870

DESCRIPTION:

A requested IMPORT template does not exist in the IMPORT template file for the file being imported into.

TEXT:

IMPORT template |1| does not exist for File #|FILE|.

PARAMETERS:

- '1' means Template Name.
- 'FILE' means File number.

Error 3021

DESCRIPTION:

A lookup in to the Form file for the given form failed.

TEXT:

Form |1| does not exist in the Form file, or DDSFILE is not the Primary File of the form.

PARAMETERS:

'1' means Form name.

Error 3022

DESCRIPTION:

There are no pages defined in the Page multiple of the given form.

TEXT:

Form |1| contains no pages.

PARAMETERS:

'1' means Form name.

Error 3023

DESCRIPTION:

The given page was not found on the form.

TEXT:

The form does not contain a page |1|.

PARAMETERS:

'1' means Page name or number.

Error 8090

DESCRIPTION:

Used in displaying an error message when the lookup value X does not pass the Pre-lookup transform code (^DD(File#,.01,7.5)node) during ^DIC or Finder lookups.

TEXT:

Pre-lookup transform (7.5 node)

PARAMETERS:

None

Error 8095

DESCRIPTION:

In calls to the Finder, IX^DIC, or MIX^DIC, if either the first index passed or the default index is a compound index, then only one index can be passed, so neither DIC(0) nor flags can contain "M".

TEXT:

First lookup index is compound, so "M"ultiple index lookups not allowed.

PARAMETERS:

None

Index

$

$$CREF^DILF( ): Root Converter (Open to Closed Format), 2-152
$$EXTERNAL^DILFD( ): Converter to External, 2-168
$$EZBLD^DIALOG( ): DIALOG Extractor (Single Line), 2-31
$$FIND1^DIC( ): Finder (Single Record), 2-65
$$FLDNUM^DILFD( ): Field Number Retriever, 2-174
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
$$GET1^DID( ): Attribute Retriever, 2-114
$$GET1^DIQ( ): Single Data Retriever, 2-182
$$HTML^DILF( ): HTML Encoder/Decoder, 2-161
$$IENS^DILF( ): IENS Creator, 2-162
$$KEYVAL^DIE( ): Key Validator, 2-129
$$OREF^DILF( ): Root Converter (Closed to Open Format), 2-163
$$ROOT^DILFD( ): File Root Resolver, 2-178
$$ROUSIZE^DILF, 1-77
$$TEST^DDBRT, 6-13
$$VALUE1^DILF( ): FDA Value Retriever (Single), 2-164
$$VFIELD^DILFD( ): Field Verifier, 2-180
$$VFILE^DILFD( ): File Verifier, 2-181

%

%DT, 1-144
%XY^%RCR, 1-164

^

^%DT, 1-145
^%DTC, 1-153
^DDGF, 3-39
^DDS, 5-1
^DI
  Programmer Access, 10-1
^DIAC, 1-12
^DIC, 1-14
^DIE, 1-47
^DIEZ, 1-59
^DIFG: Installer, 9-1
^DIK, 1-61
^DIKZ, 1-75
^DIM, 1-78
^DIOZ, 1-80
^DIPT, 1-101
^DIPZ, 1-103
^DIR, 1-113
^DIWF, 1-135
^DIWP, 1-141
^DIWW, 1-143

A

Actions
  Post-Selection Action
    Advanced File Definition, 12-10
Adding (ScreenMan Form Editor)
  Blocks, 4-10
  Fields, 4-11
  Pages:, 4-9
Adding, Selecting, and Editing
  ScreenMan Form Editor, 4-4
Advanced File Definition, 12-1
  Assigning a Location for Fields Stored within a Global, 12-3
  Assigning Sub-Dictionary Numbers, 12-5
  Audit Condition, 12-10

Computed Expressions, 12-6
Editing a Cross-reference, 12-11
Executable Help, 12-11
Field Global Storage, 12-3
File Global Storage, 12-1
INPUT Transform, 12-7
INPUT Transforms and the Verify Fields Option, 12-8
Introduction, 12-1
MUMPS Data Type, 12-6
OUTPUT Transform, 12-9
Post-Selection Action, 12-10
Screened Pointers and Set of Codes, 12-6
Special Lookup Programs, 12-9
Storing Data by Position within a Node, 12-4
Storing Data in a Global other than ^DIZ, 12-1
APIs
$$CREF^DILF( ), 2-152
$$EXTERNAL^DILFD( ), 2-168
$$EZBLD^DIALOG( ), 2-31
$$FIND1^DIC( ), 2-65
$$FLDNUM^DILFD( ), 2-174
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
$$GET1^DID( ), 2-114
$$GET1^DIQ( ), 2-182
$$HTML^DILF( ), 2-161
$$IENS^DILF( ), 2-162
$$KEYVAL^DIE( ), 2-129
$$OREF^DILF( ), 2-163
$$ROOT^DILFD( ), 2-178
$$ROUSIZE^DILF, 1-77
$$TEST^DDBRT, 6-13
$$VALUE1^DILF( ), 2-164
$$VFIELD^DILFD( ), 2-180
$$VFILE^DILFD( ), 2-181
%XY^%RCR, 1-164
^%DT, 1-145
^%DTC, 1-153
^DDS, 5-1
^DIAC, 1-12
^DIC, 1-14
^DIE, 1-47
^DIEZ, 1-59
^DIFG, 9-1
^DIK, 1-61
^DIKZ, 1-75
^DIM, 1-78
^DIOZ, 1-80
^DIPT, 1-101
^DIPZ, 1-103
^DIR, 1-113
^DIWF, 1-135
^DIWP, 1-141
^DIWW, 1-143
BLD^DIALOG( ), 2-24
BROWSE^DDBR, 6-3
Browser, 6-1
C^%DTC, 1-154
CHK^DIE( ), 2-117
Classic VA FileMan, 1-1
CLEAN^DILF, 2-151
CLOSE^DDBRZIS, 6-14
COMMA^%DTC, 1-155
D^DIQ, 1-105
DA^DILF( ), 2-153
Database Server (DBS) Calls, 2-1
DD^%DT, 1-151
DELIX^DDMOD, 2-17
DELIXN^DDMOD, 2-21
DIBT^DIPT, 1-102
DO^DIC1, 1-35
DOCLIST^DDBR, 6-10
DQ^DICQ, 1-44
DT^DICRW, 1-45
DT^DILF( ), 2-154
DT^DIO2, 1-79
DT^DIQ, 1-106
DW^%DTC, 1-157
EN^DDBR, 6-1
EN^DDIOL, 1-8
EN^DIAXU, 8-1
EN^DIB, 1-13
EN^DID, 1-46
EN^DIEZ, 1-60

EN^DIFGG, 9-5
EN^DIK, 1-63
EN^DIKZ, 1-76
EN^DIPZ, 1-104
EN^DIQ, 1-107
EN^DIQ1, 1-109
EN^DIS, 1-129
EN^DIU2, 1-130
EN^DIWE, 1-132
EN1^DIK, 1-65
EN1^DIP, 1-81
EN1^DIWF, 1-137
EN2^DIWF, 1-139
ENALL^DIK, 1-67
EXPORT^DDMP, 7-11
EXTRACT^DIAXU, 8-4
FDA^DILF( ), 2-158
FIELD^DID( ), 2-105
FIELDLST^DID( ), 2-108
FILE^DDMP, 7-2
FILE^DICN, 1-40
FILE^DID( ), 2-109
FILE^DIE( ), 2-120
Filegrams, 9-1
FILELST^DID( ), 2-112
FIND^DIC( ), 2-38
GETS^DIQ( ), 2-186
H^%DTC, 1-158
HELP^%DTC, 1-159
HELP^DIE( ), 2-125
HLP^DDSUTL( ), 5-14
Import Tool, 7-1
IX^DIC, 1-32
IX^DIK, 1-69
IX1^DIK, 1-71
IXALL^DIK, 1-73
LIST^DIC( ), 2-81
MIX^DIC1, 1-36
MSG^DDSUTL( ), 5-15
MSG^DIALOG( ), 2-33
NOW^%DTC, 1-160
OPEN^DDBRZIS, 6-15
Other, III-1
POST^DDBRZIS, 6-16
PRD^DILFD( ), 2-175
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12
RECALL^DILFD( ), 2-176
REFRESH^DDSUTL( ), 5-16
REQ^DDSUTL( ), 5-17
S^%DTC, 1-161
ScreenMan, 5-1
ScreenMan Introduction, 5-1
UNED^DDSUTL( ), 5-18
UPDATE^DIE( ), 2-131
VAL^DIE( ), 2-140
VALS^DIE( ), 2-145
VALUES^DILF( ), 2-166
WAIT^DICD, 1-39
WP^DDBR, 6-7
WP^DIE( ), 2-149
X ^DD("DD"), 1-6
Y^DIQ, 1-108
YMD^%DTC, 1-162
YN^DICN, 1-43
YX^%DTC, 1-163
Appendix A—VA FileMan Error
  Codes, 1
  Introduction, 1
Array and Variable Clean-up
  CLEAN^DILF, 2-151
ASSIGN A VERSION NUMBER, 16-5
Assigning
  A Location for Fields Stored within
    a Global
    Advanced File Definition, 12-3
  Sub-Dictionary Numbers
    Advanced File Definition, 12-5
Attribute Dictionary
  Global File Structure, 11-4
Attribute Retriever
  $$GET1^DID( ), 2-114
Audit Condition
  Advanced File Definition, 12-10

B

B Cross-reference
  Checking, 16-24

Backward Pointers
Relational Navigation
ScreenMan Forms, 3-12
BLD^DIALOG( ): DIALOG Extractor, 2-24
Block Properties (ScreenMan Forms), 3-25, 3-28
Coordinate, 3-26
DD Number, 3-27
Disable Navigation, 3-27
Name, 3-25, 3-27
Order, 3-25
Pointer Link, 3-26
Pre Action and Post Action, 3-26, 3-27
Replication, Index, Initial Position, Disallow LAYGO, Field for Selection, 3-26
Stored in the
BLOCK File, 3-27
FORM File, 3-25
That Apply Only to Repeating Blocks, 3-6
Type of Block, 3-25
Block Viewer Screen
ScreenMan Form Editor, 4-7
Blocks
Adding Blocks with ScreenMan Form Editor, 4-10
Header Blocks with ScreenMan Form Editor, 4-10
Branching Logic, Pre Action, Post Action, and Post Action on Change
ScreenMan Forms Field Properties, 3-33
BROWSE^DDBR, 6-3
Browser
APIs, 6-1
Calls
$$TEST^DDBRT, 6-13
BROWSE^DDBR, 6-3
CLOSE^DDBRZIS, 6-14
DOCLIST^DDBR, 6-10
EN^DDBR, 6-1
OPEN^DDBRZIS, 6-15
POST^DDBRZIS, 6-16
WP^DDBR, 6-7
Builds Routines Containing
Data Dictionaries
DIFROM, 16-15
Data Values
DIFROM, 16-16
Security Access Codes
DIFROM, 16-16
BULLETINS, 16-28

C

C^%DTC, 1-154
Callable Routines
ScreenMan Forms, 3-39
Caption and Data Coordinates
ScreenMan Forms Field Properties, 3-31
Caption, Executable Caption, and Suppress Colon After Caption
ScreenMan Forms Field Properties, 3-29
Check of Version Number
DIFROM, Running an INIT, 16-20
Checking the B Cross-reference or Zero Node, 16-24
CHK^DIE( ): Data Checker, 2-117
Choosing Another Form
ScreenMan Form Editor, 4-21
Classic Calls
$$ROUSIZE^DILF, 1-77
%XY^%RCR, 1-164
^%DT, 1-145
^%DTC, 1-153
^DIAC, 1-12
^DIC, 1-14
^DIE, 1-47
^DIEZ, 1-59
^DIK, 1-61
^DIKZ, 1-75
^DIM, 1-78
^DIOZ, 1-80
^DIPT, 1-101
^DIPZ, 1-103

^DIR, 1-113
^DIWP, 1-141
^DIWW, 1-143
C^%DTC, 1-154
COMMA^%DTC, 1-155
D^DIQ, 1-105
DD^%DT, 1-151
DIBT^DIPT, 1-102
DO^DIC1, 1-35
DQ^DICQ, 1-44
DT^DICRW, 1-45
DT^DIO2, 1-79
DT^DIQ, 1-106
DW^%DTC, 1-157
EN^DDIOL, 1-8
EN^DIB, 1-13
EN^DID, 1-46
EN^DIEZ, 1-60
EN^DIK, 1-63
EN^DIKZ, 1-76
EN^DIPZ, 1-104
EN^DIQ, 1-107
EN^DIQ1, 1-109
EN^DIS, 1-129
EN^DIU2, 1-130
EN^DIWE, 1-132
EN1^DIK, 1-65
EN1^DIP, 1-81
EN1^DIWF, 1-137
EN2^DIWF, 1-139
ENALL^DIK, 1-67
FILE^DICN, 1-40
H^%DTC, 1-158
HELP^%DTC, 1-159
IX^DIC, 1-32
IX^DIK, 1-69
IX1^DIK, 1-71
IXALL^DIK, 1-73
Listed Alphabetically, 1-5
MIX^DIC1, 1-36
NOW^%DTC, 1-160
S^%DTC, 1-161
WAIT^DICD, 1-39
X ^DD("DD"), 1-6
Y^DIQ, 1-108
YMD^%DTC, 1-162
YN^DICN, 1-43
YX^%DTC, 1-163
Classic Calls (Alphabetic Order), 1-5
Classic Calls By Category, 1-3
Classic Calls Cross-referenced By Category, 1-3
Classic VA FileMan API, 1-1
CLEAN^DILF: Array and Variable Clean-up, 2-151
Cleaning Up the Output Arrays
DBS Calls, 2-11
CLONE^DDS, 3-40
CLOSE^DDBRZIS, 6-14
COMMA^%DTC, 1-155
Command Summary
ScreenMan Form Editor, 4-2
Completes Building Routines of Package Components
DIFROM, 16-17
Completes the Code that Runs the Init
DIFROM, 16-18
Computed Expressions
Advanced File Definition, 12-6
Computed Fields
ScreenMan Forms, 3-13
Contents of Arrays
DBS Calls, 2-8
DIERR Array, 2-9
DIHELP Array, 2-8
DIMSG Array, 2-9
Converter to External
$$EXTERNAL^DILFD( ), 2-168
Coordinate and Lower Right Coordinate
ScreenMan Forms Page Property, 3-22
Creating
DIALOG File Entries, 14-2
LANGUAGE File Entries, 14-7
Non-English Text in the DIALOG File, 14-4
VA FileMan Functions, 15-1
Function File Entries, 15-1

Introduction, 15-1
Cross-references
^DIKZ, 1-75
EN^DIK, 1-63
EN^DIKZ, 1-76
EN1^DIK, 1-65
ENALL^DIK, 1-67
Global File Structure, 11-6, 11-17
IX^DIK, 1-69
IX1^DIK, 1-71
IXALL^DIK, 1-73
Trigger, 13-1

D

D^DIQ, 1-105
DA( ) Creator
DA^DILF( ), 2-153
DA^DILF( ): DA( ) Creator, 2-153
Data Checker
CHK^DIE( ), 2-117
DATA COMES WITH FILE, 16-6
Data Dictionary Audit
Global File Structure, 11-5
Data Dictionary DBS Calls
$$FLDNUM^DILFD( ), 2-174
$$GET1^DID( ), 2-114
$$ROOT^DILFD( ), 2-178
$$VFIELD^DILFD( ), 2-180
$$VFILE^DILFD( ), 2-181
FIELD^DID( ), 2-105
FIELDLST^DID( ), 2-108
FILE^DID( ), 2-109
FILELST^DID( ), 2-112
PRD^DILFD( ), 2-175
Data Dictionary Modification DBS Calls
DELIX^DDMOD, 2-17
DELIXN^DDMOD, 2-21
Data Editing DBS Calls
$$KEYVAL^DIE( ), 2-129
CHK^DIE( ), 2-117
FILE^DIE( ), 2-120
HELP^DIE( ), 2-125
RECALL^DILFD( ), 2-176
UPDATE^DIE( ), 2-131
VAL^DIE( ), 2-140
VALS^DIE( ), 2-145
WP^DIE( ), 2-149
Data Export
EXPORT^DDMP, 7-11
Data Filing
ScreenMan Forms, 3-19
Data Import
FILE^DDMP, 7-2
Data Length
ScreenMan Forms Field Properties, 3-30
Data Retrieval
EN^DIQ1, 1-109
Data Retrieval DBS Calls
$$GET1^DIQ( ), 2-182
GETS^DIQ( ), 2-186
Data Retriever
GETS^DIQ( ), 2-186
Data Storage Conventions
Global File Structure, 11-1
Data Types
MUMPS Data Type
Advanced File Definition, 12-6
Set of CodesData Type
Advanced File Definition, 12-6
Data Validation
ScreenMan Forms Field Properties, 3-33
ScreenMan Forms Form Properties, 3-20
Database Server (DBS) API, 2-1
Database Server (DBS) Calls (Alphabetic Order), 2-16
DataBase Server Calls Cross-referenced by Category, 2-15
Date Converter
DT^DILF( ), 2-154
Date/Time
%DT, 1-144
Date/Time Formats, Introduction
%DT, 1-144
Date/Time Utilities
%DT, 1-144

^%DT, 1-145
^%DTC, 1-153
C^%DTC, 1-154
DD^%DT, 1-151
DT^DIO2, 1-79
DW^%DTC, 1-157
H^%DTC, 1-158
NOW^%DTC, 1-160
S^%DTC, 1-161
X ^DD("DD"), 1-6
YMD^%DTC, 1-162
YX^%DTC, 1-163
DBS Calls, 2-1
$$CREF^DILF( ), 2-152
$$EXTERNAL^DILFD( ), 2-168
$$EZBLD^DIALOG( ), 2-31
$$FIND1^DIC( ), 2-65
$$FLDNUM^DILFD( ), 2-174
$$GET1^DID( ), 2-114
$$GET1^DIQ( ), 2-182
$$HTML^DILF( ), 2-161
$$IENS^DILF( ), 2-162
$$KEYVAL^DIE( ), 2-129
$$OREF^DILF( ), 2-163
$$ROOT^DILFD( ), 2-178
$$VALUE1^DILF( ), 2-164
$$VFIELD^DILFD( ), 2-180
$$VFILE^DILFD( ), 2-181
BLD^DIALOG( ), 2-24
CHK^DIE( ), 2-117
CLEAN^DILF, 2-151
Cleaning Up the Output Arrays, 2-11
Contents of Arrays, 2-8
DIERR Array, 2-9
DIHELP Array, 2-8
DIMSG Array, 2-9
DA^DILF( ), 2-153
DELIX^DDMOD, 2-17
DELIXN^DDMOD, 2-21
Documentation Conventions, 2-6
DT^DILF( ), 2-154
Example of Call to VA FileMan DBS, 2-12
FDA: Format of Data Passed to and from VA FileMan, 2-5
FDA^DILF( ), 2-158
FIELD^DID( ), 2-105
FIELDLST^DID( ), 2-108
FILE^DID( ), 2-109
FILE^DIE( ), 2-120
FILELST^DID( ), 2-112
FIND^DIC( ), 2-38
Format and Conventions, 2-2
GETS^DIQ( ), 2-186
HELP^DIE( ), 2-125
How Information Is Returned, 2-7
How the Database Server (DBS) communicates, 2-7
How to Use, 2-2
IENS: To Identify Entries and Subentries, 2-3
Introduction, 2-1
LIST^DIC( ), 2-81
Listed Alphabetically, 2-16
Listed by Category, 2-16
MSG^DIALOG( ), 2-33
Obtaining Formatted Text From The Arrays, 2-11
Overview, 2-7
PRD^DILFD( ), 2-175
RECALL^DILFD( ), 2-176
UPDATE^DIE( ), 2-131
VAL^DIE( ), 2-140
VALS^DIE( ), 2-145
VALUES^DILF( ), 2-166
WP^DIE( ), 2-149
DBS Calls by Category, 2-15
DD Deletion
EN^DIU2, 1-130
DD Field List Retriever
FIELDLST^DID( ), 2-108
DD Field Retriever
FIELD^DID( ), 2-105
DD File List Retriever
FILELST^DID( ), 2-112
DD File Retriever
FILE^DID( ), 2-109

DD Number
ScreenMan Forms Block Properties, 3-27
DD^%DT, 1-151
DDBR, 6-1
DDS Variable, 5-3
DDSBR Variable
ScreenMan Forms, 3-17
DDSSTACK Variable
ScreenMan Forms, 3-18
Default and Executable Default
ScreenMan Forms Field Properties, 3-29
DEFAULT PROTOCOL, 16-28
Delete a Form ScreenMan Forms Option, 3-36
Deleting Screen Elements
ScreenMan Form Editor, 4-22
DELIX^DDMOD: Traditional Cross-reference Deleter, 2-17
DELIXN^DDMOD\
New-Style Index Deleter, 2-21
Determining Install Status of
DDs and Data
DIFROM, Running an INIT, 16-21
Other Package Components
DIFROM, Running an INIT, 16-23
Security Codes
DIFROM, Running an INIT, 16-22
Developer Tools, IV-1
DIALOG Extractor
BLD^DIALOG( ), 2-24
DIALOG Extractor (Single Line)
$$EZBLD^DIALOG( ), 2-31
DIALOG File, 14-1
Creating DIALOG File Entries, 14-2
Creating Non-English Text in the DIALOG File, 14-4
Internationalization, 14-4
Introduction, 14-1
LANGUAGE File, 14-6
Creating LANGUAGE File Entries, 14-7
Introduction, 14-6
Use of the LANGUAGE File, 14-6
Role of the VA FileMan DIALOG File in Internationalization, 14-4
Use of the DIALOG File, 14-1
Use of the DIALOG File in Internationalization, 14-4
User Messages, 14-1
DIBT^DIPT, 1-102
Dictionary of Files, 11-1
DIERR Array, 2-9
DIFQ Variables
DIFROM, 16-20
DIFROM, 16-1
Exporting Data, 16-2
Importing Data, 16-19
Introduction, 16-1
Order Entry and DIFROM, 16-9
PACKAGE File and DIFROM, 16-2
ENVIRONMENT CHECK ROUTINE, 16-4
EXCLUDED NAME SPACE, 16-3
FILE, 16-4
NAME, 16-3
Other PACKAGE File Fields, 16-7
POST-INITIALIZATION ROUTINE, 16-4
PREFIX, 16-3
PRE-INIT AFTER USER COMMIT, 16-4
Template Multiples, 16-3
Preparing To Run DIFROM, 16-2
Running an INIT
Check of Version Number, 16-20
Determining Install Status of
DDs and Data, 16-21
Other Package Components, 16-23
Security Codes, 16-22
DIFQ Variables, 16-20
General Processing, 16-27
Installing
Data, 16-24
Data Dictionaries, 16-23
Other Package Components, 16-26

Pre-init After User Commit Routine, 16-23
Preliminary Steps, 16-19
Recording the Install on the Target System, 16-30
Reindexing the Files, 16-26
Running Environment Check Routine, 16-20
Running the Post-Initialization Routine, 16-29
Special Processing, 16-28
Starting the Update, 16-23
Running an INIT (Steps), 16-19
Running DIFROM
Builds Routines Containing
Data Dictionaries, 16-15
Data Values, 16-16
Security Access Codes, 16-16
Completes Building Routines of Package Components, 16-17
Completes the Code that Runs the Init, 16-18
Entering Current Version Information, 16-13
Exporting File Security, 16-14
Gathers Miscellaneous Package Components, 16-14
Gathers Templates and Forms, 16-17
Identifying the Init Routines, 16-12
Including Other Package Components, 16-14
Including Templates (No Package File Entry), 16-14
Package Identification, 16-12
Preliminary Validations, 16-12
Specifications for Exported Files, 16-13
Specifying Routine Size, 16-14
Starting DIFROM, 16-12
Running DIFROM (Steps), 16-11
DIHELP Array, 2-8
DIMSG Array, 2-9
Disable Editing and Disallow LAYGO
ScreenMan Forms Field Properties, 3-32
Disable Navigation
ScreenMan Forms Block Properties, 3-27
Display
D^DIQ, 1-105
Display Group
ScreenMan Forms Field Properties, 3-32
DISPLAY GROUP DEFAULT, 16-28
Displaying Multiples in Repeating Blocks
ScreenMan Forms, 3-5
Distribution Package
Global File Structure, 11-7
DO^DIC1, 1-35
DOCLIST^DDBR, 6-10
Documentation Conventions
DBS Calls, 2-6
DQ^DICQ, 1-44
DT^DICRW, 1-45
DT^DILF( ): Date Converter, 2-154
DT^DIO2, 1-79
DT^DIQ, 1-106
DW^%DTC, 1-157

E

Edit/Create a Form ScreenMan Option, 3-35
Editing (ScreenMan Form Editor)
"Pop-Up" Page Coordinates, 4-18
Block Properties, 4-17
Field Captions and Data Length, 4-15
Field Properties, 4-14
Form Properties, 4-20
Page Properties, 4-18
Editing a Cross-reference
Advanced File Definition, 12-11
Editing, Adding, and Selecting
ScreenMan Form Editor, 4-4
Editors
ScreenMan
Form Editor, 4-1
Elements

Moving Screen Elements with 1504, 31
ScreenMan Form Editor, 4-12 1610, 31
Selecting Screen Elements with 1700, 32
ScreenMan Form Editor, 4-11 1701, 32
EN^DDBR, 6-1 1805, 33
EN^DDIOL, 1-8 1810, 33
EN^DIAXU: Extract Data, 8-1 1812, 33
EN^DIB, 1-13 1820, 34
EN^DID, 1-46 1821, 34
EN^DIEZ, 1-60 1822, 34
EN^DIFGG: Generator, 9-5 1833, 35
EN^DIK, 1-63 1850, 35
EN^DIKZ, 1-76 1870, 36
EN^DIPZ, 1-104 200, 3
EN^DIQ, 1-107 201, 4
EN^DIQ1, 1-109 202, 4
EN^DIS, 1-129 203, 4
EN^DIU2, 1-130 204, 5
EN^DIWE, 1-132 205, 5
EN1^DIK, 1-65 206, 5
EN1^DIP, 1-81 207, 6
EN1^DIWF, 1-137 299, 6
EN2^DIWF, 1-139 301, 7
ENALL^DIK, 1-67 302, 7
Entering Current Version Information 3021, 36
DIFROM, 16-13 3022, 36
Entry Editing 3023, 37
^DIE, 1-47 304, 7
^DIK, 1-61 305, 8
EN^DIB, 1-13 306, 8
EN^DIQ1, 1-109 307, 8
EN^DIWE, 1-132 308, 9
ENVIRONMENT CHECK ROUTINE 309, 9
PACKAGE File and DIFROM, 16-4 310, 10
Error Codes (Appendix A), 1 311, 10
101, 1 312, 10
110, 2 330, 11
111, 2 348, 11
120, 3 351, 12
1300, 29 352, 12
1500, 30 401, 12
1501, 30 402, 13
1502, 30 403, 13
1503, 31 404, 14

405, 14
406, 14
407, 15
408, 15
409, 15
420, 16
501, 16
502, 17
505, 17
510, 17
520, 18
525, 18
537, 19
601, 19
602, 19
603, 20
630, 20
648, 21
701, 21
703, 22
710, 22
712, 23
714, 23
716, 24
720, 24
726, 24
730, 25
740, 25
742, 26
744, 26
746, 27
8090, 37
8095, 37
810, 27
820, 28
840, 28
842, 28
845, 29
Introduction, 1
Examples
Call to VA FileMan DBS, 2-12
EXCLUDED NAME SPACE
PACKAGE File and DIFROM, 16-3
Executable Help
Advanced File Definition, 12-11
Exiting, Quitting, Saving, and Obtaining Help
ScreenMan Form Editor, 4-6
Export Tool API
Calls
EXPORT^DDMP, 7-11
EXPORT^DDMP: Data Export, 7-11
Exporting Data
DIFROM, 16-2
Exporting File Security
DIFROM, 16-14
Extract Data
EN^DIAXU, 8-1
EXTRACT^DIAXU, 8-4
Extract Tool, 8-1
Calls
EN^DIAXU, 8-1
EXTRACT^DIAXU, 8-4
Introduction, 8-1
EXTRACT^DIAXU: Extract Data, 8-4

F

FDA Loader
FDA^DILF( ), 2-158
FDA Value Retriever (Single)
$$VALUE1^DILF( ), 2-164
FDA Values Retriever
VALUES^DILF( ), 2-166
FDA: Format of Data Passed to and from VA FileMan, 2-5
FDA^DILF( ): FDA Loader, 2-158
Features
ScreenMan Forms, 3-5
Field
ScreenMan Forms Field Properties, 3-28
FIELD, 16-4
Field Definition 0-Node
Global File Structure, 11-8
Field Global Storage
Advanced File Definition, 12-3
Field Identifiers
Global File Structure, 11-5

Field Number Retriever
  $$FLDNUM^DILFD( ), 2-174
Field Order
  ScreenMan Forms Field Properties, 3-28
Field Properties (ScreenMan Forms)
  Branching Logic, Pre Action, Post Action, and Post Action on Change, 3-33
  Caption and Data Coordinates, 3-31
  Caption, Executable Caption, and Suppress Colon After Caption, 3-29
  Data Length, 3-30
  Data Validation, 3-33
  Default and Executable Default, 3-29
  Disable Editing and Disallow LAYGO, 3-32
  Display Group, 3-32
  Field, 3-28
  Field Order, 3-28
  Field Type, 3-28
  Required, 3-31
  Right Justify, 3-31
  Subpage Link, 3-33
  Unique Name, 3-29
Field Type
  ScreenMan Forms Field Properties, 3-28
Field Verifier
  $$VFIELD^DILFD( ), 2-180
FIELD^DID( ): DD Field Retriever, 2-105
FIELDLST^DID( ): DD Field List Retriever, 2-108
Fields
  Adding Fields with ScreenMan Form Editor, 4-11
  Validator
    VALS^DIE( ), 2-145
FILE, 16-4
  PACKAGE File and DIFROM, 16-4
File Characteristics Nodes
  Global File Structure, 11-4
File Entries (Data Storage)
  Global File Structure, 11-16
File Global Storage
  Advanced File Definition, 12-1
File Header
  Global File Structure, 11-2
File Root Resolver
  $$ROOT^DILFD( ), 2-178
File Verifier
  $$VFILE^DILFD( ), 2-181
FILE^DDMP: Data Import, 7-2
FILE^DICN, 1-40
FILE^DID( ): DD File Retriever, 2-109
FILE^DIE( ): Filer, 2-120
Filegrams API, 9-1
  Calls
    ^DIFG, 9-1
    EN^DIFGG, 9-5
  Introduction, 9-1
FILELST^DID( ): DD File List Retriever, 2-112
FileMan Error Codes, 1
  Introduction, 1
FileMan Functions (Creating), 15-1
Filer
  FILE^DIE( ), 2-120
Files
  Advanced File Definition, 12-1
  DIALOG File, 14-1, 14-4
  Global File Structure, 11-1
  LANGUAGE File, 14-6
  MUMPS OPERATING SYSTEM File, 16-29
  PACKAGE File, 16-2
File's Entry in the Dictionary of Files
  Global File Structure, 11-1
FIND^DIC( ): Finder, 2-38
Finder
  FIND^DIC( ), 2-38
Finder (Single Record)
  $$FIND1^DIC( ), 2-65
Form (ScreenMan Forms)
  Layout: Forms and Pages Properties, 3-20

Data Validation, 3-20
Form Name, 3-20
Post Save, 3-20
Pre Action and Post Action, 3-20
Record Selection Page, 3-21
Title, 3-20
Structure, 3-1
Form Doc Print
^DIWF, 1-135
EN1^DIWF, 1-137
EN2^DIWF, 1-139
Form Editor
ScreenMan, 4-1
Adding Blocks, 4-10
Adding Fields, 4-11
Adding Pages, 4-9
Adding, Selecting, and Editing, 4-4
Block Viewer Screen, 4-7
Choosing Another Form, 4-21
Command Summary, 4-2
Deleting Screen Elements, 4-22
Editing "Pop-Up" Page Coordinates, 4-18
Editing Block Properties, 4-17
Editing Field Captions and Data Length, 4-15
Editing Field Properties, 4-14
Editing Form Properties, 4-20
Editing Page Properties, 4-18
Exiting, Quitting, Saving, and Obtaining Help, 4-6
Going to Another Page, 4-9
Header Blocks, 4-10
Introduction, 4-1
Invoking, 4-1
Main Screen, 4-5
Moving Screen Elements, 4-3, 4-12
Navigating on the Form Editor Screens, 4-8
Navigating on the Main Screen and Block Viewer Screen, 4-2
Quick Page Navigation, 4-3
Reordering All Fields on a Block, 4-16
Selecting Screen Elements, 4-11
Format and Conventions
DBS Calls, 2-2
Formatter
^DIWP, 1-141
Form-Only Fields
ScreenMan Forms, 3-7
Forms and Pages
ScreenMan Forms: Form Layout, 3-1
Forms in ScreenMan, 3-1
Forward Pointers
Relational Navigation
ScreenMan Forms, 3-9
FUNCTION, 16-29
Function File Entries
VA FileMan Functions (Creating), 15-1
Functional Description, 1
Functions
Creating VA FileMan Functions, 15-1

G

Gathers
Miscellaneous Package Components
DIFROM, 16-14
Templates and Forms
DIFROM, 16-17
General Processing
DIFROM, Running an INIT, 16-27
Generator
EN^DIFGG, 9-5
GETS^DIQ( ): Data Retriever, 2-186
Global File Structure, 11-1
Attribute Dictionary, 11-4
Cross-references, 11-6, 11-17
Data Dictionary Audit, 11-5
Data Storage Conventions, 11-1
Distribution Package, 11-7
Field Definition 0-Node, 11-8
Field Identifiers, 11-5
File Characteristics Nodes, 11-4
File Entries (Data Storage), 11-16
File Header, 11-2
File's Entry in the Dictionary of Files, 11-1

How to Read the Attribute
Dictionary: An Example, 11-14
INDEX File, 11-16
Introduction, 11-1
KEY File, 11-16
Other Field Definition Nodes, 11-11
Package Revision Data, 11-7
Post-Action, 11-4
Screens, 11-7
Special Lookup, 11-5
Version Number, 11-7
Write Identifiers, 11-6
Glossary, 1
Going to Another Page
ScreenMan Form Editor, 4-9

H

H^%DTC, 1-158
Header Block
ScreenMan Form Editor, 4-10
ScreenMan Forms Page Property, 3-23
Header Blocks
ScreenMan Form Editor, 4-10
HELP FRAMES, 16-28
Help Messages (ScreenMan)
HLP^DDSUTL( ), 5-14
MSG^DDSUTL( ), 5-15
HELP^%DTC, 1-159
HELP^DIE( ): Helper, 2-125
Helper
HELP^DIE( ), 2-125
HLP^DDSUTL( ), 5-14
How Information Is Returned
DBS Calls, 2-7
How the Database Server (DBS) communicates, 2-7
How to
Read the Attribute Dictionary: An Example
Global File Structure, 11-14
Use the DBS calls, 2-2
HTML Encoder/Decoder
$$HTML^DILF( ), 2-161

I

Identifying the Init Routines
DIFROM, 16-12
IENS Creator
$$IENS^DILF( ), 2-162
IENS: To Identify Entries and Subentries, 2-3
Import Tool API, 7-1
Calls
FILE^DDMP, 7-2
Introduction, 7-1
Importing Data
DIFROM, 16-19
Including
Other Package Components
DIFROM, 16-14
Templates (No Package File Entry)
DIFROM, 16-14
INDEX File
Global File Structure, 11-16
INPUT Template, 16-29
INPUT Transform
Advanced File Definition, 12-7
INPUT Transforms and the Verify Fields Option
Advanced File Definition, 12-8
Installer
^DIFG, 9-1
Installing
Data
DIFROM, Running an INIT, 16-24
Data Dictionaries
DIFROM, Running an INIT, 16-23
Other Package Components
DIFROM, Running an INIT, 16-26
Internal to External Date
^%DT, 1-145
DD^%DT, 1-151
Internationalization
Creating Non-English Text in the DIALOG File, 14-4
DIALOG File, 14-4
LANGUAGE File, 14-6
Role of the DIALOG File, 14-4

Use of the DIALOG File, 14-4
Introduction
Advanced File Definition, 12-1
DBS Calls, 2-1
DIALOG File, 14-1
DIFROM, 16-1
Extract Tool, 8-1
Filegrams API, 9-1
Global File Structure, 11-1
Import Tool API, 7-1
LANGUAGE File, 14-6
ScreenMan Form Editor, 4-1
ScreenMan Forms, 3-1
Trigger Cross-references, 13-1
VA FileMan Functions (Creating), 15-1
Invoke ScreenMan
^DDS, 5-1
Invoking the ScreenMan Form Editor, 4-1
Is This a "Pop-Up" Page?
ScreenMan Forms Page Property, 3-23
IX^DIC, 1-32
IX^DIK, 1-69
IX1^DIK, 1-71
IXALL^DIK, 1-73

K

KEY File
Global File Structure, 11-16
Key Validator
$$KEYVAL^DIE( ), 2-129

L

LANGUAGE File, 14-6
Creating LANGUAGE File Entries, 14-7
Introduction, 14-6
Use of the LANGUAGE File, 14-6
LAYGO
ScreenMan Forms Field Properties, 3-32
Linking Pages of a Form
ScreenMan Forms, 3-2
LIST^DIC( ): Lister, 2-81
Lister
LIST^DIC( ), 2-81
Lookup
Special Lookup Programs
Advanced File Definition, 12-9
Lookup DBS Calls
$$FIND1^DIC( ), 2-65
FIND^DIC( ), 2-38
LIST^DIC( ), 2-81
Lookup/Adding Entries
^DIAC, 1-12
^DIC, 1-14
DQ^DICQ, 1-44
FILE^DICN, 1-40
IX^DIC, 1-32
MIX^DIC1, 1-36

M

Main Screen
ScreenMan Form Editor, 4-5
MAY USER OVERRIDE DATA UPDATE, 16-7
MAY USER OVERRIDE DD UPDATE, 16-6
MENU, 16-28
MERGE OR OVERWRITE SITE'S DATA, 16-7
MIX^DIC1, 1-36
Moving Screen Elements
ScreenMan Form Editor, 4-3, 4-12
MSG^DDSUTL( ), 5-15
MSG^DIALOG( ): Output Generator, 2-33
MUMPS Data Type
Advanced File Definition, 12-6
MUMPS OPERATING SYSTEM File, 16-29

N

Name
ScreenMan Forms Block Properties, 3-27

ScreenMan Forms Page Property, 3-22
NAME
PACKAGE File and DIFROM, 16-3
Navigating
Form Editor Screens
ScreenMan Form Editor, 4-8
On the Form Editor Screens
ScreenMan Form Editor, 4-8
Quick Page Navigation ScreenMan Form Editor, 4-3
ScreenMan Form Editor on the Main Screen and Block Viewer Screen, 4-2
Via DD Fields—Syntax for Pointer Link
ScreenMan Forms, 3-10
Via Form Only Fields—Syntax for Pointer Link
ScreenMan Forms, 3-11
New-Style Index Deleter
DELIXN^DDMOD, 2-21
Next Page and Previous Page
ScreenMan Forms Page Property, 3-23
NOW^%DTC, 1-160
Number
ScreenMan Forms Page Property, 3-22

O

Obtaining Formatted Text From The Arrays
DBS Calls, 2-11
Obtaining, Exiting, Saving, and Quitting Help
ScreenMan Form Editor, 4-6
OPEN^DDBRZIS, 6-15
Options
ScreenMan, 3-35
ScreenMan Forms
Delete a Form, 3-36
Edit/Create a Form, 3-35
Purge Unused Blocks, 3-37
Run a Form, 3-35
Verify Fields Option
Advanced File Definition, 12-8
OPTIONS, 16-28
Order Entry and DIFROM, 16-9
Other APIs, III-1
Other Field Definition Nodes
Global File Structure, 11-11
Other PACKAGE File Fields
PACKAGE File and DIFROM, 16-7
Output Generator
MSG^DIALOG( ), 2-33
OUTPUT Transform
Advanced File Definition, 12-9
Overview
DBS Calls, 2-7

P

PACKAGE File and DIFROM, 16-2
ENVIRONMENT CHECK ROUTINE, 16-4
EXCLUDED NAME SPACE, 16-3
FILE, 16-4
NAME, 16-3
Other PACKAGE File Fields, 16-7
POST-INITIALIZATION ROUTINE, 16-4
PREFIX, 16-3
PRE-INIT AFTER USER COMMIT, 16-4
Template Multiples, 16-3
PACKAGE FILE ENTRIES, 16-28
Package Identification
DIFROM, 16-12
PACKAGE PARAMETERS, 16-28
Package Revision Data
Global File Structure, 11-7
Package Revision Data Initializer
PRD^DILFD( ), 2-175
Page (ScreenMan Forms)
Properties, 3-22
Coordinate and Lower Right Coordinate, 3-22
Header Block, 3-23

Is This a "Pop-Up" Page?, 3-23
Name, 3-22
Next Page and Previous Page, 3-23
Number, 3-22
Parent Field, 3-23
Pre Action and Post Action, 3-24
Pages
Adding Blocks with ScreenMan Form Editor, 4-9
Parent Field
ScreenMan Forms Page Property, 3-23
Pointer Link
ScreenMan Forms Block Properties, 3-26
Post Save
ScreenMan Forms Form Properties, 3-20
POST^DDBRZIS, 6-16
Post-Action
Global File Structure, 11-4
POST-INITIALIZATION ROUTINE
PACKAGE File and DIFROM, 16-4
Post-Selection Action
Advanced File Definition, 12-10
PRD^DILFD( ): Package Revision Data Initializer, 2-175
Pre Action and Post Action
ScreenMan Forms Block Properties, 3-26, 3-27
ScreenMan Forms Form Properties, 3-20
ScreenMan Forms Page Property, 3-24
PREFIX
PACKAGE File and DIFROM, 16-3
PRE-INIT AFTER USER COMMIT
PACKAGE File and DIFROM, 16-4
Pre-init After User Commit Routine
DIFROM, Running an INIT, 16-23
Preliminary Steps
DIFROM, Running an INIT, 16-19
Preliminary Validations
DIFROM, 16-12
Preparing To Run DIFROM, 16-2
PRINT Template, 16-29
PRINT^DDS, 3-42
Printing
^DIWF, 1-135
^DIWP, 1-141
^DIWW, 1-143
DT^DIQ, 1-106
EN^DIQ, 1-107
EN^DIS, 1-129
EN1^DIP, 1-81
EN1^DIWF, 1-137
EN2^DIWF, 1-139
Y^DIQ, 1-108
Programmer Access
^DI, 10-1
Programmer Mode Utilities
^DDGF, 3-39
CLONE^DDS, 3-40
PRINT^DDS, 3-42
RESET^DDS, 3-43
ScreenMan Forms, 3-39
Programmer Tools, IV-1
Prompting/Messages
^DIR, 1-113
EN^DDIOL, 1-8
HELP^%DTC, 1-159
WAIT^DICD, 1-39
YN^DICN, 1-43
Properties
Block Coordinate
ScreenMan Forms, 3-26
Block Name
ScreenMan Forms, 3-25
Block Order
ScreenMan Forms, 3-25
Block Properties Stored in the BLOCK File
ScreenMan Forms, 3-27
Block Properties Stored in the FORM File
ScreenMan Forms, 3-25
Branching Logic, Pre Action, Post Action, and Post Action on Change
ScreenMan Forms, 3-33

Caption and Data Coordinates
ScreenMan Forms, 3-31
Caption, Executable Caption, and Suppress Colon After Caption
ScreenMan Forms, 3-29
Data Length
ScreenMan Forms, 3-30
Data Validation
ScreenMan Forms, 3-20, 3-33
DD Number
ScreenMan Forms, 3-27
Default and Executable Default
ScreenMan Forms, 3-29
Disable Editing and Disallow LAYGO
ScreenMan Forms, 3-32
Disable Navigation
ScreenMan Forms, 3-27
Display Group
ScreenMan Forms, 3-32
Editing Block Properties with ScreenMan Form Editor, 4-17
Editing Field Properties with ScreenMan Form Editor, 4-14
Editing Form Properties with ScreenMan Form Editor, 4-20
Editing Page Properties with ScreenMan Form Editor, 4-18
Field
ScreenMan Forms, 3-28
Field Order
ScreenMan Forms, 3-28
Field Type
ScreenMan Forms, 3-28
Form Name
ScreenMan Forms, 3-20
Header Block
ScreenMan Forms, 3-23
Is This a "Pop-Up" Page?
ScreenMan Forms, 3-23
Name
ScreenMan Forms, 3-27
Next Page and Previous Page
ScreenMan Forms, 3-23
Page Coordinate and Lower Right Coordinate
ScreenMan Forms, 3-22
Page Name
ScreenMan Forms, 3-22
Page Number
ScreenMan Forms, 3-22
Parent Field
ScreenMan Forms, 3-23
Pointer Link
ScreenMan Forms, 3-26
Post Save
ScreenMan Forms, 3-20
Pre Action and Post Action
ScreenMan Forms, 3-20, 3-24, 3-26, 3-27
Record Selection Page
ScreenMan Forms, 3-21
Replication, Index, Initial Position, Disallow LAYGO, Field for Selection
ScreenMan Forms, 3-26
Required
ScreenMan Forms, 3-31
Right Justify
ScreenMan Forms, 3-31
Subpage Link
ScreenMan Forms, 3-33
Title
ScreenMan Forms, 3-20
Type of Block
ScreenMan Forms, 3-25
Unique Name
ScreenMan Forms, 3-29
Properties of Form-Only Fields
ScreenMan Forms, 3-8
PROTOCOL TO EXPORT, 16-28
Purge Unused Blocks ScreenMan Forms Option, 3-37
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12

Q

Quick Page Navigation
ScreenMan Form Editor, 4-3

Quitting, Exiting, Saving, and Obtaining Help
ScreenMan Form Editor, 4-6

R

Reader
^DIR, 1-113
Recall Record Number
RECALL^DILFD( ), 2-176
RECALL^DILFD( ): Recall Record Number, 2-176
Record Selection Page
ScreenMan Forms Form Properties, 3-21
Recording the Install on the Target System
DIFROM, Running an INIT, 16-30
Referencing
Data Dictionary Fields
ScreenMan Forms, 3-14
Form-Only and Computed Fields
ScreenMan Forms, 3-15
Refresh Screen (ScreenMan)
REFRESH^DDSUTL( ), 5-16
REFRESH^DDSUTL( ), 5-16
Reindexing the Files
DIFROM, Running an INIT, 16-26
Relational Navigation (ScreenMan Forms)
Backward Pointers, 3-12
Forward Pointers, 3-9
Reordering All Fields on a Block
ScreenMan Form Editor, 4-16
Replication, Index, Initial Position, Disallow LAYGO, Field for Selection
ScreenMan Forms Block Properties, 3-26
REQ^DDSUTL( ), 5-17
Required
ScreenMan Forms Field Properties, 3-31
RESET^DDS, 3-43
Retrieve/Stuff Fields (ScreenMan)
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12
Right Justify
ScreenMan Forms Field Properties, 3-31
Role of the VA FileMan DIALOG File in Internationalization, 14-4
Root Converter
Closed to Open Format
$$OREF^DILF( ), 2-163
Open to Closed Format
$$CREF^DILF( ), 2-152
Run a Form ScreenMan Forms Option, 3-35
Running
An INIT (Steps), 16-19
DIFROM (Steps), 16-11
Environment Check Routine
DIFROM, Running an INIT, 16-20
Post-Initialization Routine
DIFROM, Running an INIT, 16-29
Run-Time Field Status (ScreenMan)
REQ^DDSUTL( ), 5-17
UNED^DDSUTL( ), 5-18

S

S^%DTC, 1-161
Screen Elements
Moving with ScreenMan Form Editor, 4-3
SCREEN TEMPLATES (FORMS), 16-29
SCREEN TO DETERMINE DD UPDATE, 16-6
Screened Pointers and Set of Codes
Advanced File Definition, 12-6
ScreenMan, II-1
API, 5-1
Introduction, 5-1
Calls
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
^DDS, 5-1

HLP^DDSUTL( ), 5-14
MSG^DDSUTL( ), 5-15
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12
REFRESH^DDSUTL( ), 5-16
REQ^DDSUTL( ), 5-17
UNED^DDSUTL( ), 5-18
Form Editor, 4-1
Adding Blocks, 4-10
Adding Fields, 4-11
Adding Pages, 4-9
Adding, Selecting, and Editing, 4-4
Block Viewer Screen, 4-7
Choosing Another Form, 4-21
Command Summary, 4-2
Deleting Screen Elements, 4-22
Editing "Pop-Up" Page Coordinates, 4-18
Editing Block Properties, 4-17
Editing Field Captions and Data Length, 4-15
Editing Field Properties, 4-14
Editing Form Properties, 4-20
Editing Page Properties, 4-18
Exiting, Quitting, Saving, and Obtaining Help, 4-6
Going to Another Page, 4-9
Header Blocks, 4-10
Introduction, 4-1
Invoking, 4-1
Main Screen, 4-5
Moving Screen Elements, 4-3, 4-12
Navigating on the Form Editor Screens, 4-8
Navigating on the Main Screen and Block Viewer Screen, 4-2
Quick Page Navigation, 4-3
Reordering All Fields on a Block, 4-16
Selecting Screen Elements, 4-11
Forms, 3-1
Backward Pointers
Relational Navigation, 3-12
Block Properties, 3-25, 3-28
Callable Routines, 3-39
Computed Fields, 3-13
Data Filing, 3-19
DDSBR Variable, 3-17
DDSSTACK Variable, 3-18
Displaying Multiples in Repeating Blocks, 3-5
Features, 3-5
Form Layout: Forms and Pages, 3-1
Form Properties, 3-20
Form Name, 3-20
Form Structure, 3-1
Form-Only Fields, 3-7
Forward Pointers
Relational Navigation, 3-9
Introduction, 3-1
Linking Pages of a Form, 3-2
Navigating Via DD Fields—Syntax for Pointer Link, 3-10
Navigating Via Form Only Fields—Syntax for Pointer Link, 3-11
Options, 3-35
Delete a Form, 3-36
Edit/Create a Form, 3-35
Purge Unused Blocks, 3-37
Run a Form, 3-35
Page Properties, 3-22
Programmer Mode Utilities, 3-39
Properties
Block Coordinate, 3-26
Block Name, 3-25
Block Order, 3-25
Block Properties Stored in the BLOCK File, 3-27
Block Properties Stored in the FORM File, 3-25
Branching Logic, Pre Action, Post Action, and Post Action on Change, 3-33
Caption and Data Coordinates, 3-31
Caption, Executable Caption, and Suppress Colon After Caption, 3-29
Data Length, 3-30
Data Validation, 3-20, 3-33
DD Number, 3-27

Default and Executable Default, 3-29
Disable Editing and Disallow LAYGO, 3-32
Disable Navigation, 3-27
Display Group, 3-32
Field, 3-28
Field Order, 3-28
Field Type, 3-28
Header Block, 3-23
Is This a "Pop-Up" Page?, 3-23
Name, 3-27
Next Page and Previous Page, 3-23
Page Coordinate and Lower Right Coordinate, 3-22
Page Name, 3-22
Page Number, 3-22
Parent Field, 3-23
Pointer Link, 3-26
Post Save, 3-20
Pre Action and Post Action, 3-20, 3-24, 3-26, 3-27
Record Selection Page, 3-21
Replication, Index, Initial Position, Disallow LAYGO, Field for Selection, 3-26
Required, 3-31
Right Justify, 3-31
Subpage Link, 3-33
Title, 3-20
Type of Block, 3-25
Unique Name, 3-29
Properties of Form-Only Fields, 3-8
Referencing
Data Dictionary Fields, 3-14
Form-Only and Computed Fields, 3-15
Relational Navigation
Backward Pointers, 3-12
Forward Pointers, 3-9
Syntax for Pointer Link—Navigating Via
DD Fields, 3-10
Form Only Fields, 3-11
Variables Available in Repeating Blocks, 3-6
Help Messages Calls
HLP^DDSUTL( ), 5-14
MSG^DDSUTL( ), 5-15
Programmer Mode Utilities
^DDGF, 3-39
CLONE^DDS, 3-40
PRINT^DDS, 3-42
RESET^DDS, 3-43
Refresh Screen Calls
REFRESH^DDSUTL( ), 5-16
Retrieve/Stuff Fields Calls
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12
Run-Time Field Status Calls
REQ^DDSUTL( ), 5-17
UNED^DDSUTL( ), 5-18
ScreenMan Calls
$$GET^DDSVAL( ), 5-5
$$GET^DDSVALF( ), 5-10
^DDS, 5-1
HLP^DDSUTL( ), 5-14
MSG^DDSUTL( ), 5-15
PUT^DDSVAL( ), 5-7
PUT^DDSVALF( ), 5-12
REFRESH^DDSUTL( ), 5-16
REQ^DDSUTL( ), 5-17
UNED^DDSUTL( ), 5-18
ScreenMan Forms
Block Properties that Apply Only to Repeating Blocks, 3-6
Screens
Global File Structure, 11-7
Search File Entries
EN^DIS, 1-129
SECURITY KEYS, 16-29
Selecting Screen Elements
ScreenMan Form Editor, 4-11
Selecting, Adding, and Editing
ScreenMan Form Editor, 4-4
Single Data Retriever
$$GET1^DIQ( ), 2-182

SORT Template, 16-29
Special Lookup
  Global File Structure, 11-5
Special Lookup Programs
  Advanced File Definition, 12-9
Special Processing
  DIFROM, Running an INIT, 16-28
Specifications for Exported Files
  DIFROM, 16-13
Specifying Routine Size
  DIFROM, 16-14
Standalone VA FileMan, 2
Starting
  DIFROM, 16-12
  The Update
    DIFROM, Running an INIT, 16-23
Storing Data
  By Position within a Node
    Advanced File Definition, 12-4
  In a Global other than ^DIZ
    Advanced File Definition, 12-1
SUBORDINATE KEY, 16-29
Subpage Link
  ScreenMan Forms Field Properties, 3-33
Syntax for Pointer Link—Navigating Via
  DD Fields
    ScreenMan Forms, 3-10
  Form Only Fields
    ScreenMan Forms, 3-11

T

Template Compilation
  ^DIEZ, 1-59
Template Multiples
  PACKAGE File and DIFROM, 16-3
Templates
  ^DIEZ, 1-59
  ^DIOZ, 1-80
  ^DIPT, 1-101
  ^DIPZ, 1-103
  DIBT^DIPT, 1-102
  EN^DIEZ, 1-60
  EN^DIPZ, 1-104
Text Editing
  EN^DIWE, 1-132
Title
  ScreenMan Forms Form Properties, 3-20
Tools
  Extract, 8-1
  For Developers, IV-1
  Import Tool API, 7-1
Traditional Cross-reference Deleter
  DELIX^DDMOD, 2-17
Transforms
  INPUT Transform
    Advanced File Definition, 12-7
  OUTPUT Transform
    Advanced File Definition, 12-9
Trigger Cross-references, 13-1
  Introduction, 13-1
  Trigger on the Same File, 13-2
  Triggers for Different Files, 13-4
Trigger on the Same File
  Trigger Cross-references, 13-2
Triggers for Different Files
  Trigger Cross-references, 13-4
Type of Block
  ScreenMan Forms Block Properties, 3-25

U

UNED^DDSUTL( ), 5-18
Unique Name
  ScreenMan Forms Field Properties, 3-29
UPDATE THE DATA DICTIONARY, 16-5
UPDATE^DIE( ): Updater, 2-131
Updater
  UPDATE^DIE( ), 2-131
Use of the
  DIALOG File, 14-1
    In Internationalization, 14-4
  LANGUAGE File, 14-6
User Dialog DBS Calls

$$EZBLD^DIALOG( ), 2-31
BLD^DIALOG( ), 2-24
MSG^DIALOG( ), 2-33
User Messages
DIALOG File, 14-1
Using
Identifiers to Verify a Match, 16-25
Internal Entry Number to Verify a Match, 16-25
Utilities
$$ROUSIZE^DILF, 1-77
%XY^%RCR, 1-164
^DIM, 1-78
COMMA^%DTC, 1-155
DO^DIC1, 1-35
DT^DICRW, 1-45
EN^DID, 1-46
EN^DIU2, 1-130
Utility DBS Calls
$$CREF^DILF( ), 2-152
$$EXTERNAL^DILFD( ), 2-168
$$HTML^DILF( ), 2-161
$$IENS^DILF( ), 2-162
$$OREF^DILF( ), 2-163
$$VALUE1^DILF( ), 2-164
CLEAN^DILF, 2-151
DA^DILF( ), 2-153
DT^DILF( ), 2-154
FDA^DILF( ), 2-158
VALUES^DILF( ), 2-166

V

VA FileMan
Error Codes, 1
Introduction, 1
Functional Description, 1
Functions (Creating), 15-1
Function File Entries, 15-1
Introduction, 15-1
Standalone, 2
What is it?, 1
VAL^DIE( ): Validator, 2-140
Validator
VAL^DIE( ), 2-140
VALS^DIE( ): Fields Validator, 2-145
VALUES^DILF( ): FDA Values Retriever, 2-166
Variables Available in Repeating Blocks
ScreenMan Forms, 3-6
Version Number
Global File Structure, 11-7

W

WAIT^DICD, 1-39
What is VA FileMan?, 1
Word Processing Filer
WP^DIE( ), 2-149
WP Print
^DIWP, 1-141
^DIWW, 1-143
WP^DDBR, 6-7
WP^DIE( ): Word Processing Filer, 2-149
Write Identifiers
Global File Structure, 11-6

X

X ^DD("DD"), 1-6
Xecutable Help, 2-8

Y

Y^DIQ, 1-108
YMD^%DTC, 1-162
YN^DICN, 1-43
YX^%DTC, 1-163

Z

Zero Node
Checking, 16-24