Release Notes for FM Beta 0.4.19

Alex Hunter - AFH Systems
February 2001

FM built-in functions to access the Windows Registry

Alex Hunter
Draft 1.0 - April 29, 2001
Draft 1.2 (minor typos fixed) - February 12, 2002

Release Notes for FilterMeister 1.0.0

Alex Hunter, Harald Heim
AFH Systems
1 April 2003

Release Notes for FM Beta 0.4.19

Alex Hunter - AFH Systems
February 2001

Following are the incremental release notes for FM Beta version 0.4.19.
Unless otherwise noted, the release notes for versions 0.4.18 and earlier (see below) also apply to version 0.4.19.

1. New Features

1.1 Specifying PiPL Properties for Standalone Filters

While many plug-in filter properties can be set dynamically at filter execution time, some properties can be specified only statically as fields in the resource records of the plug-in filter file. Most such static properties are specified as fields of an Adobe-specific resource record called a PiPL ("Plug-in Property List", pronounced "pipple").

FM 0.4.19 now allows the filter designer to specify several of these PiPL fields. The 'SupportedModes' key specifies a list of image modes supported by your filter. The 'EnableInfo' key specifies a more general set of conditions for enabling or disabling your plug-in within the host application's menus. The 'FilterCaseInfo' key allows you to specify pre- and post-processing to be performed on the target image for each possible filter case.

Since these properties are specified statically in the PiPL resource of your generated standalone filter, you CANNOT test these properties at interactive design time (i.e., while you are designing your filter within FilterMeister itself). The only way to observe the effects of specifying these properties is to "Make" a new standalone filter within FM, exit FM, exit the host application, restart the host application, and finally invoke your new standalone filter. This is the only reliable method to force the host application to read and act upon the new PiPL information, since many hosts cache the PiPL information for each known plug-in.

1.1.1 SupportedModes Specification

Use this key to specify which image modes your filter supports. In Adobe Photoshop, this property determines whether your plug-in filter will be active (black) or inactive (gray) in Photoshop's menus, based on the current document's image mode. Third-party Photoshop-compatible hosts may or may not honor this property, so in general your filter should be prepared to be invoked with a document of any image mode, even if you indicate that your filter does not support that mode. In other words, this property is "advisory" only.

You can test the current image mode in the OnFilterStart event handler, and abort the invocation if the image mode is inappropriate. For example:

```
OnFilterStart:{
   if (imageMode != RGBMode) {
      ErrorOk("This filter works only with RGB images.");
      doAction(CA_CANCEL);
   }
   return false;
}
```

By default, FM specifies that the filter supports all image modes.
I.e., the default 'SupportedModes' specification is:

SupportedModes: BitmapMode, GrayScaleMode, IndexedColorMode, RGBMode, CMYKMode, HSLMode, HSBMode, MultichannelMode, DuotoneMode, LabMode, Gray16Mode, RGB48Mode

or simply:

SupportedModes: AllModes

This is also the default for FilterMeister itself, since you must be able to invoke FM no matter what the current image mode is.

If your standalone filter does NOT support all image modes, it is strongly advised that you code an explicit SupportedModes specification to help avoid being invoked in an unsupported mode.

Syntax

```
<supported_modes_specification>:
   'SupportedModes' ":" <image_mode_list>

<image_mode_list>:
   <image_mode> {"," <image_mode>}*

<image_mode>:
   'Bitmap*Mode'          //Bitmap mode
   'Gray*ScaleMode'       //Grayscale mode
   'Indexed*ColorMode'    //Indexed color mode
   'RGB*Mode'             //RGB color mode
   'CMYK*Mode'            //CMYK color mode
   'HSL*Mode'             //HSL color mode
   'HSB*Mode'             //HSB color mode
   'Multichannel*Mode'    //Multichannel mode
   'Duotone*Mode'         //Duotone mode
   'Lab*Mode'             //Lab color mode
   'Gray16*Mode'          //Grayscale mode, 16 bits per channel
   'RGB48*Mode'           //RGB color mode, 16 bits per channel
   'All*Modes'            //Any mode (don't gray out my filter menu!)
```

Note that the image mode names in the SupportedModes specification are case-insensitive and may be abbreviated. The asterisk (*) within the name indicates the point of minimum abbreviation, and is not part of the image mode name itself.

Examples

```
SupportedModes: RGBMode        //Handles RGB mode only

SupportedModes: Bitmap, GrayScale, RGB, CMYK, HSB   //using abbreviations

SupportedModes: ALLMODES       //Handles all image modes (the default)
```

In the last example, we specify that the filter is active in all modes, so it will never be grayed out in the Photoshop filter menu. However, the filter can still check the image mode in the OnFilterInit or OnFilterStart handler and explicitly reject those modes that we do not wish to handle (with an appropriate error message). This can sometimes be more elegant (and less confusing to the naive end user) than simply graying out the menu choice for this filter.

1.1.2 EnableInfo Specification

This key provides the filter designer with even finer control over whether the filter will be active or inactive ("grayed-out") in the Photoshop menus.

The 'EnableInfo' key supplies a (possibly quite complex) C or Modula-like Boolean expression that must be satisfied if the plug-in is to be enabled in the host menus. This expression is a superset of the 'SupportedModes' property. If you specify both the 'SupportedModes' key and the 'EnableInfo' key, the conditions in each must be compatible, or you will get unknown results. In other words, the SupportedModes and EnableInfo properties should always match with regard to image mode requests.

Note: There is no guarantee (especially with non-Adobe hosts) that the EnableInfo key will be honored. Even if the EnableInfo expression evaluates to false (therefore disabling your plug-in from being selected), your plug-in may still be invoked in some other way. Always double-check the required conditions for your filter (e.g., in the OnFilterInit event handler) before executing any code that depends on these conditions, and abort with an appropriate error message if you're in the wrong mode.

Syntax

```
<enable_info_specification>:
   'EnableInfo' ":" <booleanExpression>

<booleanExpression>:
   <conjunction> {"||" <conjunction>}*

<conjunction>:
   <relation> {"&amp;&amp;" <relation>}*

<relation>:
   <equality> {<relationOperator> <equality>}*

<equality>:
   <simpleExpression> {<equalityOperator> <simpleExpression>}*

<simpleExpression>:
   <term> {<addOperator> <term>}*

<term>:
   <factor> {<mulOperator> <factor>}*

<factor>:
   <integer> | <intrinsic> | <ident> | "(" <booleanExpression> ")" |
   "+" <factor> | "-" <factor> | "!" <factor>

<integer>:
   {digit}+

<intrinsic>:
   <limitFunction> | <dimFunction> | <inFunction>

<limitFunction>:
   {"min" | "max"} "(" <simpleExpression> "," <simpleExpression>
               {"," <simpleExpression>}* ")"

<dimFunction>:
   "dim" "(" <simpleExpression> "," <simpleExpression> ")"

<inFunction>:
   "in" "(" <simpleExpression> {"," <simpleExpression>}* ")"

<ident>:
   <namedConstant> | <namedVariable>

<mulOperator>:
   "*" | "/"

<addOperator>:
   "+" | "-"

<equalityOperator>:
   "==" | "!="

<relationOperator>:
   "<" | "<=" | ">=" | ">"
```

The <namedConstant>'s that you can use in an EnableInfo <booleanExpression> are:

Constant	*Description*
true	Boolean true
false	Boolean false
BitmapMode	Bitmap mode
GrayScaleMode	Grayscale mode
IndexedColorMode	Indexed color mode
RGBMode	RGB color mode
CMYKMode	CMYK color mode
HSLMode	HSL color mode
HSBMode	HSB color mode
MultichannelMode	Multichannel mode
DuotoneMode	Duotone mode
LabMode	Lab color mode
Gray16Mode	Grayscale mode, 16 bits per channel
RGB48Mode	RGB color mode, 16 bits per channel

Note that, unlike the SupportedModes specification, the names of image modes in the EnableInfo specification are case-sensitive and cannot be abbreviated (since they are C-like expression elements).

The <namedVariable>'s that you can use in an EnableInfo <booleanExpression> are:

Variable	*Description*
PSHOP_ImageMode	Image mode
PSHOP_ImageDepth	Image depth
PSHOP_HasLayerMask	Boolean for presence of a layer mask
PSHOP_HasSelectionMask	Boolean for presence of a selection mask
PSHOP_HasTransparencyMask	Boolean for presence of a transparency mask
PSHOP_NumTargetChannels	Number of target channels
PSHOP_NumTrueChannels	Number of image channels
PSHOP_IsTargetComposite	Boolean for whether image is flattened
PSHOP_ImageWidth	Width of the image
PSHOP_ImageHeight	Height of the image

Result of undefined values

The result of any arithmetic operation where at least one of the operands is undefined, or an undefined variable, results in 0 or 'false'. The result of a compare (see <relationOperator>) where at least one of the operands is undefined results in 'false'.

Boolean values are treated as in C/C++ where non-zero is 'true' and zero is 'false' with the exception that an undefined value is also 'false'.

in Function

The 'in' function returns 'true' if the first parameter is equal to at least one of the following parameters. A typical use might be to see if the image mode is RGB, CMYK, or Lab:

```
in(PSHOP_ImageMode, RGBMode, CMYKMode, LabMode)
```

Examples

```
EnableInfo: PSHOP_ImageMode == RGBMode      //filter works in RGB mode only
```

The following two keys specify the same set of supported image modes (as they *must* if you want predictable results):

SupportedModes: BitmapMode, GrayScaleMode, RGBMode, CMYKMode, HSBMode
EnableInfo: in(PSHOP_ImageMode, BitmapMode, GrayScaleMode, RGBMode, CMYKMode, HSBMode)

In the next example, we require that a selection mask be present if the filter is to be enabled (and the image mode must be RGB or CMYK):

SupportedModes: RGBMODE, CMYKMODE
EnableInfo: PSHOP_HasSelectionMask &&
(PSHOP_ImageMode==RGBMode || PSHOP_ImageMode==CMYKMode)

The following specification is for use in a demonstration version of a filter, which will handle images only in RGB mode, and no larger than 300 x 200 pixels in size:

EnableInfo: (PSHOP_ImageMode == RGBMode) &&
(PSHOP_ImageWidth <= 300) && (PSHOP_ImageHeight <= 200)

In the last example, the filter designer should also test the specified conditions in the OnFilterInit handler to ensure that the EnableInfo conditions are not somehow bypassed.

1.1.3 FilterCaseInfo Specification

This key specifies which filter cases your filter can handle, and exactly how each case will be processed.

Prior to Photoshop 3.0, there were essentially only three filter cases:

1. A flat image with no selection mask. This is a background layer or a flat image with no transparency data or selection.
2. A flat image with a selection mask. The image has no transparency data, but a selection may be present. The selection is presented to the filter as mask data.
3. A "floating" selection. The image data is presented to the filter with an accompanying mask.

Some third-party host applications still support only these three basic cases.

Photoshop 3.0 introduced support for dynamically composited layers of image data. A layer consists of color and transparency information for each pixel it contains. Filters can choose to alter transparency data as well as color data. For example, a spatial distortion filter will most likely choose to move or distort transparency data along with color data.

The following new filter cases were added to provide full support for layers:

4. Editable transparency with no selection. This is a layer with transparency editing enabled and no selection. Image data is presented to the filter as color data and an alpha channel.
5. Editable transparency with selection. This is a layer with transparency editing enabled and a selection. Image data is presented to the filter as color data, an alpha channel, and selection mask data.
6. Protected transparency with no selection. This is a layer with transparency editing disabled and no selection. Image data is presented to the filter as color data and a read-only alpha channel.
7. Protected transparency with selection. This is a layer with transparency editing disabled and a selection. Image data is presented to the filter as color data, a read-only alpha channel, and selection mask data.

Photoshop 3.0+ offers quite a bit of flexibility in how transparency data is presented to filters. The 'FilterCaseInfo' property key is used to control the filtering process and how image data is presented to the plug-in. This property provides information to Photoshop about what image data cases the plug-in supports. Photoshop then compares the current filtering situation to the filter-supported cases and chooses the best-fitting case that the plug-in is able to handle. The image data is then presented to the filter in that format. If none of the filter-supported cases are usable, the filter will be disabled in the menus.

If the editable transparency cases are unsupported by your filter, then Photoshop will try the corresponding protected transparency cases. This governs whether your filter will be expected to filter the transparency data along with the color data.

In Photoshop 4.0, support for floating selections was dropped or subsumed under more general layer support. However, Photoshop 4.0+ (and some third- party hosts) may still choose to present image data to a plug-in filter as a floating selection, particularly if the filter does not provide support for the more general layer cases. This allows filters that were designed for use with Photoshop 2.5 and earlier to still handle many of the cases involving layer data (e.g., by presenting the layer data with transparency as though it were a floating selection).

If the protected transparency case without a selection is not supported by your filter, then the layer data is treated as a floating selection. The transparency data will be presented to your filter via the mask portion of the interface rather than with the input data as an alpha channel.

In addition to defining the supported filter cases, the 'FilterCaseInfo' property key also defines what kind of pre- and post-processing will be performed on the image data by the host application (such as matting and defringing); whether input image data will be pre-copied to the output image; whether the filter can be applied to completely "blank" images; whether the filter can process layer masks; and whether the filter can write data outside the selection mask.

Syntax

```
<filter_case_info_specification>:
   'FilterCaseInfo' ":" <filter_case_list>

<filter_case_list>:
   <filter_case_spec> {";" <filter_case_spec>}*

<filter_case_spec>:
   <filter_case_name_list> "=" <filter_handling_list>

<filter_case_name_list>:
   'DEFAULT'
   <filter_case_name> {"," <filter_case_name>}*

<filter_case_name>:
   'FlatImageNoSelection'
   'FlatImageWithSelection'
   'FloatingSelection'
   'EditableTransparencyNoSelection'
   'EditableTransparencyWithSelection'
   'ProtectedTransparencyNoSelection'
   'ProtectedTransparencyWithSelection'

<filter_handling_list>:
   <filter_handling_item> {"," <filter_handling_item>}*

<filter_handling_item>:
   <filter_input_handling_mode>
   <filter_output_handling_mode>
   <filter_handling_property>

<filter_input_handling_mode>:
   'inCantFilter'
   'inStraightData'
   'inBlackMat'
   'inGrayMat'
   'inWhiteMat'
   'inDefringe'
   'inBlackZap'
```

'inGrayZap'
'inWhiteZap'
'inBackgroundZap'
'inForegroundZap'

<filter_output_handling_mode>:
'outCantFilter'
'outStraightData'
'outBlackMat'
'outGrayMat'
'outWhiteMat'
'outFillMask'

<filter_handling_property>:
'copySourceToDestination' 'doNotCopySourceToDestination'
'doesNotWorkWithBlankData' 'worksWithBlankData'
'doesNotFilterLayerMasks' 'filtersLayerMasks'
'doesNotWriteOutsideSelection' 'writesOutsideSelection'

The <filter_input_handling_mode> and <filter_output_handling_mode> fields specify the pre-processing and post-processing actions on the image data, respectively. For input handling, the filter designer specifies one of 12 mutually exclusive modes to determine whether the filter supports this case, and if so, how transparent or partially transparent pixels should be treated. For output handling, the filter designer specifies one of six mutually exclusive modes to determine whether the filter supports this case, and if so, how transparency should be post-processed in the output image.

The first two options for input and output handling modes are simple:

'inCantFilter', 'outCantFilter'
These options specify that this case is not supported by the plug-in filter. If you specify either of these options, you should specify both (it makes no sense otherwise).

'inStraightData', 'outStraightData'
These options indicate that the plug-in filter does support this case, and that no special pre- or post-processing of the image data by the host is required. This is the default input handling mode for the 'FlatImageNoSelection' and 'FlatImageWithSelection' filter cases, and the default output handling mode for all filter cases.

One option for treating partially transparent pixel data involves blending the color data with a fixed color value in proportion to the degree of transparency. This process is called "matting" for input data or "dematting" for output data. Matting is particularly useful when performing distortions and blurs.

Mathematically, matting is defined as follows:

c' = ((c * a) + 128)/255 + ((K * (255 - a)) + 128)/255

where c is the unmatted color value of the pixel for the current color channel, c' is the resulting matted value, a is the alpha (transparency) channel value for the current pixel, and K is the matting constant (0 to 255). In effect, c is mixed in proportion to the alpha channel (a/255), while K is mixed in inverse proportion to the alpha channel ((255-a)/255). The "+ 128" terms are for rounding.

Note that when a is 255 (fully opaque), the above formula reduces to:

c' = c

and that when a is 0 (fully transparent), the formula reduces to:

c = K

Dematting is defined mathematically as the inverse of the matting operation:

c = (c' - K)*a/255 + K

where c' is the matted color value of the pixel for the current color channel, c is the resulting dematted value, a is the alpha (transparency) channel value for the current pixel, and K is the matting constant (0 to 255).

Again, when a is 255 (fully opaque), the dematting formula reduces to:

c = c'

and when a is 0 (fully transparent), the formula reduces to:

c = K

The three matting cases for each input/output mode differ only in the value of the matting constant K:

'inBlackMat', 'outBlackMat'
For input, matte the image data with black (K=0) based on the transparency.
For output, dematte the image data using black (K=0).

'inGrayMat', 'outGrayMat'
For input, matte the image data with gray (K=128) based on the transparency.
For output, dematte the image data using gray (K=128).

'inWhiteMat', 'outWhiteMat'
For input, matte the image data with white (K=255) based on the transparency.
For output, dematte the image data using white (K=255).

For input processing only, the following six handling modes are also defined:

'inDefringe'
Defringe transparent areas by filling any fully transparent pixels (a==0) with the color values from the nearest opaque or partially transparent pixels (a > 0) using "taxicab" distance. (The "taxicab" distance between a pixel at point x1,y1 and a pixel at point x2,y2 is abs(x2-x1)+abs(y2-y1); i.e., it is the distance a taxicab would traverse by driving between the two points only along streets defined by an orthogonal "grid".)

'inBlackZap'
Sets the color component of all totally transparent pixels (a==0) to black.

'inGrayZap'
Sets the color component of all totally transparent pixels (a==0) to gray.

'inWhiteZap'
Sets the color component of all totally transparent pixels (a==0) to white.

'inBackgroundZap'
Sets the color component of all totally transparent pixels (a==0) to the current background color. This is the default input handling mode for all filter cases except 'FlatImageNoSelection' and 'FlatImageWithSelection' (for compatibility with Adobe's Filter Factory).

'inForegroundZap'
Sets the color component of all totally transparent pixels (a==0) to the current foreground color.
For output processing only, the following mode is also defined:
'outFillMask'
This mode results in the transparency mask being automatically filled with full opacity (a==255) in the area affected by the filter. This is only valid for the editable transparency cases. This option is provided to make it easy to write a plug-in filter similar to Photoshop's Clouds filter, which fills an area with a value.

In addition to the input and output handling modes, the filter designer can specify one or more of the following properties for each filter case:

'copySourceToDestination' (host default)
'doNotCopySourceToDestination' (FM default)

These two mutually exclusive choices determine whether the host will automatically copy all source data to the destination before invoking your filter. By default, the host copies the data for you. This can result in a small but unnecessary performance degradation if your filter also writes data to all of the output pixels; you can avoid this penalty by specifying the 'doNotCopySourceToDestination' property.

Note: For compatibility with all hosts, FilterMeister always explicitly copies the source data to the destination for each tile before invoking the ForEveryTile handler. To avoid unnecessary overhead, use the FM default setting 'doNotCopySourceToDestination' to turn off superfluous copying by the host.

'doesNotWorkWithBlankData'
'worksWithBlankData'

Many filters are useless when applied to a completely "blank" image area (i.e., to an area that is totally transparent). This property determines whether your filter can be applied to such blank areas ('worksWithBlankData'), or whether it will be grayed-out in the menus in such a case ('doesNotWorkWithBlankData'). Note that the 'worksWithBlankData' property is only useful for the editable transparency cases where your filter can create opacity by modifying the output alpha channel -- in the protected transparency cases, you would be left with what you started with: completely blank (transparent) data. By default, FilterMeister sets the 'worksWithBlankData' property for the editable transparency cases, and 'doesNotWorkWithBlankData' for all other cases.

'doesNotFilterLayerMasks' (default)
'filtersLayerMasks'

In cases where transparency is editable, this property determines if Layer Masks are filtered (see the "Add Layer Mask" item in the Layers palette menu to create a layer mask). Specifying 'filtersLayerMasks' adds the layer mask to the set of target channels presented to your filter if (1) transparency for the layer is editable (i.e., this must be one of the editable transparency cases), and (2) the layer mask is specified as being positioned relative to the layer rather than to the image in Layer Mask Options. (The distinction based on position is based on the assumption that layer-relative masks are distorted with the layer, while image-relative masks are independent of the layer.)

'doesNotWriteOutsideSelection' (default)
'writesOutsideSelection'

In the image-with-selection and layer-with-selection cases, this property specifies whether the filter wants to write image data beyond the confines of the selection. The default is 'doesNotWriteOutsideSelection'. While it is generally considered rude to alter data outside of the user's selection, in some cases it is necessary and appropriate (for example, when creating a drop shadow based on the current selection). If you use the 'writesOutsideSelection' property, be sure to support layer transparency data as an alternate mask.

Examples

The default FilterCaseInfo created by FilterMeister is equivalent to the following FilterCaseInfo specification:

```
FilterCaseInfo:
   FlatImageNoSelection,
   FlatImageWithSelection=
      inStraightData,outStraightData,
      doNotCopySourceToDestination,
      doesNotWorkWithBlankData,
      doesNotFilterLayerMasks,
      doesNotWriteOutsideSelection;
   FloatingSelection=
      inBackgroundZap,outStraightData,
      doNotCopySourceToDestination,
      doesNotWorkWithBlankData,
      doesNotFilterLayerMasks,
      doesNotWriteOutsideSelection;
```

```
    EditableTransparencyNoSelection,
    EditableTransparencyWithSelection=
        inBackgroundZap,outStraightData,
        doNotCopySourceToDestination,
        worksWithBlankData,
        doesNotFilterLayerMasks,
        doesNotWriteOutsideSelection;

    ProtectedTransparencyNoSelection,
    ProtectedTransparencyWithSelection=
        inBackgroundZap,outStraightData,
        doNotCopySourceToDestination,
        doesNotWorkWithBlankData,
        doesNotFilterLayerMasks,
        doesNotWriteOutsideSelection
```

The following example is for a drop shadow filter. We require a filter case that either has an active selection or an editable transparency mask; a flat image with no selection, or a protected transparency case would be unusable for our purposes. Furthermore, we specify that in the active cases we intend to write outside the selection mask, and we request that the output transparency be set to fully opaque.

```
FilterCaseInfo:
    //unusable cases...
    FlatImageNoSelection,
    ProtectedTransparencyNoSelection,
    ProtectedTransparencyWithSelection
        = inCantFilter,outCantFilter;

    //active cases...
    FlatImageWithSelection,
    FloatingSelection,
    EditableTransparencyNoSelection,
    EditableTransparencyWithSelection
        = inStraightData,outFillMask,writesOutsideSelection
```

To set all filter cases active, with the input handling mode set to 'inDefringe' and the 'worksWithBlankData' property set for all cases, you can simply code:

```
FilterCaseInfo:
    DEFAULT = inDefringe, worksWithBlankData
```

1.2 Open/Save Filename Common Dialogs

Two new built-in functions have been added to expose the Windows Open File and Save File common dialogs to the filter designer.

1.2.1 getOpenFileName

The getOpenFileName() built-in function allows the filter designer to display the standard Windows 95+ Open File common dialog to obtain the name of a file to be opened.

The calling sequence for getOpenFileName() is:

```
int iError =
    getOpenFileName( flags,
                lpstrFile, nMaxFile,
                [&amp;amp;nFileOffset], [&amp;amp;nFileExtension],
                [lpstrFileTitle], nMaxFileTitle,
                [lpstrFilter],
                [lpstrCustomFilter], nMaxCustFilter,
```

```
                [&amp;amp;nFilterIndex],
                [lpstrInitialDir],
                [lpstrDialogTitle],
                [lpstrDefExt],
                [&amp;amp;oflags]
              );
```

Input parameters (see 1.2.3):

```
    int flags;  //  May include 0 or more of the following flag bits (see 1.2.4):
                OFN_EXPLORER (always set)
                OFN_ALLOWMULTISELECT
                OFN_CREATEPROMPT
                OFN_FILEMUSTEXIST (implies OFN_PATHMUSTEXIST)
                OFN_HIDEREADONLY
                OFN_NOCHANGEDIR
                OFN_NODEREFERENCELINKS
                OFN_NOREADONLYRETURN
                OFN_NOTESTFILECREATE
                OFN_NOVALIDATE
                OFN_OVERWRITEPROMPT (ignored for Open)
                OFN_PATHMUSTEXIST
                OFN_READONLY (initial state of readonly checkbox)
                OFN_SHAREAWARE
                OFN_SHOWHELP

    [const char *lpstrFilter];          //e.g. "JPEG Files (*.jpg;*.jpeg)\0*.jpg;*.jpeg\0"
                                        "Compuserve GIF Files (*.gif)\0*.gif\0"
                                        "TIFF Files (*.tif;*.tiff)\0*.tif;*.tiff\0"
                                        "All Files (*.*)\0*.*\0"
    int nMaxCustFilter;                 // length of lpstrCustomFilter buffer in chars.
    int nMaxFile;                       // length of lpstrFile buffer in chars.
    int nMaxFileTitle;                  // length of lpstrFileTitle buffer.
    [const char *lpstrInitialDir];      //initial file directory [NULL=cwd]
    [const char *lpstrDialogTitle];     //for title bar [NULL="Open"]
    [const char *lpstrDefExt];          //default file extension [NULL=none]
```

Input/output parameters (see 1.2.3):

```
    [char *lpstrCustomFilter];          //init to e.g. "Custom filter\0*.*\0"
    [int *nFilterIndex];                //default is nFilterIndex=1
    char *lpstrFile;                    //default/returned file name
```

Output parameters (see 1.2.3):

```
    [char *lpstrFileTitle];             // receives short title
    [int *nFileOffset];                 // index into lpstrFile buffer of base filename
    [int *nFileExtension];              // index into lpstrFile buffer of file extension
    [int *oflags];                      // May contain 0 or more of the following flag bits (see 1.2.4):
                                        OFN_EXTENSIONDIFFERENT
                                        OFN_READONLY    //state of checkbox on return
```

Return value:

```
     0  if successful,
    -1  if user closed or canceled the dialog,
        else some other error code (see 1.2.5)
```

Example 1 (minimal):

This example shows a minimal usage of getOpenFileName, in which all optional arguments are defaulted to NULL. No file filter is specified, so all files in the current working directory (cwd) will be displayed in the "Open" dialog box.

The OFN_HIDEREADONLY flag is specified to suppress the "Open as read-only" checkbox that would otherwise be displayed in the dialog box.

The currently selected file name is maintained in global string str0, which is initialized by default to the null string. On subsequent invocations, the previously selected file name will be displayed as the initial choice.

```
int status;

status = getOpenFileName(OFN_HIDEREADONLY,      //flags
                    str0, 256,              //set, receive file name
                    NULL, NULL,      //no file/extension offsets
                    NULL, 0,              //no file title for display
                    NULL,                 //no file filters
                    NULL, 0,               //no custom filter
                    NULL,                 //no filter index
                    NULL,                 //initial dir = cwd
                    NULL,                 //dialog title="Open"
                    NULL,                 //no default extension
                    NULL                  //no output flags
                   );
if (status == 0) {
   //success:
   //open the file and read from it...
}
else if (status == -1) {
   //user closed or canceled the dialog box
}
else {
   //some other error
}
```

Example 2 (typical):

This example shows a more typical use of getOpenFileName, in which a set of file filters is specified, as well as a default file extension. The filter index is allowed to default to 1, meaning that the first file filter ("Image Files...") will be used by default on each invocation.

```
int status;

status = getOpenFileName(OFN_HIDEREADONLY,      //flags
                    str0, 256,              //set, receive file name
                    NULL, NULL,      //no file/extension offsets
                    NULL, 0,               //no file title for display
                                           //file filters...
                    "Image Files (*.bmp;*.gif;*.jpg;*.tif)\0*.bmp;*.gif;*.jpg;*.tif\0"
                    "All Files (*.*)\0*.*\0",
                    NULL, 0,              //no custom filter
                    NULL,                 //filter index=1
                    NULL,                 //initial dir = cwd
                    NULL,                 //dialog title="Open"
                    "bmp",                //default extension
                    NULL                  //no output flags
                   );
if (status == 0) {
   //success
}
else if (status == -1) {
   //user closed or canceled the dialog box
}
```

```
else {
   //some other error
}
```

Example 3 (full):

This example shows how to fully utilize all the optional arguments to getOpenFileName.

Global variable 'k0' is used as a one-shot "first time" flag to initialize the default file name (str0), initial directory (str1), custom user filter (str2), and current filter index (k1) when the plug-in is first started. The initial directory is set to the directory in which the plug-in was installed ('filterInstallDir'), which may be a more appropriate choice than allowing it to default to the current working directory (cwd). The directory is set back to the null string for subsequent invocations, which allows the Open File dialog to track any changes to the cwd (by default, selecting a file from the Open File dialog sets the cwd to the directory for that file). Note, however, that the cwd may also be changed in other manners during the course of your plug-in's execution, so the Open File dialog may not always be set to the directory you expect.

A more elaborate method to keep track of the default directory for the Open File dialog might involve parsing and saving the directory path from str0 each time a file is successfully selected. Variables 'nFileOffset' and 'nFileExtension' will receive the offsets of the base file name and file extension, respectively, within the fully-qualified file name in str0 upon successful selection. This can be useful, for example, for parsing the file name to extract the directory path for use in a subsequent invocation.

Global variable k1 keeps track of the most recently selected file filter index, so a subsequent invocation of getOpenFileName will by default use the file filter selected on the previous invocation.

A custom user file filter is established in global string str2. This filter will be automatically updated with any wildcard pattern entered by the user during a successful file selection, and can be explicitly selected by setting the filter index (k1) to 0.

A truncated version of the selected file name, suitable for display, will be returned in string str3. Typically, the path name will be stripped from this file name. The file extension may also be stripped, depending on Windows Explorer options set by the end user.

To demonstrate the use of the output flags, the input flags are set to 0, which causes the Open File dialog to display an "Open as read-only" checkbox by default. Upon return from getOpenFileName, the state of this checkbox is determined by testing the OFN_READONLY bit in the output flags (oflags) so that appropriate action can be taken.

Finally, note that the Open File dialog title is explicitly set to "Load Parameters".

```
int nFileOffset;
int nFileExtension;
int oflags;
int status;

if (k0 == 0) {
   //first time plug-in is invoked...
   strcpy(str0, "");                          //set default file name
   strcpy(str1, filterInstallDir);            //set initial directory
   k1 = 1;                                    //set initial filter index
   memcpy(str2, "Custom Filter\0*.*\0", 18);  //initialize custom filter (can't use strcpy
                                              // because of embedded NULs!)
   k0 = 1;                                    //no longer the first time
}
status = getOpenFileName(0,                   //show "read-only" checkbox by default
                  str0, 256,                  //set, receive file name
                  &amp;nFileOffset,           //receives offset of filename in str0
                  &amp;nFileExtension,        //receives offset of extension in str0
                  str3, 256,                  //receives file title for display
```

```
                                                    //File filters...
                    "Text Files (*.txt)\0*.txt\0"
                    "All Files (*.*)\0*.*\0",
                    str2, 256,                      //custom filter
                    &amp;k1,                        //current filter index
                    str1,                           //initial dir
                    "Load Parameters",              //dialog title
                    "txt",                          //default extension
                    &amp;oflags                     //output flags
                   );

  if (status == 0) {
    //success
    strcpy(str1, "");   //use cwd on subsequent calls
    if (oflags &amp; OFN_READONLY) {
      //warning, User checked "Open as read-only"!
      Warn("File %s will be opened as read-only!", str3);
    }
  }
  else if (status == -1) {
    //user closed or canceled the dialog box
  }
  else {
    //some other error
  }
```

1.2.2 getSaveFileName

The getSaveFileName() built-in function allows the filter designer to display the standard Windows 95+ Save File common dialog to obtain the name of a file to be saved (written to).

The calling sequence for getSaveFileName() is:

```
int iError =
  getSaveFileName( flags,
                   lpstrFile, nMaxFile,
                   [&amp;nFileOffset], [&amp;nFileExtension],
                   [lpstrFileTitle], nMaxFileTitle,
                   [lpstrFilter],
                   [lpstrCustomFilter], nMaxCustFilter,
                   [&amp;nFilterIndex],
                   [lpstrInitialDir],
                   [lpstrDialogTitle],
                   [lpstrDefExt],
                   [&amp;oflags]
                 );
```

Input parameters (see 1.2.3):

```
    int flags;  //  May include 0 or more of the following flag bits (see 1.2.4):
                  OFN_EXPLORER (always set)
                  OFN_ALLOWMULTISELECT
                  OFN_CREATEPROMPT
                  OFN_FILEMUSTEXIST (implies OFN_PATHMUSTEXIST)
                  OFN_HIDEREADONLY
                  OFN_NOCHANGEDIR
                  OFN_NODEREFERENCELINKS
                  OFN_NOREADONLYRETURN
                  OFN_NOTESTFILECREATE
                  OFN_NOVALIDATE
```

```
                OFN_OVERWRITEPROMPT (recommended)
                OFN_PATHMUSTEXIST
                OFN_READONLY (initial state of readonly checkbox)
                OFN_SHAREAWARE
                OFN_SHOWHELP

    [const char *lpstrFilter];          //e.g. "JPEG Files (*.jpg;*.jpeg)\0*.jpg;*.jpeg\0"
                                        "Compuserve GIF Files (*.gif)\0*.gif\0"
                                        "TIFF Files (*.tif;*.tiff)\0*.tif;*.tiff\0"
                                        "All Files (*.*)\0*.*\0"
    int nMaxCustFilter;                 // length of lpstrCustomFilter buffer in chars.
    int nMaxFile;                       // length of lpstrFile buffer in chars.
    int nMaxFileTitle;                  // length of lpstrFileTitle buffer.
    [const char *lpstrInitialDir];      //initial file directory [NULL=cwd]
    [const char *lpstrDialogTitle];     //for title bar [NULL="Save As"]
    [const char *lpstrDefExt];          //default file extension [NULL=none]

Input/output parameters (see 1.2.3):
    [char *lpstrCustomFilter];  //init to e.g. "Custom filter\0*.*\0"
    [int *nFilterIndex];        //default is nFilterIndex=1
    char *lpstrFile;            //default/returned file name

Output parameters (see 1.2.3):
    [char *lpstrFileTitle];     // receives short title
    [int *nFileOffset];         // index into lpstrFile buffer of base filename
    [int *nFileExtension];      // index into lpstrFile buffer of file extension
    [int *oflags];              // May contain 0 or more of the following flag bits (see 1.2.4):
                                OFN_EXTENSIONDIFFERENT
                                OFN_READONLY    //state of checkbox on return

Return value:
     0  if successful,
    -1  if user closed or canceled the dialog,
        else some other error code (see 1.2.5)
```

Example 1 (minimal):

This example shows a minimal usage of getSaveFileName, in which all optional arguments are defaulted to NULL. No file filter is specified, so all files in the current working directory (cwd) will be displayed in the "Save As" dialog box. The OFN_HIDEREADONLY flag is specified to suppress the "Open as read-only" checkbox that would otherwise be displayed in the dialog box. The OFN_OVERWRITEPROMPT flag is also specified to prompt the user for permission to overwrite an already existing file.

The currently selected file name is maintained in global string str0, which is initialized by default to the null string. On subsequent invocations, the previously selected file name will be displayed as the initial choice.

```
    int status;

    status = getSaveFileName(OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,      //flags
                    str0, 256,          //set, receive file name
                    NULL, NULL,         //no file/extension offsets
                    NULL, 0,            //no file title for display
                    NULL,               //no file filters
                    NULL, 0,            //no custom filter
                    NULL,               //no filter index
                    NULL,               //initial dir = cwd
                    NULL,               //dialog title="Save As"
                    NULL,               //no default extension
                    NULL                //no output flags                       );
```

```
if (status == 0) {
   //success
   //open the file and write to it...
}
else if (status == -1) {
   //user closed or canceled the dialog box
}
else {
   //some other error
}
```

Example 2 (typical):

This example shows a more typical use of getSaveFileName, in which a set of file filters is specified, as well as a default file extension. The filter index is allowed to default to 1, meaning that the first file filter ("Bitmap Files...") will be used by default on each invocation.

If the currently selected file name is null (as it will be the first time the plug-in is invoked), the file name "myfile.bmp" is proposed as a default.

```
int status;

if (strcmp(str0, "") == 0) {
   //propose a default file name...
   strcpy(str0, "myfile.bmp");
}
status = getSaveFileName(OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,     //flags
                  str0, 256,            //set, receive file name
                  NULL, NULL,     //no file/extension offsets
                  NULL, 0,            //no file title for display
                                      //File filters...
                  "Bitmap Files (*.bmp)\0*.bmp\0"
                  "Image Files (*.bmp;*.gif;*.jpg;*.tif)\0*.bmp;*.gif;*.jpg;*.tif\0"
                  "All Files (*.*)\0*.*\0",
                  NULL, 0,            //no custom filter
                  NULL,               //filter index=1
                  NULL,               //initial dir = cwd
                  NULL,               //dialog title="Save As"
                  "bmp",              //default extension
                  NULL                //no output flags
                 );
if (status == 0) {
   //success
}
else if (status == -1) {
   //user closed or canceled the dialog box
}
else {
   //some other error
}
```

Example 3 (full):

This example shows how to fully utilize all the optional arguments to getSaveFileName.

The logic for setting and maintaining the default file name (str0) initial directory (str1), custom user filter (str2), and current filter index (k1) is the same as in Example 3 for getOpenFileName above.

If the currently selected file name is null (as it will be the first time the plug-in is invoked), the file name of the plug-in itself (filterFilenameText) is proposed as a default file name.

A truncated version of the selected file name, suitable for display, will be returned in string str3.

In addition to the OFN_HIDEREADONLY and OFN_OVERWRITEPROMPT bits, the OFN_NOREADONLYRETURN bit is set in the input flags to prevent the user from selecting an existing read-only file or a file in a write- protected directory or device.

To demonstrate the use of the output flags, the OFN_EXTENSIONDIFFERENT bit in the output flags (oflags) is tested upon return, and a warning is issued if the extension of the selected file is not the default ("txt").

Finally, note that the Save File dialog title is explicitly set to "Save Parameters".

```
int nFileOffset;
int nFileExtension;
int oflags;
int status;

if (k0 == 0) {
   //first time...
   strcpy(str0, "");                              //set default file name
   strcpy(str1, filterInstallDir);                //set initial directory
   k1 = 1;                                        //initial filter index
   memcpy(str2, "Custom Filter\0*.*\0", 18);      //initialize custom filter (can't use strcpy
                                                  // because of embedded NULs!)
   k0 = 1;                                        //no longer first time
}
if (strcmp(str0, "") == 0) {
   //propose a default file name...
   strcpy(str0, filterFilenameText);
}
status = getSaveFileName(OFN_HIDEREADONLY |       //flags
                  OFN_OVERWRITEPROMPT |
                  OFN_NOREADONLYRETURN,
                  str0, 256,                      //set, receive file name
                  &amp;nFileOffset,               //receives offset of filename in str0
                  &amp;nFileExtension,            //receives offset of extension in str0
                  str3, 256,                      //receives file title for display
                                                  //File filters...
                  "Text Files (*.txt)\0*.txt\0"
                  "All Files (*.*)\0*.*\0",
                  str2, 256,                      //custom filter
                  &amp;k1,                        //current filter index
                  str1,                           //initial dir
                  "Save Parameters",              //dialog title
                  "txt",                          //default extension
                  &amp;oflags                     //output flags
                 );

if (status == 0) {
   //success
   strcpy(str1, "");   //use cwd on subsequent calls
   if (oflags &amp; OFN_EXTENSIONDIFFERENT) {
      //warning, output file is has non-standard extension!
      Warn("File %s does not have extension 'txt'!", str3);
   }
}
```

```
else if (status == -1) {
   //user closed or canceled the dialog box
}
else {
   //some other error
}
```

1.2.3 Parameter Definitions

const char *lpstrFilter [optional]

Pointer to a buffer containing pairs of NUL-terminated filter strings.
The last string in the buffer must be terminated by two NUL characters.

The first string in each pair is a display string that describes the filter (for example, "Text Files"), and the second string specifies the filter pattern (for example, "*.TXT"). To specify multiple filter patterns for a single display string, use a semicolon to separate the patterns (for example, "*.TXT;*.DOC;*.BAK"). A pattern string can be a combination of valid filename characters and the asterisk (*) wildcard character. Do not include spaces in the pattern string.

The operating system does not change the order of the filters. It displays them in the File Types combo box in the order specified in lpstrFilter.

If lpstrFilter is NULL, the dialog box does not display any filters.

char *lpstrCustomFilter [optional]

Pointer to a static buffer that contains a pair of NUL-terminated filter strings for preserving the filter pattern chosen by the user. The first string is your display string that describes the custom filter, and the second string is the filter pattern selected by the user. The first time your application creates the dialog box, you specify both the first and second strings. When the user selects a file, the dialog box copies the current filter pattern to the second string. The preserved filter pattern can be one of the patterns specified in the lpstrFilter buffer, or it can be a filter pattern typed by the user. The system uses the strings to initialize the user-defined file filter the next time the dialog box is created. If the nFilterIndex parameter is zero, the dialog box uses the custom filter.

If this parameter is NULL, the dialog box does not preserve user-defined filter patterns.

If this parameter is not NULL, the value of the nMaxCustFilter parameter must specify the size, in characters, of the lpstrCustomFilter buffer.

int nMaxCustFilter

Specifies the size, in characters, of the buffer identified by lpstrCustomFilter.
This buffer should be at least 40 characters long. This parameter is ignored if lpstrCustomFilter is NULL or points to a null string.

int *nFilterIndex [optional]

Points to an integer variable containing the index of the currently selected filter in the File Types control. The buffer pointed to by lpstrFilter contains pairs of strings that define the filters. The first pair of strings has an index value of 1, the second pair 2, and so on. An index of zero indicates the custom filter specified by lpstrCustomFilter. You can specify an index on input to indicate the initial filter description and filter pattern for the dialog box. When the user selects a file, nFilterIndex is set to the index of the currently displayed filter.

If this parameter is NULL, the current filter index is set to 1.

If nFilterIndex is zero and lpstrCustomFilter is NULL, the system uses the first filter in the lpstrFilter buffer. If all three parameters are zero or NULL, the system does not use any filters and shows all files in the file list control of the dialog box.

char *lpstrFile [optional]

Pointer to a buffer that contains a filename used to initialize the File Name edit control. The first character of this buffer must be NUL if initialization is not necessary. When the getOpenFileName or getSaveFileName function returns successfully, this buffer contains the drive designator, path, filename, and extension of the selected file.

If the OFN_ALLOWMULTISELECT flag is set and the user selects multiple files, the buffer contains the current directory followed by the filenames of the selected files. The directory and filename strings are NUL separated, with an extra NUL character after the last filename.

If the buffer is too small, the function returns FNERR_BUFFERTOOSMALL. In this case, the first two bytes of the lpstrFile buffer contain the required size in characters.

int nMaxFile

Specifies the size, in characters, of the buffer pointed to by lpstrFile. The getOpenFileName and getSaveFileName functions return FNERR_BUFFERTOOSMALL if the buffer is too small to contain the file information. The buffer should be at least 256 characters long.

char *lpstrFileTitle [optional]

Pointer to a buffer that receives the filename and extension (without path information) of the selected file. This parameter can be NULL.

int nMaxFileTitle

Specifies the size, in characters, of the buffer pointed to by lpstrFileTitle. This parameter is ignored if lpstrFileTitle is NULL.

const char *lpstrInitialDir [optional]

Pointer to a string that specifies the initial file directory. If this parameter is NULL, the system uses the current directory as the initial directory.

const char *lpstrDialogTitle [optional]

Pointer to a string to be placed in the title bar of the dialog box. If this parameter is NULL, the system uses the default title (that is, "Open" or "Save As").

int Flags

A set of bit flags you can use to initialize the dialog box. This parameter can be a combination of the following flags (see section 1.2.4 for a definition of each flag):

OFN_EXPLORER (always set)
OFN_ALLOWMULTISELECT
OFN_CREATEPROMPT
OFN_FILEMUSTEXIST (implies OFN_PATHMUSTEXIST)
OFN_HIDEREADONLY
OFN_NOCHANGEDIR
OFN_NODEREFERENCELINKS
OFN_NOREADONLYRETURN
OFN_NOTESTFILECREATE
OFN_NOVALIDATE
OFN_OVERWRITEPROMPT (recommended for Save, ignored for Open)
OFN_PATHMUSTEXIST
OFN_READONLY (initial state of readonly checkbox)
OFN_SHAREAWARE
OFN_SHOWHELP

int *nFileOffset [optional]

Pointer to an integer variable that will receive a zero-based offset from the beginning of the path to the filename in the string pointed to by lpstrFile. For example, if lpstrFile points to the following string, "c:\\dir1\\dir2\\file.ext", this parameter contains the value 13 to indicate the offset of the "file.ext" string.

int *nFileExtension [optional]

Pointer to an integer variable that will receive a zero-based offset from the beginning of the path to the filename extension in the string pointed to by lpstrFile. For example, if lpstrFile points to the following string, "c:\\dir1\\dir2\\file.ext", this parameter contains the value 18. If the user did not type an extension and lpstrDefExt is NULL, this member specifies an offset to the terminating NUL character. If the user typed "." as the last character in the filename, this parameter is set to zero.

const char *lpstrDefExt [optional]

Points to a buffer that contains the default extension. getOpenFileName and getSaveFileName append this extension to the filename if the user fails to type an extension. This string can be any length, but only the first three characters are appended. The string should not contain a period (.). If this parameter is NULL and the user fails to type an extension, no extension is appended.

int *oflags [optional]

Points to an integer variable which will receive a set of flag bits upon return from the function call. The following flag bits can be set in this parameter (see section 1.2.4 for a definition of each flag):

```
OFN_EXTENSIONDIFFERENT
OFN_READONLY    //state of read-only checkbox on return
```

Note: The bits from the input flags may also appear in the output flags.

If this parameter is NULL, no flag bits will be stored.

1.2.4 Input/Output Flags

OFN_ALLOWMULTISELECT (input)

Specifies that the File Name list box allows multiple selections.

If the user selects more than one file, the lpstrFile buffer returns the path to the current directory followed by the filenames of the selected files. The nFileOffset parameter receives the offset to the first filename, and the FileExtension parameter is not used. The directory and filename strings are NUL separated, with an extra NUL character after the last filename.

OFN_CREATEPROMPT (input)

If the user specifies a file that does not exist, this flag causes the dialog box to prompt the user for permission to create the file. If the user chooses to create the file, the dialog box closes and the function returns the specified name; otherwise, the dialog box remains open.

OFN_EXPLORER (input)

Specifies that the dialog box use the new Explorer-style interface. This flag will always be set, even if the caller does not explicitly set it.

OFN_EXTENSIONDIFFERENT (output)

Specifies that the user typed a filename extension that differs from the extension specified by lpstrDefExt. The function does not use this flag if lpstrDefExt is NULL.

OFN_FILEMUSTEXIST (input)

Specifies that the user can type only names of existing files in the File Name entry field. If this flag is specified and the user enters an invalid name, the dialog box procedure displays a warning in a message box. If this flag is specified, the OFN_PATHMUSTEXIST flag is also used.

OFN_HIDEREADONLY (input)

Hides the Read Only check box.

OFN_NOCHANGEDIR (input)

Restores the current directory to its original value if the user changed the directory while searching for files.

OFN_NODEREFERENCELINKS (input)

Directs the dialog box to return the path and filename of the selected shortcut (.LNK) file. If this value is not given, the dialog box returns the path and filename of the file *referenced* by the shortcut.

OFN_NOREADONLYRETURN (input)

Specifies that the returned file does not have the Read Only attribute set and is not in a write-protected directory.

OFN_NOTESTFILECREATE (input)

Specifies that the file is not created before the dialog box is closed. This flag should be specified if the application saves the file on a create-nonmodify network sharepoint. When an application specifies this flag, the library does not check for write protection, a full disk, an open drive door, or network protection. Applications using this flag must perform file operations carefully, because a file cannot be reopened once it is closed.

OFN_NOVALIDATE (input)

Specifies that the common dialog boxes allow invalid characters in the returned filename. Typically, the calling code checks the filename for validity. If the text box in the edit control is empty or contains nothing but spaces, the lists of files and directories are updated. If the text box in the edit control contains anything else, nFileOffset and nFileExtension are set to values generated by parsing the text. No default extension is added to the text, nor is text copied to the buffer specified by lpstrFileTitle.

If the value returned in nFileOffset is less than zero, the filename is invalid. Otherwise, the filename is valid, and nFileExtension and nFileOffset can be used as if the OFN_NOVALIDATE flag had not been specified.

OFN_OVERWRITEPROMPT (input)

Causes the Save As dialog box to generate a message box if the selected file already exists. The user must confirm whether to overwrite the file.

OFN_PATHMUSTEXIST (input)

Specifies that the user can type only valid paths and filenames. If this flag is used and the user types an invalid path and filename in the File Name entry field, the dialog box function displays a warning in a message box.

OFN_READONLY (input/output)

When specified as an input flag, causes the Read Only check box to be checked initially when the dialog box is created. In the output flags, this flag indicates the state of the Read Only check box when the dialog box is closed.

OFN_SHAREAWARE (input)

Specifies that if a call to the OpenFile function fails because of a network sharing violation, the error is ignored and the dialog box returns the selected filename.

OFN_SHOWHELP (input)

Causes the dialog box to display the Help button (not yet useful with FM).

1.2.5 Error Return Values

The following symbolic constants are predefined in FM to represent the various error values that can be returned by these functions.

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
FNERR_BUFFERTOOSMALL
FNERR_INVALIDFILENAME
FNERR_SUBCLASSFAILURE

Most of these error return values represent internal failures. However, you might wish to test explicitly for FNERR_INVALIDFILENAME and FNERR_BUFFERTOOSMALL.

The latter error indicates that the file name chosen was too large to fit in the buffer specified by the lpstrFile and nMaxFile parameters.

2. Changes

2.1 Source Editor

The FM source text editor now uses a Windows Rich Edit control instead of the standard Text Edit control as the basis for the editor.

You can choose the font used by the editor by clicking the new "Font..." button in the editor dialog box. Font preferences will be rememberedon a per-user basis across invocations and host sessions. The default font is the Windows stock ANSI fixed pitch font (usually Courier 10 pt).

The advantages of the Rich Edit control are:

- Greatly increased source capacity. FM now allows a maximum of 99,999 source characters. The previous limit was 30,000 characters.
- Drag and drop editing (internal and external).
- Improved selection of source tokens when double-clicked.
- Future support for a text selection bar.
- Future support for multiple-level Undo.
- Future support for the Unicode character set.
- Future support for syntax coloring.

2.2 Global Variables Saved Across Invocations

The values of the following predefined global variables are now saved across filter invocations within a single host session (this only works for hosts that correctly save the filter parameter block across invocations; Paint Shop Pro 6.0 and earlier does NOT save the filter parameter block):

```
int i2, i3, i4, i5, i6, i7, i8, i9;
int j0, j1, j2, j3, j4, j5, j6, j7, j8, j9;
int k0, k1, k2, k3, k4, k5, k6, k7, k8, k9;

double x0, x1, x2, x3, x4, x5, x6, x7, x8, x9;
double y0, y1, y2, y3, y4, y5, y6, y7, y8, y9;
double z0, z1, z2, z3, z4, z5, z6, z7, z8, z9;

char str0[256], str1[256], str2[256], str3[256], str4[256];
char str5[256], str6[256], str7[256], str8[256], str9[256];
```

Warning: Global variables i0 and i1 are NOT saved across invocations; these variables are reserved for Premiere compatibility.

Note that these variables are initialized to 0 (for int's), 0.0 (for doubles), and "" (for strings) during the first invocation of the filter within a host session. This means you can easily implemented a one-shot "first time" flag; for example, using global variable k0:

```
if (k0 == 0) {
   //perform first-time initializations here...
   k0 = 1;        //no longer the first time
}
```

2.3 Increased Literal Pool

The size of the literal pool has been significantly increased, allowing your FM program to contain many more floating point constants and string literals.

2.4 New Built-in C-RTL Functions

The following functions from the standard C Runtime Library (C-RTL) have been added. Microsoft extensions to the C-RTL are flagged with '(*)' and documented to the right. See any standard C Runtime Library reference for documentation of the other functions.

function name	***(*)=MS***	***prototype/description (MS extensions only)***
/* from <time.h> */		
strdate	(*)	char *strdate(char *datestr); Sets datestr to "mm/dd/yy"; returns datestr.
strtime (*)		char *strtime(char *timestr); Sets timestr to "hh:mm:ss"; returns timestr.
/* from <string.h> */		
strchr		
strcoll		
strcspn		
strdup	(*)	char *strdup(const char *strSource); Calls malloc to allocate a copy of strSource; returns allocated string, or NULL if error. Use free() to deallocate the duplicate.
strerror		
_strerror	(*)	char *_strerror(const char *strErrMsg); Returns system error message for last C-RTL call,

prefaced by strErrMsg if non-NULL.

stricmp (*) int stricmp(const char *str1, const char *str2);
Performs lowercase comparison of strings; returns
< 0 if str1 < str2, 0 if str1==str2, else > 0.

stricoll (*) int stricoll(const char *str1, const char *str2);
Performs locale-specific lowercase comparison of
strings; returns < 0 if str1 < str2, 0 if str1==str2, else > 0.

strlwr (*) char *_strlwr(char *str);
Converts str to lowercase; returns str.

strncoll (*) int strncoll(const char *str1, const char *str2, int n);
Performs locale-specific comparison of strings, max of
n chars; returns < 0 if str1 < str2, 0 if str1==str2, else > 0.

strnicmp (*) int strnicmp(const char *str1, const char *str2, int n);
Performs lowercase comparison of strings, max of n chars;
returns < 0 if str1 < str2, 0 if str1==str2, else > 0.

strnicoll (*) int strnicoll(const char *str1, const char *str2, int n);
Performs locale-specific lowercase comparison of strings,
max of n chars; returns < 0 if str1 < str2, 0 if str1==str2, else > 0.

strnset (*) char *strnset(char *str, int c, int n);
Sets (at most) first n chars of str to char c; returns str.

strpbrk

strrchr

strrev (*) char *strrev(char *str);
Reverses order of chars in str; returns str.

strset (*) char *strset(char *str, int c);
Sets all chars in str to char c; returns str.

strspn

strstr

strtok

strupr (*) char *_strupr(char *str);
Converts str to uppercase; returns str.

strxfrm

/* from <memory.h> */

memccpy (*) void *memccpy(void *dest, const void *src, int c, int n);
Copies bytes from src to dest, stopping after a byte value
c or n bytes have been copied; returns pointer to byte past
c in dest if c was copied, else NULL.

memchr

memcmp

memcpy

memicmp (*) int memicmp(const void *buf1, const void *buf2, int n);
Performs lowercase comparison of first n chars in buffers;
returns < 0 if buf1 < buf2, 0 if buf1==buf2, else > 0.

memmove

memset

/* from <stdlib.h> */

strtod

strtol

strtoul

/* from <malloc.h> */

calloc

expand	(*)	void *expand(void *memblock, int n); Changes size of heap memory block memblock to n bytes, without moving it; returns memblock if successful, else NULL.
free		
malloc		
msize	(*)	int msize(void *memblock); Returns size in bytes of heap-allocated memblock (possibly rounded upward to include padding).
realloc		
/* From <stdio.h> */		
fgetpos		
fsetpos		
printf		(Outputs to a message box titled "FilterMeister stdout".)
snprintf	(*)	int snprintf(char *buf, int n, const char *fmt[, arg]...); A "buffer-overflow safe" version of sprintf, which writes at most n chars to buf. Returns the number of chars written to buf, not counting the final NUL, or a negative value if more than n chars would be written to buf (in which case only the first n chars are written to buf).

2.5 New Named Constants

The following built-in named constants have been added:

Flag bit definitions for getOpen/SaveFileName

OFN_READONLY, OFN_OVERWRITEPROMPT, OFN_HIDEREADONLY, OFN_NOCHANGEDIR, OFN_SHOWHELP, OFN_NOVALIDATE, OFN_ALLOWMULTISELECT, OFN_EXTENSIONDIFFERENT, OFN_PATHMUSTEXIST, OFN_FILEMUSTEXIST, OFN_CREATEPROMPT, OFN_SHAREAWARE, OFN_NOREADONLYRETURN, OFN_NOTESTFILECREATE, OFN_EXPLORER, OFN_NODEREFERENCELINKS

Common Dialog Error Return Values for getOpen/SaveFileName

CDERR_DIALOGFAILURE, CDERR_STRUCTSIZE, CDERR_INITIALIZATION, CDERR_NOTEMPLATE, CDERR_NOHINSTANCE, CDERR_LOADSTRFAILURE, CDERR_FINDRESFAILURE, CDERR_LOADRESFAILURE, CDERR_LOCKRESFAILURE, CDERR_MEMALLOCFAILURE, CDERR_MEMLOCKFAILURE, CDERR_NOHOOK, CDERR_REGISTERMSGFAIL, CFERR_NOFONTS, CFERR_MAXLESSTHANMIN, FNERR_SUBCLASSFAILURE, FNERR_INVALIDFILENAME, FNERR_BUFFERTOOSMALL, FRERR_BUFFERLENGTHZERO, PDERR_SETUPFAILURE, PDERR_PARSEFAILURE, PDERR_RETDEFFAILURE, PDERR_LOADDRVFAILURE, PDERR_GETDEVMODEFAIL, PDERR_INITFAILURE, PDERR_NODEVICES, PDERR_NODEFAULTPRN, PDERR_DNDMMISMATCH, PDERR_CREATEICFAILURE, PDERR_PRINTERNOTFOUND, PDERR_DEFAULTDIFFERENT

New Control Indexes

CTL_ZOOM	-index of the system-reserved Zoom control
CTL_LAST_USER	-index of last available user-defined control

FilterMeister Limits

MAX_LABEL_SIZE	-max number of chars in a control label or dropdown list
MAX_TOOLTIP_SIZE	-max number of chars in a tooltip
MAX_SOURCE_CODE_SIZE	-max number of chars allowed by the source editor
MAX_DLITS	-size of floating-point/string literal pool (in 8 byte units)
MAX_LOCALS	-max number of local variables in a handler or user-defined function
MAX_TEMPS	-max number of compiler-generated temporaries per handler or user-defined function

MAX_CASE_LABELS	-max number of case labels per switch statement
N_CELLS	-number of anonymous get/put cells
N_CTLS	-max number of controls (user-defined and system reserved)

Image Modes (mixed case and uppercase equivalents)

BitmapMode, GrayScaleMode, IndexedColorMode,
RGBMode, CMYKMode, HSLMode, HSBMode, MultichannelMode,
DuotoneMode, LabMode, Gray16Mode, RGB48Mode

BITMAPMODE, GRAYSCALEMODE, INDEXEDCOLORMODE,
RGBMODE, CMYKMODE, HSLMODE, HSBMODE, MULTICHANNELMODE,
DUOTONEMODE, LABMODE, GRAY16MODE, RGB48MODE

2.6 Macromedia Fireworks Support

FM now recognizes 0 as the host signature for Macromedia Fireworks 4.0.

2.7 Increased Number of Controls

The number of user-defined and system reserved controls (N_CTLS) has been doubled to 128. The highest available user-defined control index (CTL_LAST_USER) is now 117.

2.8 New Built-in Variables

The following additional built-in (read-only) variables have been added to expose information provided by the host application. (Note that these values are guaranteed to be valid only when Adobe Photoshop is the host; other so-called Photoshop-compatible hosts may or may not provide useful information in these variables.)

Variable Name	*Type*	*Value*
filterRectBottom	int	Bottom coordinate (exclusive) of the area the image to be filtered. (This is the bounding box of the selection, or if there is no selection, the bounding box of the image.)
filterRectHeight	int	Height of the area of the image to be filtered.
filterRectLeft	int	Left coordinate (inclusive) of the area of the image to be filtered.
filterRectRight	int	Right coordinate (exclusive) of the area of the image to be filtered.
filterRectTop	int	Top coordinate (inclusive) of the area of the image to be filtered.
filterRectWidth	int	Width of the area of the image to be filtered.

3. Bug Fixes

3.1 String literals with embedded NULs were being truncated after the first NUL. Fixed in 0.4.19.

3.2 In some cases, choosing to save your source file after hitting Cancel resulted in setting the source file to 0 length. Fixed in 0.4.19.

3.3 A bug that prevented the recognition of the BN_* named constants has been fixed.

3.4 A bug that generated incorrect code when the second operand of the conditional expression ternary operator (?:) was a floating-point type has been fixed.

4. Known Problems

4.1 The 'sizeof' operator generates bug 4 in module MDC.

4.2 The FM dialog is not set back to its default size when compiling a new FM program with no explicit size for the dialog (i.e., the size given by a previous FM program is sticky).

4.3 Incorrect code is generated for the array index operator [] in some cases. (E.g., &str0[i] generates correct code, but str0[i] generates incorrect code.)

4.4 The FM compiler does not accept the combination 'unsigned long' in a type specification.

4.5 The strcoll() and strxfrm() built-in functions give incorrect results under Paint Shop Pro (results are correct under Photoshop and Fireworks).

4.6 The new source editor based on the Rich Edit control has the following problems:

-The right-click context menu has been (temporarily) lost.

-The scrollbars are not always adjusted correctly.

-Double-clicking an identifier does not select the entire identifier if it contains an underscore (_).

-Tabs outside the visible clipping rectangle are formatted incorrectly. I.e., only the first 18 or so tab stops work correctly. If you indent your code more than 18 levels deep, tab stops beyond the 18th level may be incorrect.

-Dragging or pasting source code from an external source can retain undesired formatting. To reset the source formatting to the default, close and re-open the source editor by clicking the Edit button twice.

FM built-in functions to access the Windows Registry

Alex Hunter
Draft 1.0 - April 29, 2001
Draft 1.2 (minor typos fixed) - February 12, 2002

1. Overview

The Windows Registry is a hierarchically-structured database which allows programs to save persistent data in various formats.

The FM registry access functions are designed to give the Filter Designer (FD) quick, convenient, and safe access to the Windows Registry.

There are 14 registry access functions:

- iErr = setRegRoot (int hkey);
- iErr = getRegRoot (int *hkey);
- iErr = setRegPath (lpsz szPath[, varargs]...);
- iErr = getRegPath (lpsz szPath, int maxPathLen);
- iErr = putRegInt (int iValue, lpsz szValueName[, varargs]...);
- iErr = getRegInt (int *piValue, lpsz szValueName[, varargs]...);
- iErr = putRegString (lpsz szString, lpsz szValueName[, varargs]...);
- iErr = getRegString (lpsz szString, int iMaxlen, lpsz szValueName[, varargs]...);
- iErr = putRegData (void *pArrayOrStruct, int dataLen, lpsz szValueName[, varargs]...);
- iErr = getRegData (void *pArrayOrStruct, int dataLen, lpsz szValueName[, varargs]...);
- iErr = enumRegSubKey (int index, lpsz szSubKey, int maxSubKeyLen);
- iErr = enumRegValue (int index, lpsz szValueName, int maxValueNameLen, int *piType, int *pcbData);
- iErr = deleteRegSubKey (lpsz szSubKey[, varargs]...);
- iErr = deleteRegValue (lpsz szValueName[, varargs]...);

The remainder of Section 1 gives an overview of these functions, grouped by functionality.

Section 2 serves as a reference manual, giving a detailed description and full usage information for each function.

Section 3 presents a number of examples of the usage of these functions, providing a source of useful code "snippets".

1.1 The default registry key

By default, FM accesses data values in the Registry under the key named:

HKEY_CURRENT_USER\Software\<Organization>\<Filter-category>\<Filter-title>

where:

<Organization> is the value set by the Organization: key in your FFP program,
<Filter-category> is the value set by the Category: key in your FFP program,
<Filter-title> is the value set by the Title: (or Name:) key in your FFP program (with any trailing ellipsis removed).

For example, the registry values for the following FFP Program:

```
%ffp
Category:      "FM Examples"
Title:         "Example 1"
Version:       "1.0"
Organization:  "AFH Systems"
Author:        "Alex Hunter"
   . . .
```

will be stored by default under the Registry key:

HKEY_CURRENT_USER\Software\AFH Systems\FM Examples\Example 1

Note that this default key path is in compliance with the Microsoft convention which requires that keys for "Product X" produced by "Company ABC" should be stored under "ROOT\Software\Company ABC\Product X\". However, the FD is free to deviate from this convention if desired by calling the 'setRegPath' function.

By storing default keys and values under the root key HKEY_CURRENT_USER, these keys and values will be stored on a per-user basis. For example, if a filter stores its default settings under HKEY_CURRENT_USER on a multi-user PC, each user will have his/her own private set of default settings. Again, the FD is free to change the root key from the default if desired, by calling the 'setRegRoot' function. The root key can be changed to HKEY_LOCAL_MACHINE if it is desired to save/store values on a machine-wide basis instead of a per-user basis. For example, the FD might choose to store registration information under HKEY_LOCAL_MACHINE instead of HKEY_CURRENT_USER if the filter is to be registered on a per-machine basis rather than a per-user basis.

1.2 Storing and retrieving data values

The core registry functions (put/getRegInt, put/getRegString) allow the FD to save and restore named integer and string values in the Windows Registry, thus providing a form of persistent storage across filter invocations.

Other functions (put/getRegData) provide for the storage of arbitrary data structures such as arrays and structs, once these are implemented in FM.

1.3 Enumerating keys and values

Registry functions are provided to allow the FD to blindly enumerate subkey names (enumRegSubKey) and value names (enumRegValue) beneath a given key. This is useful for iterating through named groups of preset values, recently-used file names, lists of installation directories for third-party products, or in general to explore and enumerate entire subtrees within the Windows Registry.

1.4 Deleting registry keys and values

Finally, functions are also provided to delete keys and values. The FD can use these functions, for example, to delete named groups of user settings or to clear out the filter's Registry information if the filter is being uninstalled.

1.5 Cautions and warnings

A word of caution: By giving the user Write and Delete access to the Windows Registry, FM gives the FD the potential to cause great harm to a user's system. The default key paths provided by FM have been designed to reduce (but not prevent) the possibility of causing widespread damage to the registry. Please use these functions wisely and responsibly, and test your filters well before distributing them!

WARNING: Always make a backup copy of your Windows Registry before trying to test and debug any new registry access code!

2. Registry Access Functions

The general format of each registry access function is:

```
iErr = <verb>Reg<what>( args... );
```

where iErr is an integer error code returned by the function. A value of 0 (ERROR_SUCCESS) indicates success; any other value (see section 2.5) indicates a failure. <verb> indicates the action to be performed ('set', 'get', 'put', 'enum', or 'delete'), and <what> indicates the object of the action ('Root', 'Path', 'SubKey', 'Int', etc.).

Some of the registry access functions take a variable list of arguments:

```
iErr = <verb>Reg<what>( ..., szName [, vararg]... );
```

In such cases, the variable (optional) arguments ("varargs") are used as input to apply printf-style formatting to the preceding string argument ("szName"). This aids in dynamically creating names of keys and values such as "ctl(<n>)" where <n> is to be replaced at run time by an actual decimal integer index. Any argument subject to such printf-style formatting will also be subject to the standard FM !-code interpolation; i.e., "!A" will be replaced with the current value of the FM built-in string 'fmFilterAuthor', "!C" will be replaced with the value of 'fmFilterCategory', etc. Such !-code interpolation is always performed *after* any printf-style formatting, at the last possible moment before actual use, so that the most current value of the corresponding built-in string is always applied.

2.1 Setting the default registry key path

This group of registry access functions is used to determine the current registry key, which consists of one of the predefined "root" keys followed by a set of subkeys, separated by backslashes ('\'). Remember that in C-style character strings, a single backslash must be represented by the pair "\\".

Functions setRegRoot and getRegRoot are used to set or retrieve the current registry root key:

```
iErr = setRegRoot(int hkey);
iErr = getRegRoot(int *hkey);
```

where hkey is:

HKEY_LOCAL_MACHINE for the root of the registry subtree that contains keys and values global to the entire PC, or
HKEY_CURRENT_USER for the root of the registry subtree that contains keys and values which are local to the currently logged-on user.

There are other possible predefined registry roots, but the ones listed above are the only two that the FD will usually be concerned with.

When FM starts up, it sets the default root key to HKEY_CURRENT_USER.

Functions setRegPath and getRegPath are used to set or retrieve the current registry subkey path beneath the current root key:

```
iErr = setRegPath(lpsz szPath[, varargs]...);
iErr = getRegPath(lpsz szPath, int maxPathLen);
```

where:

szPath	is a character string used to set or retrieve the current registry path.
maxPathLen	is the size of the szPath buffer in bytes, which specifies the longest path character string that can be stored in szPath, including the terminating null character.
varargs	is a list of optional arguments used to perform printf-style formatting on the szPath string [setRegPath only].

When FM starts up, it sets the default path string to "Software\\!O\\!C\\!t", which will expand to "Software\\<Organization>\\<Filter-category>\\<Filter-title>" as described in the Overview section.

If the length of the path to be retrieved (including the terminating null) is greater than maxPathLen, the path name will be truncated to fit into the szPath buffer, and getRegPath will return a nonzero error value. (A suitable value for maxPathLen to ensure that the path will always fit is MAX_PATH+1.)

2.2 Accessing registry data

This group of registry access functions is used to store and retrieve named data values from the registry, and might be the only set of registry functions required by the FD if the default registry path set by FM is suitable.

The Windows Registry recognizes several specific data types, and attempts to treat them in a host-independent manner. FM provides direct support for the following three Registry data types:

- 32-bit integers (FM treats these as signed, while the Registry treats them as unsigned)
- null-terminated character strings
- sequences of arbitrary byte values

(Several other data types are recognized by the Windows Registry, but not supported by FM.)

For each supported data type, there is a "putReg<type>" function to store a value of that type into the registry, and a corresponding "getReg<type>" function to retrieve a value of that type from the registry. The last argument(s) of each function specifies the name of the value to be stored or retrieved. The optional trailing arguments are used to apply printf-style formatting to the value name.
Value names may contain FM !-codes, which will be expanded just before use.

2.2.1 Integer data

Functions getRegInt and putRegInt are used to fetch or store 32-bit integer values:

```
iErr = putRegInt(int iValue, lpsz szValueName[, varargs]...);
iErr = getRegInt(int *piValue, lpsz szValueName[, varargs]...);
```

where:

iValue	is the 32-bit integer value to be stored in the registry
piValue	is the address of a 32-bit variable which will receive a value retrieved from the registry
szValueName	is the name of the value to be set or retrieved, and may contain printf-style formatting codes as well as FM !-codes.
varargs	is a list of optional arguments used to perform printf-style formatting on the szValueName string.

2.2.2 Character string data

Functions getRegString and putRegString are used to fetch or store C-style character strings:

```
iErr = putRegString(lpsz szString, lpsz szValueName[, varargs]...);
iErr = getRegString(lpsz szString, int iMaxlen, lpsz szValueName[, varargs]...);
```

where:

szString	is the address of the string to be stored or retrieved (for putRegString, FM requires that strlen(szString) <= 2048 in order to prevent inadvertent abuse of the registry)
iMaxLen	is the size of the szString buffer in bytes, which specifies the longest character string that can be stored in szString, including the terminating null character [getRegString only]
szValueName	is the name of the string to be set or retrieved, and may contain printf-style formatting codes as well as FM !-codes.
varargs	is a list of optional arguments used to perform printf-style formatting on the szValueName string.

2.2.3 Byte sequence data (arrays and structures)

Functions getRegData and putRegData are used to fetch or store arbitrary sequences of bytes, which is useful for storing an entire array or structure in FM (as long as host-independence is not a requirement):

```
iErr = putRegData(void *pArrayOrStruct, int dataLen, lpsz szValueName[, varargs]...);
iErr = getRegData(void *pArrayOrStruct, int dataLen, lpsz szValueName[, varargs]...);
```

where:

pArrayOrStruct	is the address of the array or structure to be stored or retrieved
dataLen	is the exact size of the pArrayOrStruct buffer in bytes (to prevent abuse of the registry, FM requires that dataLen <= 2048)
szValueName	is the name of the data to be set or retrieved, and may contain printf-style formatting codes as well as FM !-codes.
varargs	is a list of optional arguments used to perform printf-style formatting on the szValueName string.

2.2.4 Other data types

There is no direct registry support for other FM data types, such as bool, float, and double, since these data types are not support by the Windows Registry in a host-independent manner. However, if you are not concerned with host-independence, you can use get/putRegData for these data types. For example, to save and restore a double value in the registry, you can use:

- double dVal;
- putRegData(&dVal, 8, "Double Val");
- getRegData(&dVal, 8, "Double Val");

If you require a host-independent representation for a non-supported data type, you can use sprintf/sscanf to translate the data to/from a character string representation (possibly with some loss of precision). For example, you can save and restore a double-precision floating-point value in a host-independent manner with the following code (note that error checking has been omitted):

```
double dVal;

// save double value as a character string with 12 digits of precision...
sprintf(str0, "%.12g", dVal);
putRegString(str0, "Double String");

// retrieve double value represented as a character string...
getRegString(str0, 256, "Double String");
sscanf(str0, "%lg", &dVal);
```

2.3 Enumerating Keys and Values

This group of registry access functions is used to enumerate all subkey names and value names beneath the current registry key. Names are accessed by a zero-based index; the index is incremented by one until an error value is returned. Names are returned in no particular order; in particular, do not assume that names will be returned in alphabetical order. If the registry is modified (either by your process or by another process) at the current key level while an enumeration is in progress, results are unpredictable.

To enumerate subkey names beneath the current key:

```
iErr = enumRegSubKey(int index, lpsz szSubKey, int maxSubKeyLen);
```

where:

index	is the zero-based index for the next subkey name to be retrieved
szSubKey	is a character string buffer to receive the subkey name
maxSubKeyLen	is the size of the szSubKey buffer in bytes (including room for the terminating null character)

To enumerate the names of all values stored under the current key:

```
iErr = enumRegValue(int index, lpsz szValueName, int maxValueNameLen, int *piType, int *pcbData);
```

where:

index	is the zero-based index for the next value name to be retrieved
szValueName	is a character string buffer to receive the value name
maxValueNameLen	is the size of the szValueName buffer in bytes (including room for the terminating null character)
piType	is the address of an int variable which will receive a code indicating the type of data associated with this value (REG_DWORD for 32-bit integer data, REG_SZ for character string data, REG_BINARY for byte sequence data)
pcbData	is the address of an int variable which will receive the size in bytes of the data associated with this value

In general, you would perform a complete enumeration of all subkeys or values beneath the current key by setting the index to 0, then repeatedly calling enumRegSubKey or enumRegValue and incrementing the index until an error value (probably ERROR_NO_MORE_ITEMS) is returned.

2.4 Deleting Keys and Values

This group of registry access functions is used to delete subkeys or values beneath the current registry key. On Windows 95, you can delete an entire subtree at once by deleting the subkey at the root of the subtree. However, on Windows NT you cannot delete a subkey if the subkey has values or further subkeys beneath it; you must recursively delete all subkeys and values before you can delete the subkey at the root of a subtree.

To delete a subkey beneath the current registry key:

```
iErr = deleteRegSubKey(lpsz szSubKey[, varargs]...);
```

where:

szSubKey	is the name of the subkey to be deleted, and may contain printf-style formatting codes as well as FM !-codes.
varargs	is a list of optional arguments used to perform printf-style formatting on the szSubKey string.

To delete a value stored beneath the current registry key:

```
iErr = deleteRegValue(lpsz szValueName[, varargs]...);
```

where:

szValueName	is the name of the value to be deleted, and may contain printf-style formatting codes as well as FM !-codes.
varargs	is a list of optional arguments used to perform printf-style formatting on the szValueName string.

2.5 Error Return Values

The registry access functions can return the following values (among others).
A value of 0 (ERROR_SUCCESS) indicates success; any other value indicates a failure of some sort.

symbolic name	*meaning*
ERROR_SUCCESS	(==0) no error
ERROR_FILE_NOT_FOUND	key or value name not found
ERROR_MORE_DATA	buffer wasn't big enough (e.g., getRegString, getRegData)
ERROR_NO_MORE_ITEMS	index >= # of values or subkeys (enumRegValue, enumRegSubKey)
ERROR_INVALID_FUNCTION	bad top-level key, etc.
ERROR_INVALID_DATA	wrong data type or size (or size > 2048)
ERROR_BADDB	registry database is corrupt
ERROR_BADKEY	registry key is invalid
ERROR_CANTOPEN	registry key could not be opened
ERROR_CANTREAD	registry key could not be read
ERROR_CANTWRITE	registry key could not be written
ERROR_REGISTRY_CORRUPT	registry is corrupt
ERROR_REGISTRY_IO_FAILED	input/output to registry failed
ERROR_KEY_DELETED	Illegal operation attempted on a Registry key which has been marked for deletion.
ERROR_KEY_HAS_CHILDREN	cannot delete a key with subkeys (Windows NT)

2.6. Registry data type codes

The Windows Registry defines the following data type codes. Only three of these data types are recognized by FM: REG_SZ, REG_BINARY, and REG_DWORD.

symbolic code name	*value*	*FM type*	*Windows registry type*
REG_NONE	(0)	none	No value type
REG_SZ	(1)	char*	Unicode nul terminated string
REG_EXPAND_SZ	(2)	n/a	Unicode nul terminated string (with environment variable references)
REG_BINARY	(3)	array or struct	Free form binary
REG_DWORD	(4)	signed int	32-bit unsigned number
REG_DWORD_LITTLE_ENDIAN	(4)	n/a	32-bit number (same as REG_DWORD)
REG_DWORD_BIG_ENDIAN	(5)	n/a	32-bit number (MSB stored first)
REG_LINK	(6)	n/a	Symbolic Link (Unicode)
REG_MULTI_SZ	(7)	n/a	Multiple Unicode strings
REG_RESOURCE_LIST	(8)	n/a	Resource list in the resource map
REG_FULL_RESOURCE_DESCRIPTOR	(9)	n/a	Resource list in the hardware description

2.7. Random Notes

Some final notes:

1. The default (unnamed) value associated with a key can be accessed by specifying a null string ("") for the value name (szValueName).

2. Registry path names and value names are case-insensitive.

3. FM restricts the maximum data length that can be stored as a single value to 2048 bytes to discourage abuse of the registry.

4. The maximum possible size of a registry path string, not counting the terminating null character, is MAX_PATH.

5. getReg<datatype>() will return a zero or null data value (e.g., 0 or "") if an error occurs (e.g., if the specified value name is not found under the current key).

3. Examples

This section presents actual code samples, showing typical uses for the registry access functions. These examples can serve as a rich source of code "snippets" for writing your own registry access code.

Caution: At present, not all of the following examples have been fully tested. Some samples may also make use of features which are not yet implemented in the current version of FM.

3.1 Saving and retrieving simple data values

The following examples show how to save and restore simple data values to/from the Windows Registry under the default registry key path. In many cases, this is all your filter will require in order to preserve state across invocations.

3.1.1 Save and restore a *single integer parameter*

```
int iRadius;
   ...
putRegInt(iRadius, "Radius");   //Save value of iRadius
   ...
getRegInt(&iRadius, "Radius");  //Retrieve value of iRadius
```

Note that iRadius is passed "by value" in the call to putRegInt, whereas it is passed "by reference" in the call to getRegInt (i.e., use the &-operator to pass the address of iRadius instead of the value when calling getRegInt).

3.1.2 Save and restore a *set of 8 integer controls* (0 - 7)

This example uses for-loops and printf-style formatting to save or restore an entire set of 8 integer control values, using value names "ctl(0)", "ctl(1)", ..., "ctl(7)". When retrieving the control values, we check for an error (probably ERROR_FILE_NOT_FOUND, meaning a value has not yet been stored with this name); if an error occurs, we set the control to a specific default value (100) instead of using the default value (0) set by getRegInt whenever an error occurs.

```
//save ctl(0) through ctl(7)...
int i;
for (i=0; i<=7; i++) putRegInt(ctl(i), "ctl(%d)", i);

//restore ctl(0) through ctl(7), using a default value if the
//registry entry is missing...
int i, iVal;
for (i=0; i<=7; i++) {
   if (getRegInt(&iVal, "ctl(%d)", i) == ERROR_SUCCESS) {
      // value obtained from registry
      setCtlVal(i, iVal);
   } else {
      // use a default value of 100...
      setCtlVal(i, 100);
   }
}
```

3.1.3 Saving and retrieving *character strings*

To save a title for your filter dialog box, and then retrieve it on a subsequent invocation:

```
putRegString("Filter #2", "Dialog title");  //save title for next time
   . . .
getRegString(str0, 256, "Dialog title");    //retrieve saved title
if (strcmp(str0, "")==0) {
   //title is null or missing...
   strcpy(str0, "Default title");  //set a default title
}
setDialogText(str0);   //set title for our filter dialog
```

To save and restore the dropdown list for a ComboBox control:

```
//get dropdown list contents from control CTL_CB...
getCtlText(CTL_CB, str0, 256);  //warning: max of 255 list chars!
putRegString(str0, "Dropdown List");    //save list in registry
  …..

//retrieve dropdownlist from registry...
//(returns null string if missing)
getRegString(str0, 256, "Dropdown List");
setCtlText(CTL_CB, str0);   //set dropdown list for control
```

Note: This example calls "getCtlText", which is not yet implemented in the current versions of FM.

3.1.4 Saving and retrieving *arrays and structures*

To save and restore an entire array or structure to/from the registry as a sequence of binary bytes (NOT host-independent!):

```
struct {
   char first_name[32];
   char middle_initial;
   char last_name[32];
   int age;
   float salary;
} person;

int gamma_vals[256];

//save and restore the 'person' structure...
putRegData(&person, sizeof(person), "Person");
   ...
getRegData(&person, sizeof(person), "Person");

//save and restore the 'gamma_vals' array...
putRegData(gamma_vals, sizeof(gamma_vals), "Gamma Values");
   ...
getRegData(gamma_vals, sizeof(gamma_vals), "Gamma Values");
```

Notes:
- Structures, arrays, and the sizeof()-operator are not yet implemented in the current version of FM.
- In FM, as in C, array names are automatically promoted to the address of the array, so it is not necessary to preface "gamma_vals" with the &-operator in the calls to putRegData and getRegData.

In order to prevent abuse of the Windows Registry, the maximum amount of data that can be stored by a call to putRegData is 2048 bytes (as recommended by Microsoft). If you really need to save an array larger than this, you could use something like the following code (though this definitely borders on abuse, and is presented here as an instructional example only!):

```
double mydata[500];       //4000 bytes of data
int i;

for (i=0; i < 500; i++) {
   //store array element i as "MyData[i]"...
   putRegData(&mydata[i], "MyData[%d]", i);
}
```

3.2 Setting a non-default registry key path

Assume that your filter's %ffp program contains the following definitions:

```
Category:      "AFH Filter Pack 1"
Title:         "Zap White..."
Version:       "2.0"
Author:        "Alex Hunter"
Organization:  "AFH Systems"
```

The default registry key root (HKEY_CURRENT_USER) and path ("Software\\!O\\!C\\!t") will expand to:

```
HKEY_CURRENT_USER\Software\AFH Systems\AFH Filter Pack 1\Zap White
```

which is where your registry values will be stored by default.

If you want your registry values to be stored on a user-independent basis (i.e., global to the entire PC), then change the root key by calling:

```
setRegRoot(HKEY_LOCAL_MACHINE);
```

Your registry values will now be stored by default under the key:

```
HKEY_LOCAL_MACHINE\Software\AFH Systems\AFH Filter Pack 1\Zap White
```

While "Software\\!O\\!C\\!t" is the path recommended by Microsoft for storing your product's registry data, you may prefer to choose some other path. For example, to use your Author: name instead of Organization: as the 2nd=level path component, call:

```
setRegPath("Software\\!A\\!C\\!t");
```

In our example, this will cause your registry values to be stored under:

```
HKEY_CURRENT_USER\Software\Alex Hunter\AFH Filter Pack 1\Zap White
```

If you want your settings to be unique to a specific version of your product (i.e., filter pack), you might call:

```
setRegPath("Software\\!O\\!C\\!V\\!t");
```

which, in our example, will cause registry values to be stored under:

```
HKEY_CURRENT_USER\Software\AFH Systems\AFH Filter Pack 1\2.0\Zap White
```

If you want your settings to be specific to a particular host application (e.g., you want separate settings for Adobe Photoshop vs. Paint Shop Pro),
you can call:

```
setRegPath("Software\\!O\\!C\\!H\\!t");
```

which in our example, assuming the current host is Paint Shop Pro, will store the settings under:

```
HKEY_CURRENT_USER\Software\AFH Systems\AFH Filter Pack 1\Paint Shop Pro(tm)\Zap White
```

As you can see, the possible variations are endless. But, in order to maintain some order and consistency within the Windows Registry, please adhere to at least the following minimum standards:

1. The top-level component of the path should always be "Software".
2. The second-level component should distinguish your product in some unique way from all other vendor's products. I.e., use your Organization name ("!O") or Author name ("!A") as the second path component
3. Beyond that (third-level and below), you are relatively free to choose your own registry path structure

Other notes:

-To set the registry key and path back to the FM default values:

```
setRegRoot(HKEY_CURRENT_USER);
setRegPath("Software\\!O\\!C\\!t");
```

-You don't need to use the !-codes in calls to setRegPath; you can hardcode the actual string values if you prefer, such as:

```
setRegPath("Software\\My Company\\My Filter Pack\\My Filter");
```

However, using the !-codes makes it easier to maintain your code across changes to filter names, version numbers, etc.; and makes it easier to copy-and-paste code snippets from one filter to another without having to edit all the hardcoded string constants.

3.3 To restore the values of controls 0-7 upon initial entry and save them upon exit

This example demonstrates how to use the OnFilterInit and OnFilterExit handlers to automatically save the values of controls 0-7 whenever your filter exits, and then to restore the previously saved values upon the next invocation. The first time the filter is invoked, no previously saved values exist, so the controls are allowed to retain their initial default values.

```
OnFilterInit:{
    int i, iVal;
    for (i=0; i<=7; i++) {
        if (getRegInt(&iVal, "ctl(%d)", i) == ERROR_SUCCESS) {
            // control value obtained from registry
            setCtlVal(i, iVal);
        }
    }
    return false;
}


OnFilterExit:{
    int i;
    for (i=0; i<=7; i++) {
        // save control value in registry
        putRegInt(ctl(i), "ctl(%d)", i);
    }
    return false;
}
```

3.4 To populate a ComboBox control with subkey names for control presets

In this example, we assume that named groups of control settings have been stored under the registry path "Software\\!O\\!C\\!t\\Presets". Each group of presets is identified by a unique subkey under this path, and the control values for a particular group of presets are in turn stored beneath the subkey name for that group.

Groups of presets may have been predefined by the filter when it was installed, or added and deleted by the end user at run time.

The following code retrieves the names of all currently defined preset groups and presents them as a list in a dropdown ComboBox with index CTL_CB. The user can then select a particular preset group from this dropdown list.

```
int iRetVal;

setCtlText(CTL_CB, ""); //clear the ComboBox control
//set registry path to "Software\\!O\\!C\\!t\\Presets"
getRegPath(str9, 256);   //save current registry path
setRegPath("%s\\Presets", str9);    //move down to "Preset" subkey
for (i=0, iRetVal=ERROR_SUCCESS; iRetVal == ERROR_SUCCESS; i++) {
   iRetVal = enumRegSubKey(i, str0, 256);
   if (iRetVal == ERROR_SUCCESS) {
      //got the name of a preset group...
      insertCtlText(CTL_CB, str0);
   }
}
setRegPath(str9);    //restore registry path to original value
```

Note that this example calls a hypothetical FM built-in function "insertCtlText", which is not yet implemented in the current version of FM. Alternatively, you could build up a list of all preset names, separated by '\n' characters, in a global string (say str1) and then call setCtltext(CTL_CB, str1) to set the list of preset names all at once.

3.5 To retrieve a set of control presets saved under key ...\Presets\<subkey>

Continuing the previous example, once the user has selected a preset group name from the ComboBox control, you can retrieve the preset values for controls 0 through 7 and the preset dialog title as follows (where str0 contains the selected preset subkey name):

```
// Retrieve control presets given by subkey str0...
// set registry path to "Software\\!O\\!C\\!t\\Presets\\<subkey>"
getRegPath(str9, 256);   //save current registry path
setRegPath("%s\\Presets\\%s", str9, str0); //move down to Presets\<subkey>
// get and set the 8 integer control values...
for (i=0; i < 8; i++) {
   int iVal;
   getRegInt(&iVal, "ctl(%d)", i);
   setCtlVal(i, iVal);
}
// get and set the dialog title...
getRegString(str1, 256, "Title");
setDialogText(str1);
setRegPath(str9);    //restore registry path
```

3.6 To Delete a group of presets

In this example, we again assume that presets are stored in the registry as groups of named values under key:

```
<default-key>\Presets\<preset-group-name>
```

where <default-key> is the usual product path (HKEY_CURRENT_USER\Software\!O\!C\!t), and <preset-group-name> is the user-defined name for the group of presets to be deleted.

Note that we take care to delete all values beneath the <preset-group-name> subkey before deleting the subkey itself. On Windows 95, it might suffice to delete the subkey directly; but under Windows NT, it is necessary to first delete all values belonging to the subkey before deleting the subkey itself.

```
   int iRetVal;
//global string usage:
// str0 - name of preset group
```

```
// str1 - value name
// str9 - save/restore current reg path

   //assume name of preset group is in str0...
   getRegPath(str9, 256);   //save current registry path
   setRegPath("%s\\Presets\\%s", str9, str0);
   //repeatedly delete the first value under this preset,
   //until no more values remain...
   do {
      //get the 0-th value name in str1...
      iRetVal = enumRegValue(0, str1, 256);
      if (iRetVal == ERROR_SUCCESS) {
         //delete this value if it exists
         deleteRegValue(str1);
      }
   } while (iRetVal == ERROR_SUCCESS);
   //finally, climb up the path one level and
   //delete the preset group name subkey itself...
   setRegPath("%s\\Presets", str9);
   deleteRegSubKey(str0);
   setRegPath(str9);    //restore registry path
```

3.7 To get the local computer name

In addition to your filter's private registry information, the Windows Registry contains all sorts of other useful (and useless) information. This example shows how to retrieve the network name of your PC, which is stored as a string value named "ComputerName" under the registry key HKEY_LOCAL_MACHINE\System\CurrentControlSet\control\ComputerName\ComputerName:

```
   setRegRoot(HKEY_LOCAL_MACHINE);
   setRegPath("System\\CurrentControlSet\\control\\ComputerName\\ComputerName");
   getRegString(str0, 256, "ComputerName");
   Info("The name of your computer is %s", str0);
```

3.8 To find Photoshop installation directories (4.0 and later)

Adobe Photoshop version 4.0 (?) and later stores the application installation directory path in the Registry under key:

```
   HKEY_LOCAL_MACHINE\Software\Adobe\Photoshop\x.x\ApplicationPath
```

and the plug-in directory path under key:

```
   HKEY_LOCAL_MACHINE\Software\Adobe\Photoshop\x.x\PluginPath
```

where 'x.x' is the Photoshop version number.

The following code finds all installed versions of Photoshop (4.0 and later), and displays the application directory and plug-in path for each version.

```
 { //enumerate Photoshop installation directories
  int saveRootKey;
  int iRetVal;
  int i;
  //global string usage:
  // str0 - subkey name (PS version # "x.x")
  // str1 - app directory
  // str2 - plugin directory
```

```
 // str9 - save/restore current reg path

 //save registry root and path...
 getRegRoot(&saveRootKey);
 getRegPath(str9, 256);
   setRegRoot(HKEY_LOCAL_MACHINE);
   for (i=0, iRetVal=ERROR_SUCCESS; iRetVal == ERROR_SUCCESS; i++) {
      setRegPath("Software\\Adobe\\Photoshop");
      iRetVal = enumRegSubKey(i, str0, 256);
      if (iRetVal == ERROR_SUCCESS) {
           //found an installed version of Photoshop...
         Info("Installed version = %s", str0);
         setRegPath("Software\\Adobe\\Photoshop\\%s\\ApplicationPath", str0);
         if (getRegString(str1, 256, "") == ERROR_SUCCESS) {
            Info("Application directory for version %s is %s", str0, str1);
         }
         setRegPath("Software\\Adobe\\Photoshop\\%s\\PluginPath", str0);
         if (getRegString(str2, 256, "") == ERROR_SUCCESS) {
            Info("Plugin directory for version %s is %s", str0, str2);
         }
      }
   }//for
 //restore registry root and path...
 setRegRoot(saveRootKey);
 setRegPath(str9);
}
```

Note that the application directory and plug-in path are stored as the default (unnamed) value under each respective key. To retrieve this unnamed value, call getRegString with a null string ("") as the name of the value.

3.9 To enumerate Paint Shop Pro plug-in directories

JASC Paint Shop Pro version 5 (?) and later stores the current user's plug-in directory paths as string values named "Directory1", "Directory2", and "Directory3" under Registry key:

HKEY_CURRENT_USER\Software\JASC\Paint Shop Pro #\ImageProcessingFilter

where "#" is the Paint Shop Pro version number.

The following code finds all installed versions of PSP (5 and later) for the current user, and displays the plug-in directory paths for each version found.

```
{ //enumerate Paint Shop Pro plug-in directories under
   //HKEY_CURRENT_USER\Software\JASC\Paint Shop Pro #\ImageProcessingFilter\Directory#
 int saveRootKey;
 int iRetVal;
 int i, j;
 //global string usage:
 // str0 - product name and version #
 // str1 - plug-in directory
 // str9 - save/restore current reg path

 //save registry root and path...
 getRegRoot(&saveRootKey);
 getRegPath(str9, 256);
   setRegRoot(HKEY_CURRENT_USER);
   for (i=0, iRetVal=ERROR_SUCCESS; iRetVal == ERROR_SUCCESS; i++) {
      setRegPath("Software\\JASC");
      iRetVal = enumRegSubKey(i, str0, 256);
```

```
      if (iRetVal==ERROR_SUCCESS && strncmp(str0, "Paint Shop Pro", 14)==0) {
          //found an installed version of PSP...
        Info("Installed product = %s", str0);
        setRegPath("Software\\JASC\\%s\\ImageProcessingFilter", str0);
          //display all 3 plug-in directory paths...
        for (j = 1; j <= 3; j++) {
          getRegString(str1, 256, "Directory%d", j);
          Info("Plug-in directory %d = \"%s\"", j, str1);
        }//for j
      }//if
   }//for i
 //restore registry root and path...
 setRegRoot(saveRootKey);
 setRegPath(str9);
}
```

3.10 To enumerate Adobe Registration Info for the current user

When you register an Adobe product, a variety of information is saved in the Windows Registry under the key:

HKEY_CURRENT_USER\Software\Adobe\Registration\User

The following code enumerates and displays all the Adobe registration info for the current user (which may contain information you did not expect to make public!):

```
{ //enumerate Adobe Registration Info under
 //HKEY_CURRENT_USER\Software\Adobe\Registration\User
 int saveRootKey;
 int iRetVal;
 int iDataType;
 int iDataLen;
 int iDataValue;
 int i;
 //global string usage:
 // str0 - value name
 // str1 - value data (string)
 // str9 - save/restore current reg path

 //save registry root and path...
 getRegRoot(&saveRootKey);
 getRegPath(str9, 256);
   setRegRoot(HKEY_CURRENT_USER);
 setRegPath("Software\\Adobe\\Registration\\User");
      //enumerate and display each value under this key...
   for (i=0, iRetVal=ERROR_SUCCESS; iRetVal == ERROR_SUCCESS; i++) {
      iRetVal = enumRegValue(i, str0, 256, &iDataType, &iDataLen);
      if (iRetVal == ERROR_SUCCESS) {
  switch (iDataType) {
  case REG_SZ:
   //value is a string
   getRegString(str1, 256, str0);
   Info("(%d) %s = \"%s\"", i, str0, str1);
   break;
  case REG_DWORD:
   //value is 32-bit integer
   getRegInt(&iDataValue, str0);
   Info("(%d) %s = %d", i, str0, iDataValue);
   break;
```

```
   case REG_BINARY:
    //value is sequence of n bytes...
    Info("(%d) %s = sequence of %d bytes", i, str0, iDataLen);
    break;
   default:
    //some other data type...
    Info("(%d) %s = some other data type (%d)", i, str0, iDataType);
    break;
   }//switch
       }//if
    }//for
 //restore registry root and path...
 setRegRoot(saveRootKey);
 setRegPath(str9);
}
```

Note that the only registry data types officially recognized by FM are REG_SZ (null-terminated character string), REG_DWORD (32-bit integer), and REG_BINARY (sequence of binary data bytes).

3.11 To check that a filter has been registered for use on this machine

The FM registry access functions can be used to read values stored in the Windows Registry by your filter's installation program. For example, assume that as a form of (weak) copy protection, the installation program stores a magic value 'szMagicString' as a string named "Regcode" under the key:

```
HKEY_LOCAL_MACHINE\Software\!O\!C\!t
```

Your filter can check this registration code at run time with the following code. (I'm sure you can come up with stronger protection schemes than this -- this example is only illustrative.)

```
setRegRoot(HKEY_LOCAL_MACHINE); //registration code is global for this machine
// Registration code was entered in the Windows Registry by
// the filter installation program...
getRegString(str0, 256, "Regcode");
// Test for valid registration code...
// (If not present in Registry, the value returned will be "".)
if (strcmp(str0, szMagicString) != 0) {
   //incorrect registration code...
   ErrorOK("Filter not registered on this machine!");
   abort();
}
```

Release Notes for FilterMeister 1.0.0

Alex Hunter, Harald Heim
AFH Systems
1 April 2003

Following are the incremental release notes for FM Beta version 1.0.0.
Unless otherwise noted, the release notes for versions 0.4.19 and earlier (see below) also apply to version 1.0.0.

1. New Features

1.3 New Color Modes:

- Lab 48 bit
- CMYK 64 bit
- DeepMultichannel
- Duotone 16 bit

1.4 New Functions:

- pget and pgetr support 16 bit images
- for registry access
- for color space conversion (YUV, Lab, HSL, YCbCr)
- iget() for interpolated image access with 5 methods (nearest, bisquare, bicosine, bilinear, bicubic)
- srcp(), pgetp() etc. for accessing a whole pixel at once. Using these functions lets you read and write image data almost double as fast as with src() and pset().
- for drawing rectangles, circles and triangles easily
- for locking the window redraw and for redrawing the whole window, a control or a region
- for setting the preview zoom
- for reading the coordinates of the mouse pointer above preview
- for using up to 10 arrays with one, two or three dimensions
- for performing 20 different blending operation with two image sources
- for drawing a dozen two-dimensional gradients
- for edge-wraping values
- for checking if keys were pressed
- for getting the current Window Version, the screen resolution and screen refresh rate.
- for triggering up to 10 different timer events e.g. to perform multi-threading-like activities.

1.5 New Events:

- for asynchonous activities (timer)
- indicating left and right clicks on the preview (leftclicked_up, leftclicked_down, rightclicked_up, rightclicked_down)
- indicating a mouse movement above the preview (mousemove)

2. Changes

2.1 Dialog Changes

- Larger main dialog
- 200% larger preview
- system color is used for the dialog background
- system color is used for slider labels and statictext controls
- The editor window is now automatically displayed when you start FilterMeister
- The Edit >>> button was removed and instead a Minimize button was added to the title bar of the editor
- When disabling a standard slider control, the slider label is now disabled, too.

2.2 Code Buffers

Larger code buffer which means that filters with a larger and more complex code can be created:
- onCtl: 64K -> 128K
- ForEveryTile: 128K -> 256K

3. Known Problems

Appendix:
Detailed Notes about the new features

For detailed descriptions of the new function, please have a look at the **"New in Version 1.0"** page of the FilterMeister help files which can be displayed by clicking on the Help button in FilterMeister.