

5. Any WRITE (display-only) identifiers
6. The results of executing the Lister's IDENTIFIER parameter

FLAGS

(Optional) Flags to control processing:

- B** **Backwards.** Traverses the index in the opposite direction of normal traversal.
- I** **Internal 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 Key** used for default index.
- M** **Mnemonic suppression.** Tells the Lister to ignore any mnemonic cross-reference entries it finds in the index.
- P** **Pack 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** **Quick 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 **Unscreened 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 "nth" 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.
- Y0 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
```

- o 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_Identifier^Write_Identifier^
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 "^". 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.

FIELDLIST^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

```
FIELDLIST^DID(TARGET_ROOT)
```

Input Parameters

TARGET_ROOT (Required) The root of an output array.

Output

TARGET_ROOT The descendants 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 FIELDLIST^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	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:
	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 Zero 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 Help (single "?") is returned if VALUE is not valid. E External 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, Xexecutable 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 External 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 Locking is done by the Filer. (See discussion of Locking.)

S Save 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 Transaction 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 Uniqueness or completeness. Without the U flag, the values in the FDA are checked to ensure that the integrity of any key in which an included field participates is not violated.

(CAUTION: 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 All available help for the field.
 - B Brief variable pointer help. A single line beginning with "To see the entries ...".
- NOTE:** See also Limitations under Details and Features below.
- C Set of Codes screen description.

- D** Description text for the field; this may be multiple lines.
- F** Fields 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** Getting help from pointed-to file. Help for the .01 field of pointed-to file is returned.
- H** Help prompt text.
- M** More variable pointer help. Detailed description of how to enter variable pointer data.
- P** Pointer screen description.
- S** Set 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/Time generic help. This help text is customized based on the allowable and required elements of the particular Date/Time field.
- U** Unscreened set of codes choices.
- V** Variable pointer help that lists the prefixes and messages associated with a particular variable pointer field.
- X** Xecutable 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 Quit 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 External values are processed. If this flag is set, the values in the FDA must be in the format input by the user. The Updater validates all values and converts them to internal format. Invalid values cancel the entire transaction.

If the flag is not set, values must be in internal format and must be valid.

K If a file has a primary key, the primary **Key** fields, not the .01 field, are used for lookup for Finding and LAYGO Finding nodes.

- S** The Updater Saves 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: <ul style="list-style-type: none"> E External value is returned in RESULT(0). F FDA node is set for valid data in array identified by FDA_ROOT. H Help (single ?) is returned if VALUE is not valid. R Record 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: <ul style="list-style-type: none"> R Records 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: <ul style="list-style-type: none"> A Append 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 Lock 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** External, readable date returned in zero-node of **RESULT**.
- F** Future dates are assumed.
- N** Numeric-only input is not allowed.
- P** Past dates are assumed.
- R** Required time input.
- S** Seconds will be returned.
- T** Time input is allowed but not required.
- X** **EX**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 "YYYYDDMM.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 Record 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** Use 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 Internal format is returned. (The default is external.)
Z	Z Zero 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 External values in nodes ending with "E". I Returns Internal values in nodes ending with "I". (Otherwise, external is returned).

N	Does not return Null values.
R	Resolves field numbers to field names in target array subscripts.
Z	Word processing fields include Zero 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
```

Database Server (DBS) API

```
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 OS

```

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.