FilterMeister

Book One - Getting Started

Book Two - User Guide

Book Three - Reference Manual

Table of Contents

1. Introduction

We welcome you to FilterMeister by AFH Systems Group! FilterMeister is a plug-in for Adobe Photoshop and plug-in compatible programs (such as Corel PhotoPaint, JASC Paint Shop Pro, Macromedia Freehand, etc.) with which you can create your own astounding filter effects such as fractals, explosions, 3D-effects, and lots more! *Currently, FilterMeister is available only for the Windows 95/NT Operating Systems.*

The entire plug-in window is fully configurable to meet your needs. You can:

* use sliders, checkbuttons and other user controls for user interaction,
* load background bitmap images,
* use radio buttons and pull-down menus to select predefined settings or multiple-filters in one plug-in,
* save and load image or settings data to and from disk,

and lots more!

In FilterMeister (short: FM), you will be working with the FF+ (FF plus) language, which is a superset of the Adobe Filter Factory (FF) language. FF+ is a subset of the C language, with extensions that include:

- additional built-in variables and functions for image processing and filter design
- additional syntax to describe the design of the User Interface
- predefined event handlers, which are special purpose functions for processing specific filter events and User Interface actions

You won't need advanced programming skills, since we will be introducing the language to you bit by bit. With the FilterMeister package you received three manuals in the PDF file format:

```
GettingStarted.pdf
UserGuide.pdf
ReferenceManual.pdf
```

In this manual, we will introduce you to the FilterMeister's programming language: FF+. You will learn how a filter program is built with the necessary handlers, variables, operators and functions. We will then teach you how to customize your own filter dialog. In the following chapter, the navigation in the editing window and the creation of stand-alone filters are described. Finally, there are some limitations you ought to know such as maximum filter program size, host support, etc.

We strongly recommend that you read the **Getting Started** manual first, where FilterMeister installation and first filters are described. The FM language is so powerful that we can't teach you all the possibilites, functions, variables, handlers, etc. This is where you can consult the **Reference Manual** to learn the elements not discussed in the first two manuals.

2. The FF+ Language

This chapter will introduce you to the powerful language used in FilterMeister: FF+ (pronunciated ”eff eff plus”). We will begin with the different possible structures an FM program can have. You will learn which head and body elements exist. Next, the image-related variables, constants and functions are presented. You will also need some knowledge of the C language programming techniques. Please note that we cannot teach you the complete C language. There are lots of good (and really fat) books which are predestinated for that matter. Nevertheless, feel free to post questions and ideas or problems you are having in our Mailing List. We are sure that you will get excellent support from the members of the Mailing List. Following chapters include tutorials that show some programming techniques with FM.

2.1 Program structures

A FilterMeister program is built up of two parts. The program head and the program body. Although it is possible to list head elements and the body elements in any order, we recommend the order presented in the following sections. These will help you build up a clear structure every FM programmer can understand. The clearer the program design and documentation, the easier people can help you, but also the faster you will understand your own program (after a long-time vacation, for example).

Program head

The program head describes the looks and feel of the filter. This is where the filtering does *not* take place (FM programmers compute with their bodies, the heads are for food and beverage input, really)! The program head can (but does not have to) include the following elements (in this recommended order):

– Filter identification keys (discussed in chapter 3.2)
– Dialog window properties (discussed in chapter 3.1)
– User controls and their properties (discussed in chapter 3.3)

The **filter identification** keys give information about the filter plug-in: Author, Copyright, Title, Category, Version, Filename and About box. This information is essential for the graphics program (at least it has to know where to place the filter plug-in). The **dialog window properties** describe the looks (or dialog window design) of your program. Finally, the **user controls** are defined so users can play a bit with different filter settings.

Program body

The program body is where all the filter action takes place. Picture it as if FilterMeister is the stomach, digesting and convoluting all that pizza and Coke into one big pudding... hm, that's a bad metaphor – forget that last part! Seriously, the progam body consists of:

– handlers

Yes, that's it (what did you expect)! **But**, there are various handlers you can use. You can use one of them, you can use different handlers in one FM program. It all depends on what the handlers' tasks are. These are the handlers you can use in FilterMeister programs:

– `ForEveryTile`
– `ForEveryPixel`
– `RGBA (or R,G,B,A)`

While the first two handlers are true FM handlers, the RGBA handler is based on the Adobe Filter Factory program structure. The handlers process the image differently. The `ForEveryTile` handler processes the image tile-by-tile, thus applying the filter code to each individual tile. The ForEveryPixel and RGBA handlers process each pixel individually, as discussed in chapter 2.4 of the Getting Started Manual.

If no handlers are present, the original image is copied onto itself. Actually, each handler returns a boolean value, either true or false. The handler that returns true is the one which will do the filtering. For example, if the `ForEveryTile` handler is set to return true, the `ForEveryPixel` handler will not process the image and all code in it is ignored.

ForEveryTile

While the other handlers process each individual pixel of the image, you can adress only a certain amount of pixels in the image. This feature can be very time-saving if you only want to create a border, for example. This handler can also be used for iterated filters, meaning that the code is applied again and again (accomplished with loops) based on the last results (the game Life is such an example).

Originally, the `ForEveryTile` handler was designed to change the information of the current tile. You have to understand that the host program (e.g., Adobe Photoshop) can send parts of the image (tiles) or the whole image information to the filter requesting image data. Sometimes, the whole image to be processed is too large to be loaded at once into memory, so the host "cuts" the image into tiles and sends them separately to the filter.

The reason for tiling of an image can also depend on certain keywords in an FM program. If, for example, you are processing the leftmost pixels while you need image data from the right side of the image (e.g., mirroring), tiling the image would not be very useful (since you have only access to the rightmost pixels of the tile).

The FM compiler will do the thinking for this problem. If it finds at least one src()-function or pset()-function in the program, it will automatically prepare the filter to accept only complete images. These two functions allow the FM filter to have access to all pixels in the image, not only the tile. We should only worry when a large image takes too much memory and the FM filter is not able to load a copy of it for filter application.

A very basic FM program which does absolutely nothing looks like the following:

```
%ffp

ForEveryTile:
{
                //This space could be filled with the filter code
return true;    //Do not let ForEveryPixel or RGBA take control
}

%%eof
```

As one can see, the handler is followed by a colon and the filter code is placed between the braces { }. The ForEveryTile handler *always* has to return a boolean value, either true or false. Note that all program lines end with a semicolon and remarks are set after a double slash, so the text is ignored until the end of the line. The last line tells FM that the filter program ends at that line (EOF = end of file). Everything is ignored after that so-called footer. This line is optional, though.

The `ForEveryTile` handler gets along well with the `ForEveryPixel`and RGBA handlers, too. It can be used as an animation controller which makes a loop and calls the filter in the `ForEveryPixel` or RGBA handler with different settings (we will show you one or two filter animation examples).

ForEveryPixel and RGBA

These handlers process each individual pixel in each specified channel from left-to-right and from top-to-bottom. The filter code used here is usually very compact – whenever possible, try to program your filter codes in one of these handlers. One does not need to include return function (it is not allowed, either). These handlers take over when the `ForEveryTile` handler is set to return false. Furthermore, these two pixel handlers are rivals: if these two are defined in a program, only the last pixel-based handler in the FM program will filter the image.

A basic RGBA structure looks like the following:

```
%ffp

R: r    //fill current pixel with red value of current pixel
G: g    //fill current pixel with green value of current pixel
B: b    //fill current pixel with blue value of current pixel
A: a    //fill current pixel with alpha value of current pixel
```

The above code simply replaces the current pixel color with its own color, thus leaving the image untouched. For example, the red channel element R is assigned to the variable r – in plain language: "give the current red color of the pixel the red color value of the pixel".

If all the codes of each channel element are equal, it is in some cases practical to use special variables (only in the RGBA-handler!) to save space and structure the program in a clearer way. For example, the above code could look like the following:

```
%ffp

R,G,B,A:  c
```

The variable c represents the color of the pixel, depending on which channel it is used in. The above method would save you the copy-and-paste time if you had to copy a code from one channel element and paste it on the others. Filter Factory programmers have developed tricks to create compact but also cross-element filter codes (i.e., codes which can be used in any R, G, B or A element).

It is possible to exchange variables (see chapter 2.2 for more details on variables):

```
%ffp

R: g    //apply to red component value of current pixel with green value of current pixel
G: r    //apply to green component value of current pixel with red value of current pixel
```

In the above case, the red and green color components of the current pixel are exchanged and a new visual effect has been accomplished. Note that it is not necessary to include the other two channel elements B: and A:, although we really recommend to include all channel components.

A basic `ForEveryPixel` structure looks like the following:

```
ForEveryPixel:
{
R = r;
G = g;
B = b;
A = a;
}
```

Note that is resembles the ForEveryTile program structure. The handler is followed by a colon, the filter code is typed in between braces { } and all lines end with semicolons. Unlike the ForEveryTile handler, no return false statement is needed (nor allowed). Unlike the RGBA handler, the ForEveryPixel does not support multiple channel element filter coding as in:

```
ForEveryPixel:
{
R, G, B, A = c;         //fill current pixel with color value of current pixel
}
```

According to the C programming rules (see comma operator in chapter 2.2), the variable c in the above program is assigned only to the last channel element A. The R, G and B channel elements are not changed. Note that the comma in a handler is used as a separator and we used it outside of the RGBA handler as a special case to assign the filter code to the four channel elements.

Certain variables allowed in the RGBA handler are prohibited to be used in the ForEveryTile and ForEveryPixel handlers. This is the case for the variables c and z (explained in chapter 2.2).

We have shown here the typical structures of the pixel-based handlers. Interesting is the fact that the channel elements can be ordered in any way. The first channel element in the handler is processed first. Although the following program does the same thing as the others in this chapter, the pixel's color component processing order is different.

```
%ffp

R:      r
B:      b
A:      a
G:      g
```

Also note that the channel processing is not recursive. This is important to know when making a channel code dependant from color variables from other channels. Let's say the red color component of a pixel was 50 and is changed according to the filter code to 100. The green color component is for example added to the red component. In this case, green is added to 50, the original color component value and not the recently changed value 100. This is meant by channel processing being not recursive.

Many programmers forget the alpha channel, which holds the transparency information of the image. Although most of the users work and apply filters to the background layer, it is sometimes essential to think of the transparency. Think of a filter which pans the only the color channels to the left in a layer and the alpha channel stays as it is. In this case, it is important to tell FM to pan the alpha channel, too.

2.2 Variables, Constants and Operators

Before we begin with the hands-on programming, you should know which variables and constants you can use and how they can be ”operated”(besides childish adding and subtracting, FM knows a variety of other operators – *no, we are not talking about multiplication nor division!* ;-)). We will start with a small reference and at the end of the chapter you will find some tutorials.

Constants

Constants are elements which do not change their value during the whole filtering process. Numbers, for example are constants, because the number 15, for example, will not change during the filtering process. There are some pseudo-constants which can vary from image to image, but stay constant during the filtering process. Example: an image's width never changes during the filtering process.

These are the pseudo-constants which can be used in FM programs:

Constant(s)	Description	Value range
X, xmax	image's width	varies
Y, ymax	image's height	varies
M	image's total magnitude (half the diagonal size of the image)	
D	total amount of angles within the image	1024
Z, inPlanes	total amount of channels *depends on whether FM is called from the background layer or a normal layer, respectively*	3 or 4
R, G, B, A, C, I, U, V	maximum value a channel can have	255
dmin, mmin	minimum values of angle (direction) and magnitude	0
true	equals 1	
false, NULL	equal 0	
RAND_MAX	the maximum value that can be returned by the `rand` function	32767
_MAX_DIR	maximum length of directory component in filename string	256
_MAX_DRIVE	maximum length of drive component in filename string	3
_MAX_EXT	maximum length of extension component in filename string	256
_MAX_FNAME	maximum length of filename component in filename string	256
_MAX_PATH	maximum length of full path in filename string	260

Note that the above constants and pseudo-constants return integer numbers. The following example presents an information window and returns the image's height and width (see chapter 2.4 for message windows):

```
%ffp

ForEveryTile:
{
Info("Image's metrics:\nHeight:\t%d\nWidth:\t%d", X, Y);        //Pop up window message box
return false;                            //Leave the filtering to the ForEveryPixel handler
}
```

Variables

Variables are alphanumerical terms which represent numerical values that can change during the filtering process. For example, the variable x changes from pixel to pixel when used in the RGBA and ForEveryPixel handlers. In addition to the predefined variables in FM, new variables of different types can be defined (for example as counter variables in loops).

These are the image-based variables used in FilterMeister programs supported only the ForEveryPixel and RGBA handlers:

Variable(s)	Description	Value range
r, g, b, a	specify the red, green, blue and alpha color values of the current pixel, regardless of the color channel the variable is used in	0 – 255
c*	specifies the color channel value of the current pixel depending on the code channel (R, G, B or A) the variable is used in	0 – 255
x	specifies the x-coordinate (horizontal position) of the current pixel	0 – X-1
y	specifies the y-coordinate (vertical position) of the current pixel	0 – Y-1
z*	specifies the channel index of the current code channel the variable is called from. 0 = red, 1 = green, 2 = blue, 3 = alpha	0 – 3
i	specifies the Y color value of the current pixel in the YUV-color space	0 – 255
u	specifies the U color value of the current pixel in the YUV-color space	-56 – 56
v	specifies the V color value of the current pixel in the YUV-color space	-78 – 78
d	specifies the angle (or direction) of the current pixel around the center of the image	-512 – 511
m	specifies the distance (or magnitude) of the current pixel from the center of the image	0 – M

**Note: the variables* c *and* z *are not supported by the ForEveryPixel handler.*

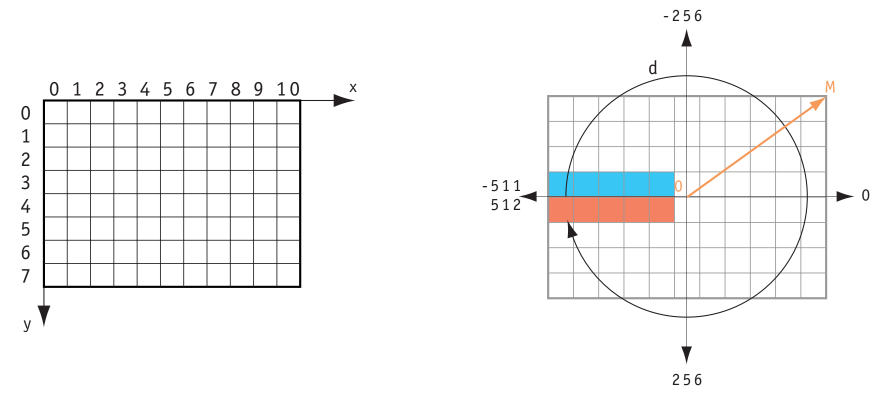


Fig. 2-2-a: Visualization of the x, y, d, m variables and X, Y, M pseudo-constants

Figure 2-2-a visualizes the variables x, y, d and m and the pseudo-constants X, Y and M on an imaginary image. The x coordinates range from 0 to 10, giving a total of 11 pixels in the width (X); the y coordinates range from 0 to 7, giving a total of 8 pixels in the height (Y). The blue pixels from (0, 3) to (4,3) have an angle or direction value of -511, while the red pixels from (0,4) to (4,4) have a direction value of 511. The direction changes clockwise, as you can see The magnitude m (red line) of the center pixel at (5,3) is 0, while the pixels at the edge all have a magnitude of M (half the image's diagonal) or 7 (due to integer inaccuracy, some magnitude values may differ from this number).

These variables can be used in FilterMeister programs and are supported by all handlers:

Variable(s)	Description	Value range
scaleFactor	specifies the factor to be scaled to 100% according to the preview window's zoom factor (always in integer, meaning a slight inaccuracy in the lowest zoom factors has to be taken in consideration)	1 – 16
rows, columns	specifies the total amount of rows and columns the preview window is currently showing	varies
doingProxy	specifies if the filtering process is applied to the preview image or to the image layer in the host program	0 or 1

Among these variables, you can define your own. These can be used as constants, loop counters, flags, etc. First, you should know that FilterMeister not only supports integers but also other types of numerical expressions:

Type	Memory size	Value range
int	4 bytes	-2,147,483,648 – 2,147,483,647
unsigned int	4 bytes	0 – 4,294,967,295
double	8 bytes	1.7×10^{-308} – 1.7×10^{308}

Integers are whole numbers. The computer works the fastest when operating with integers. The difference between *signed* and *unsigned* is that signed numbers can be negative, while unsigned numbers are always positive or zero. **Double** numbers give the programmer more precision by using floating point numbers. As you can see, the memory requirements for the individual types change.

For further information, consult any C programming language documentation about type specifiers.

The following program shows how new variables are declared. Note that for easier understanding (especially in larger programs) we prefixed the variable names with i, f, d and c. That way it is easier to guess what type the variable is. It is possible to declare a variable and give it a value right away – this is called initializing.

```
//Listing 2-2.a

%ffp

ForEveryTile:
{
int iCount, iCheck=5;                     //declare integer variables and initialize iCheck
double dPi=3.1415926;                     //initialize floating point (double) variable

for (iCount=1; iCount<5; iCount++)        //do four loops
        Info("Pi × %d = %f", iCount, iCount*dPi);      //present info window
return false;                             //do not alter the image
}
```

Note that some variables are only declared and others are additionally initialized with a certain value. Loops are described later in this manual. All you have to know is that the progam above presents four Info windows in sequence that show multiples of the greek Pi constant.

Operators

FilterMeister offers you arithmetical, logical, relational, shifting, conditional and the separator operators to combine two operands in some mathematical way (exceptions: logical NOT, bitiwise NOT, conditional and comma operators)! For example, in the term 5+6*x, the numbers and the variable x are the operands, while the + and * are the operators.

Arithmetic operators

+	adds left and right operands
-	subtracts the right operand from the left operand
*	multiplies the left and right operands
/	the left operand is divided by the right operand
%	this is the modulo operator and returns the remainder of the division of the operands

Examples:	10 % 3 returns 1 (ten divided by three equals three, remainder one)
	15 % 5 returns 0, 22 % 19 returns 3

Logical operators

&&	AND operator – returns 1 (true), if left and right operands/terms equal 1, else returns 0 (false)
||	OR operator – returns 1 (true), if left or/and right operands/terms equal 1, else returns 0 (false)
!	NOT operator – exchanges value of right operand: true to false and false to true

Relational operators

==	compares if left and right operands are equal – returns 1 if true, else 0
!=	compares if left and right operands are not equal – returns 1 if true, else 0
>	compares if left operand is greater than the right operand – returns 1 if true, else 0
>=	compares if left operand is greater than or equal to the right operand – returns 1 if true, else 0
<	compares if left operand is less than the right operand – returns 1 if true, else 0
<=	compares if left operand is less than or equal to the right operand – returns 1 if true, else 0

Bitwise and shifting operators

&	bitwise AND operator
|	bitwise OR operator
^	bitwise XOR operator
~	bitwise NOT operator

The XOR operator is an exclusive OR operator which results to 1 (true) if only one of both operands is true. It returns 0 (false) when both operands are true respectively false.

There is a difference between the logical and bitwise operators. While the former are used to compare two terms or expressions (is x less than 10 AND y less than 10, then color the pixels black), the latter combines the left and right operands to a new value. Bitwise means that corresponding bits from both operands are compared logically.

Example:

The two binary values 10100101 (decimal 165) and 01101010 (decimal 106) are bitwise operated:

```
         10100101 (dec 165)          10100101                    10100101
& (AND)  01101010 (dec 106)   | (OR) 01101010             ^ (XOR) 01101010
--------------------------------------------------------------------------------
         00100000 (dec 32)           11101111 (dec 239)          11001111 (dec 207)
```

The NOT value of 10100101 (decimal 165) is simply the exchange of 0's and 1's. This leads to the value 01011010 (decimal 90).

Shifting operators:

<<	shifts bits of the left operand by the right operand's value to the *left*
>>	shifts bits of the left operand by the right operand's value to the *right*

Example:

In the term 00001111 << 3 , the left operand is shifted three places to the left by pushing in a 0 in the rightmost pixel:

binary value	**decimal value**	**shift amount**
00001111	15	no shift
00011110	30	one bit shifted
00111100	60	two bits shifted
01111000	120	three bits shifted

So 00001111 (decimal 10) shifted three times to the left equals 011110000 (decimal 120).

Look what happens to a number when shifted one bit to the left. The decimal value is doubled at every shift! If say the muliplication of that number by 2 does the same job. Indeed it does, but how long does it take to shift all pixels of an image once to the left respectively to multiply all pixels by 2? Try to do that with the RGBA handler, then check the answer in the tutorial section later in this chapter...

Conditional operator

a ? b : c	if expression **a** is true, continue with **b**; otherwise continue with **c**

This operator is a simple but effective IF-THEN-ELSE function. Let's try to understand the conditional operator in some examples:

```
%ffp

R:      g<128 ? r-50 : r+50
G:      x<10 || x>X-11 ? 255-g : g
B:      r>150 ? (g>100 ? b-g : b-r ) : (g>100 ? r-b : g-b)
A:      c
```

In the red channel, the expression preceding the question mark tests if the current pixel's green color value is lower than 128. If it is lower (condition is true), then the current pixel's red color (we are working in the red channel) is lowered by 50. If the condition is false (pixel's green color value is equal or higher than 128), the current pixel's red color value is added to 50.

The green channel's code is an example with logical operators. The condition reads as follows: is the current pixel's horizontal position x lower than 10 OR is it higher than X-11 (image's width minus 11)? *We combined two conditions with the AND operator.* If any of these conditions is true, then invert the green channel's pixel value ELSE leave the green channel unchanged. It is possible to combine many more conditions using the logical operators – we recommend the use of parentheses for easier reading.

The last example visualizes the nesting of conditional operators. As you can see, the overall conditional term reads r > 150 ? () : (). In each parentheses term, additional conditional expressions are used.

Comma operator

This operator is used simply as a separator between functions, expressions, key properties, variable definitions etc. A special case is the separation of functions in the RGBA handler (typical Filter Factory programs). Each channel code of this handler returns a numercial value to the current pixel. When a channel code has different expressions that are separated with a comma, both expressions are evaluated, but only the last one is applied to the current pixel. The following program shows you the separation of two terms in the red channel.

```
%ffp

R:      r+g-b, r
G,B:    c
A:      0
```

If you apply the above filter to your image, nothing will happen, although the term r+g-b is evaluated by FilterMeister. This can easily be checked by timing the filter application with and without that term.

Tutorial Section 2.2

GradientMeister 1

Let's experiment a bit with the different operators here. Load any RGB image (with a width greater or equal to 512) and try out the following program:

```
%ffp

R, G, B:   x
A:         a
```

All the above program does is change the current pixel's value to its current x-coordinate. You will see a smooth gradient from left to right. It does stop in the middle, though – the right side of the image is pure white. If you look at the preview window, it displays also a completely different preview (except you zoomed in to 100%). One way to create a smooth border from left to right is to integrate the image's width:

```
%ffp

R, G, B:   x/X*255
A:         a
```

How does your image look like? Did we make an error? Let's check the formula pixel-by-pixel. The first pixel's x-coordinate is 0. So 0/512×256 = 0. That is true. The last pixel's x-coordinate is 511 (X-1), so 511/512×255 = 254.5 . If you zoom in the image, call the information box in Photoshop and move your cursor of the rightmost pixel. It is set to 0! What happened?

FilterMeister is set to operate with integers, so the division x/X returns the number 0, because there are no fractions in integer operations. And zero multiplied with 255 equals 0.

We have to pay attention in our filter programs that no term or parts of it returns 0. Therefore, we can change the order of the operations as in:

```
// Listing 2.2.c
// Smooth operator

%ffp

R, G, B:   x*255/X
A:         a
```

The above program leads to a smooth gradient from left to right and the preview window displays it correctly. Is it really necessary to correct the preview window? In fact, there are two reasons: The users *want* WYSIWYG (what you see is what you get) and get pretty enerved when they are happy with the results on the preview window, but get a completely different result in the final image. The second reason is that the FM programmer always has to remember that his code has to be usable on any image size. If you are using image-dependent variables or constants such as x, y, m, X, Y and M, try to find a way to make your filter image-independent – your reputation as a professional programmer is at stake!

While the above term x*255/X is fine to use, there is a function that helps a lot, too, especially when the term is not easy as above. The function is called scl(), which is explained in the next chapter.

Let's take a look at the modulo operator. Why is it so useful if it only returns a remainder? It is useful for repeating sequences and these are interesting for patterns or textures. Look at the following table which represents the term x%5:

x	0	1	2	3	4	5	6	7	8	9	10
x%5	0	1	2	3	4	0	1	2	3	4	0

As you can see, the sequence 0, 1, 2, 3, 4 is repeated over and over again. Continue using the 512 × ? RGB-image and run the following filter:

```
%ffp

R, G, B:   x%256
A:         a
```

Two smooth black-to-white gradients, but it works only with 512 pixel wide images. The standardized code for all image widths looks like the following:

```
(2*x*255/X)%256
```

The first number gives the amount of gradients, so you can change it to 3, 4 or whatever you want. But a user will not have the possibility of changing this number – except he uses a user control function...

In chapter 4.2 of the Getting Started Manual, you learned a bit about user controls. We simply have to define a user control and call its value from within the filter code:

```
%ffp

ctl[0]: "Amount of gradients"

R, G, B:   ( ctl(0) * x*255/X ) %256
A:         a
```

Great! Now you can toy around with the scroll bar without having to retype and recompile the code over and over again. Let's implement the conditional operator to create either a horizontal or a vertical gradient. We will be needing a check box for that matter.

```
// Listing 2.2.d
// B/W gradients (Part 1)

%ffp

ctl[0]: "Amount of gradients", val=3
ctl[2]: CHECKBOX, "Horizontal/Vertical Toggle", size=(94, *)

R, G, B:
ctl(2) ?                        // is the checkbox checked?
( ctl(0) * y*255/Y ) %256 :     // make a vertical gradient
( ctl(0) * x*255/X ) %256       // make a horizontal gradient

A: a
```

The first user control is initialized with the value of 3 gradients. The check box had to be resized because the text string was too large for the default size. In the RGB channel code elements the check box is tested if it is checked or not. If it is, the vertical gradient code is applied to the image – if not, the horizontal gradient code is applied.

The above program is not complete, though. In the tutorial of the next chapter, we will be adding some colors to the gradient filter!

But wait, here is an insider tip: the usage of the cartesian x and y variables can often be transferred to polar coordinate variables m and d. Picture this with the help of the gradients. With x and y, we created a horizontal gradient respectively a vertical gradient. With m and d, a radial respectively rotating gradient is possible. In the case of m, a simple variable exchange is possible:

```
%ffp

ctl[0]: "Amount of gradients", val=3

R, G, B:   ( ctl(0) * m*255/M ) %256
A:         a
```

The filter code of the direction (or angle) variable d will be explained in the tutorial of the next chapter.

BorderMeister

The creation of borders is a bit time-consuming in Photoshop, so FM comes in handy to do the job. For example, we want a 10-pixels-thick border. In the horizontal line, the first ten pixels range from 0 to 9 and the last ten pixels from X-11 to X-1 (remember, X is the width and X-1 the rightmost pixel coordinate). The same ranges apply to the vertical line. How can it all be combined in one code? We can use the logical operators:

"IF the current pixel is either at the beginning OR at the end of the horizontal line OR at the beginning OR at the end of the vertical line, then color it black ELSE leave the pixel unchanged."

In FM code, the filter would look like:

```
%ffp

R, G, B:   ( x < 10 || x > X-11 || y < 10 || y > Y-11) ? 0 : c
A:         a
```

Let's add some user controls for border thickness and border color! We simply replace the numbers in the condition by the first user control, the following three user controls represent the color values (RGB) of the border.

```
// Listing 2.2.e.1
// Bordermeister (Part 1)

%ffp

ctl[0]: "Border thickness"

ctl[2]: "Red coloring"
ctl[3]: "Green coloring"
ctl[4]: "Blue coloring

R:   ( x < ctl(0) || x > X-ctl(0)-1 || y < ctl(0) || y > Y-ctl(0)-1) ? ctl(2) : r
G:   ( x < ctl(0) || x > X-ctl(0)-1 || y < ctl(0) || y > Y-ctl(0)-1) ? ctl(3) : g
B:   ( x < ctl(0) || x > X-ctl(0)-1 || y < ctl(0) || y > Y-ctl(0)-1) ? ctl(4) : b
A:   a
```

Are you missing something? Maybe WYSIWYG? That's a tough one. Sometimes, users want to be very exact ("I want a border exactly 10 pixels thick! Not 9 nor 11, 10!") and that is a dilemma. If we introduce WYSIWYG, the accuracy is lost. Darned! Nevertheless, we need another new function that can help us do some WYSIWYG. Of course, it is possible to implement a checkbox which toggles between accuracy and WYSIWYG.

You are still missing something? Oh, you mean the alpha channel... We have neglected it up to now and surely, the programs in this tutorial section would not work on an empty layer (shame on us!). The alpha channel code would have to fill the border with opacity (255).

```
A:   ( x < ctl(0) || x > X-ctl(0)-1 || y < ctl(0) || y > Y-ctl(0)-1) ? 255 : a
```

Check, mate?

This tutorial deals with checkered tiling. If you are afraid of boolean maths, skip this section. It is a bit difficult to understand, but if you play with it, you will get a hang on it.

Think of a 4 × 4 checkerboard image with the size 20 × 20 pixels.

On the top row, the first five pixels are black, the next five are white, the next five black and the last five white. Consecutive ordering, that is a job for the modulo operator! We have a two-color group that changes every ten pixels. If we use x%10, we will get a sequence of 0, 1, 2, 3, 4, 5, 6,7, 8, and 9 over and over again.

This sequence can be divided into two parts: 0–4 and 5–9. We can test if a pixel's modulo lies in one of these two groups and color it black or white.

```
%ffp

R, G, B:   x%10 < 5 ? 0 : 255
A:         a
```

The above program leads to a zebra crossing in your image. A vertical zebra crossing can be accomplished if x is substituted with y. There are various ways to combine both stripe textures to a checkerboard textu-re. A vertical black stripe combined with the horizontal black stripe leads to a black checkerboard cell. A white stripe combined with a black stripe leads to a white checkerboard cell. Finally, a vertical white stripe combined with a horizontal white stripe leads to a black checkerboard cell.

Remember that a relational operator (as in x%10 < 5) returns a 1 or 0, depending on whether the term is true or false. We can combine both terms with the XOR operator. If both boolean values are equal, return false; if both boolean values are different, return true.

		x%10 < 5	
		true	false
y%10 < 5	true	false	true
	false	true	false

The expression for the table reads as follows:

```
(x%10 < 5) ^ (y%10 < 5)
```

Since the above expression returns only 0 or 1, we have to multiply it with 255. That way, we get the two values 0 (black) and 255 (white):

```
( (x%10 < 5) ^ (y%10 < 5) ) * 255
```

All we now need is the implementation of two scrollbars to define the amount of checkerboard cells in a row and column. How is the relationship between the checkerboard cell width in pixels and the user control setting? If the scroll bar is set to 3 (to make a 3 × 3 checkerboard), the cell width is the image width divided by 3 or X/3. Note in the above checkrboard expression that we have to replace the numbers 10 (double cell width) and 5 (cell width) with X/ctl(0)*2 and X/ctl(0) respectively Y/ctl(0)*2 and Y/ctl(0):

```
%ffp

ctl[0]: "Cell amount"

R, G, B:    ( (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ) * 255
A:          255
```

The final effect we will do is to add some cell coloring. In this case, we need six scroll bars for RGB values for the two cell types. We simply remove the multiplication with 255 and the expression returns either 0 or 1. This is great for the conditional operator. Look at the program below:

```
// Listing 2.2.f
// CheckerMeister

ctl[0]: "Cell amount", val=3, range=(2,50)

ctl[2]: "Red value (cell 1)"
ctl[3]: "Green value (cell 1)"
ctl[4]: "Blue value (cell 1)"

ctl[6]: "Red value (cell 2)", val=255
ctl[7]: "Green value (cell 2)", val=255
ctl[8]: "Blue value (cell 2)", val=255

R: (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ? ctl(6) : ctl(2)
G: (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ? ctl(7) : ctl(3)
B: (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ? ctl(8) : ctl(4)

A: 255
```

We initialized the cell amount scroll bar with 3 (default value was 0 or ”no cells visible”) and redefined the range from 2 to 50. Lower or higher values are mostly not needed. The first three scroll bar group define the color of the first cell and the second scroll bar group defines the color of the second cell. We removed the multiplication and replaced it with the conditional operator, which evaluates the true/false state and applies the respective scroll bar value to the color of the cell.

Let’s take the checkerboard pattern filter and modify it at follows: All white tiles leave image as it is and all black tiles invert the image. The original black and white CheckerMeister looked as follows:

```
%ffp
```

```
ctl[0]: "Cell amount"

R, G, B:   ( (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ) * 255
A:         255
```

Remember why we multiplied by 255? Because the term returned only 0 or 1. This is great for the conditional operator! All we have to do is replace the *255 with ? tr : fa. The first term tr leaves the image as it is by using the variable c. The second term fa inverts the image by using the term 255-c. The complete program reads as follows:

```
%ffp

ctl[0]: "Cell amount", val=3

R, G, B:   (x%(X/ctl(0)*2) < (X/ctl(0))) ^ (y%(Y/ctl(0)*2) < (Y/ctl(0))) ? c : 255-c
A:         255
```

This filter tutorial has shown you an insight of the filter creation process. We do not claim that it is **the** way, but it surely shows a step-by-step process of how complex filters can be programmed. Start easy and extend the filter slowly. That way, you will have an overview of what is happening. If you document the process, it is easy to understand the code a couple of months later.

Shift worker struck by a bit

Remember we mentioned how handy bit shifting can be? It can be very helpful for multiplication reasons. In the information bar of Photoshop, activate the timing function (click on the triangle for the fly-out menu) and run the following two programs:

```
%ffp

R, G, B:   c*2
A:         a
```

```
%ffp

R, G, B:   c<<1
A:         a
```

Depending on the image size, the last code is faster!

Can you handle it?

As you can see from the tutorial section, we have used only the RGBA handler which is very handy for simple (but also some complicated) filters. Of course, the programs could have been listed in the

ForEveryPixel and ForEveryTile handler structures. When do we use a certain handler? It depends on the filter's task. If it needs to collect and evaluate some data before applying the filter algorithm, one would have to work with the ForEveryTile handler.

The usage of a certain handler can mean a speed optimization. Often it depends on how many pixels of the image are processed or changed. For example, take the filter BorderMeister. The RGBA handler runs through all the pixels in the image, but it only changes pixels within the image's borders. The rest of the image is left unchanged. Let's reprogram BorderMeister for the ForEveryTile structure:

```
// Listing 2.2.e.2
// BorderMeister (Part 2)

%ffp
ctl[0]: "Border thickness", val=10
ctl[2]: "Red coloring"
ctl[3]: "Green coloring"
ctl[4]: "Blue coloring

ForEveryTile:
{
for (x=0; x<X; x++)                     // Loop 1
for (y=0; y<ctl(0); y++)
for (z=0; z<Z; z++)
{
        pset(x, y, z, ctl(z+2));
        pset(x, Y-y-1, z, ctl(z+2));
}

for (y=ctl(0); y<Y-ctl(0); y++)         // Loop 2
for (x=0; x<ctl(0); x++)
for (z=0; z<Z; z++)
{
        pset(x, y, z, ctl(z+2));
        pset(X-x-1, y, z, ctl(z+2));
}
return true;
}
```

The program consists of two loops. The first loop "paints" the top and bottom borders. The second loop "paints" the left and right borders and is also optimized not to paint the left and right parts of the top and bottom borders. Although the program listing is larger than the RGBA handler listing, the speed improvement is marvelous – we painted a 10 pixel border and a 200 pixel border on a 2048 × 1535 RGB image:

The ForEveryTile handler was five times faster (10 pixel border) respectively three times faster (200 pixel border) than the RGBA handler! This was an insider tip, so keep it always in mind!

2.3 Functions

The following chapters describe most of the available functions for FM filter programs. Please understand that we cannot describe all of the functions. Please consult any C or C++ language reference for further study. All functions described here work in all handlers, except otherwise noted.

2.3.1 Math functions

FilterMeister offers a variety of mathematical functions.

abs(n), fabs(n)

These functions return the absolute value of the number `n`. While `abs()` is used for integers, `fabs()` is used for floating point numbers.

Example:

```
Info("Absolute value of %d is %d\nAbsolute value of %.2f is %.2f", -43, abs(-43), 23.224,
fabs(23.224) );
```

add(a, b, c), sub(a, b, c)

Introduced in Filter Factory, the `add()`-function adds the terms a and b, compares the result with c and returns the lower integer value. The `sub()`-function subtracts the term b from a, compares the result with c and returns the integer higher value. *See our comment in the max()/min() function section later.*

Examples:

```
Info("Comparing %d+%d with %d, the lower value is %d", 100, 50, 128, add(100,50,128) );

Info("Comparing %d-%d with %d, the higher value is %d", 10, 20, -20, sub(10,20,-20) );
```

dif(a,b)

This function returns the absolute integer difference between `a` and `b`. Example:

```
Info("The difference between %d and %d is %d", -5, 6, dif(-5,6));
```

max(a,b), min(a,b)

These functions evaluate the terms a and b and return the higher respectively the lower integer value.
Tip: max(a-b, c) and min(a+b,c) work exactly like the add() and sub() functions and are computed faster.

Examples:

```
Info("max(%d,%d) returns %d", -10, 55, max(-10,55) );

Info("min(%d,%d) returns %d", -10, 55, min(-10,55) );
```

scl(i, ilo, ihi, olo, ohi)

One of the most practical functions is the *scaling* function. It takes the term `i` and places its range onto the low and high input values (often, the normal range is set here). Then it rescales the input range onto a different (output) range, which has new low and high output (integer) values.

Example:

```
R,G,B:  scl(x, 0, X-1, 0, 255)
```

In the above function, the input range of `x` is `(0,X-1)` and is now rescaled to `(0,255)`. This function leads to the same image when using the term `x*255/X`.

Also note that the step size is set to the step sizes set in (ilo, ihi) or (olo, ohi), whichever has a minimum amount of steps. The following example visualizes this clearly:

```
R, G, B:   scl( scl(x,0,X-1,0,3), 0, 3, 0, 255)
```

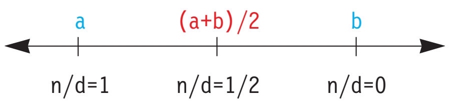
The above example first minimizes the range from (0, X-1) to (0, 3). Next, although it maximizes the range from (0,3) to (0,255), the step amount is still 3.

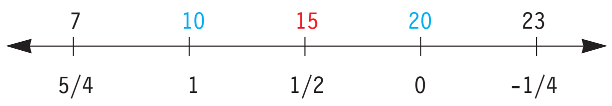
mix(a, b, n, d)

The mix()-function allows a special access within a range `(a, b)` and out of its bounds through the fraction `n/d`. These are the rules:

If the fraction is close to 0, values near b are returned.
If the fraction is close to 1, values near a are returned.
If the fraction is exactly 1/2, the mid-point value between a and b is returned.

Visual representation (with example):





Random functions: rnd(a, b), rst(a), rand(), srand(a)

The `rnd()`-function returns a random number between a and b, inclusive. The `rand()`-function returns a random number between 0 and 32767. These functions are pseudo-random functions: if you call one of these functions over and over again, you will get the same number, which is not in the random nature. Apply the following filter to any RGB image:

```
%ffp

R,G,B: rnd(0,255)               //give current pixel a random color
```

If you apply it again, you will note that the image does not change. These functions work with a certain "seed" that is fed upon function call and this seed does not change constantly. For true random numbers, this seed can be changed by calling the functions `rst(a)` and `srand(a)`. The most effective term for a is the call of the `clock()` function, which constantly changes the seed. If you want your filter to always return true random numbers, insert the `rst()` or `srand()` functions. Example:

```
%ffp

ForEveryTile:
{
rst(clock());                   // change random seed
return false;                   // let the RGBA handler do the filtering
}

R,G,B: rnd(0,255)               // give current pixel a random color
```

Note that the `rst(clock())` function is called once every tile and not from within every pixel. Although it is possible to insert the function in the RGB handler, it would start a new seed in every pixel, slowing down the filter considerably.

sqr(a), sqrt(a), pow(a, b)

The first two functions both return the square roots of the term `a`. Note that `sqr()` returns integers and floating point numbers, while `sqrt()` returns only floating point numbers. `pow(a,b)` returns the floating point value of the term "a raised to the b power" (as in 2 to the 3 power $= 2 \times 2 \times 2 = 8$).

Examples:

```
Info("sqr(9) = %d\nsqr (19.2) = %.2f\nsqrt(81.0) = %.2f", sqr(9), sqr(19.2), sqrt(81.0) );

Info("Three raised to the 5th power: %.2f", pow(3.0, 5.0) );
```

Logarithmic functions: exp(a), log(b), log10(c), ldexp(f, i)

The `exp()`-function represents 2.7182818 (the base of natural numbers) raised to the power of `a`. The `log()`-function returns the natural logarithm of the term `b`. `log10()` returns the base-10 logarithm of the term `c`. The function `ldexp(f,i)` multiplies the floating point number `f` by two raised to the `n`-th power ($m \times 2^n$). Note that `i` is an integer. All functions return floating point values.

Example:

```
Info("exp(4.596) = %.2f \nlog(99.0) = %.2f \nlog10(1000.0) = %.2f", exp(4.596), log(99.0),
log10(1000.0) );

Info("5.3 × 2^4 = %.2f", ldexp(5.3, 4) );
```

ceil(u), floor(v)

These two functions take the term u respectively v and return the whole numbers directly above respectively below the actual term. For example, in the number 5.3, the ceiling is equal to 6 and the floor is equal to 5. Both functions return floating point values.

Example:

```
Info("ceil(5.2) = %.2f \n floor(5.2) = %.2f", ceil(5.2), floor(5.2));
```

fmod(a, b)

This function resembles the MODULO operator `%` but is used for floating point values. This function returns the remainder of the division `a/b`.

Example:

```
Info("The remainder of 8.0 / 4.2 is %.2f", fmod(8.0,4.2) );
```

Trigonometrical functions: sin(x), cos(y), tan(z), et al.

FilterMeister supports integer and floating point trigonometry. Note in the figure 2-3-1-a below that the return values are different. In **integer** arithmetics (and backward Filter Factory compatibility), the trigonometry functions return values within the range (-512, 511) while in **floating point** arithmetics, the trigonometry-functions return values within the range (-1, 1).

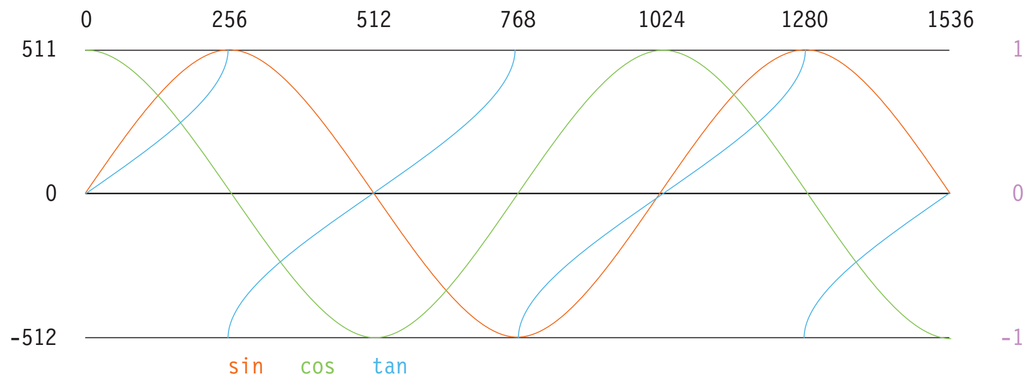


Fig. 2-3-1-a: Trigonometry functions

In addition to the above functions, FilterMeister offers the following trigonometry functions (which return only floating point values):

`fsin(a), fcos(b), ftan(c)`	*resemble the above functions with extra precision (slower)*
`asin(a), acos(b), atan(c)`	*return the arcsine, arccosine and arctangens of the corresponding term*
`sinh(a), cosh(b), tanh(c)`	*return the hyperbolic sine, cosine and tangens of the corresponding term*

Please consult a math book for correct return values of these functions.

Polar coordinate functions r2x(d,m), r2y(d,m), c2d(x,y), c2m(x,y)

These last math functions help to convert cartesian to polar coordinates and vice-versa. In cartesian coordinates, you have the two axes x and y which are in a 90° angle to each other. In polar (or radial) coordinates, you have the two axes d and m which represent an angle (direction) and magnitude (or distance from the (0,0) point). Figure 2-3-1-b shows the two different coordinate types.

See chapter 2.2 where we introduced the variables d and m.

The functions `r2x(d,m)` and `r2y(d,m)` can be read as to "convert the **r**adial coordinate `(d,m)` **to** its corresponding **x**- (horizontal) respectively **y**- (vertical) cartesian coordinate". The functions `c2d(x,y)` and `c2m(x,y)` can be read as to "compute the **c**artesian coordinate `(x,y)` **to** its corresponding **d** (angle) respectively **m** (magnitude) coordinate". All four functions return integers values.

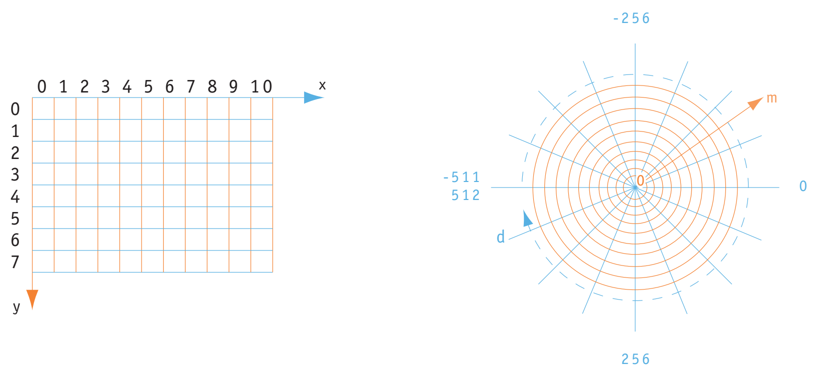


Fig. 2-3-1-b: Visualization of the cartesian and polar coordinate types

2.3.2 User control functions

This very short chapter will introduce you to the implementation of user control settings within the filter program. Most of the filters include some kind of user interaction, because the users demand that they want to fiddle around with different settings. FilterMeister offers the filter programmer a variety of user controls (described in chapter 3.3) which are defined and initialized in the head of the FM program. The most common user controls are the scroll bars and checkboxes. Other user controls are combo boxes (pull-down menus) radio buttons and push buttons and many more.

As soon as a user control is defined, it can be implemented within any filter program handler. The current user control setting can be called with two functions: `ctl(i)` and `val(i,m,n)`. `ctl(i)` calls the setting of the user control with the index `i` and `val(i,m,n)` calls the setting of the user control with the index `i`, but projects (or scales) the range of the user control to a new range `(m,n)`.

Note that calling the user control setting with `val()` is not very efficient, since it rescales the range, but the range's new borders are not shown in the user control itself. Example:

```
%ffp

ctl[0]: "Move me"                // default range is (0, 255)

R, G, B: val(0,-128,128)
A: a
```

If you compile the above program you will see that the user control text box will still display values between 0 and 255, but the preview window shows the possible range from 0 (a setting of -128 to -1 is clamped to 0) to 128 (mid-gray). The val() function is a relict from ancient Filter Factory times and we strongly recommend the usage of the property range in the user control definition.

The above program can be rewritten as follows:

```
%ffp

ctl[0]: "Move me", range=(-128,128)

R,G,B:  ctl(0)
A:      a
```

After compilation one can see that the user control edit box correctly shows the ranges and the preview window behaves as before. Study chapter 3.3 for more user control options.

2.3.3 Image and Tile Buffer functions

Image functions are functions which have direct/indirect access to pixels, especially when you recolor or distort images or image parts. Some functions are true digital image synthesis functions which help analyze and improve images. All image functions return integer values.

src(x,y,z), rad(d,m,z)

Both functions access and return the pixel value in the cartesian coordinates (x,y) in channel z respectively in the polar (or radial) coordinates (d,m) in channel z.

Look at the following simple mirror and rotation codes:

`src(X-x-1, y, z)`	mirrors image horizontally
`src(x, Y-y-1, z)`	mirrors image vertically
`src(y, x, z)`	rotates image 90° counterclockwise*
`src(y, X-x-1, z)`	rotates image 90° clockwise*
`src(X-x-1, Y-y-1, z)`	rotates image 180° (or simultaneous horizontal and vertical mirror)

** the above codes work only well in images which have same width and height, e.g. 250 × 250 pixels*

cnv(m11, m12, m13, m21, m22, m23, m31, m32, m33, d),
cnvX(k, off, d, pGetf, x, y, z), cnvY(k, off, d, pGetf, x, y, z)

Do you know the custom filter in Photoshop? It is also known as a convolution filter and the above functions are special convolution functions.

Figure 2-3-3-a visualizes the procedure of the convolution function. The cnv() function is based on a 3 × 3 matrix where the weighting cells are named m11, ... m33 and d is the denominator. Picture it as if the matrix's center cell (kernel) is placed onto the current pixel. Each cell represents a weight factor which is multiplied with the pixel on that position relative to the kernel. All nine pixels are weighed and added altogether. The result is divided by d, which in most cases is the sum of the weight factors.

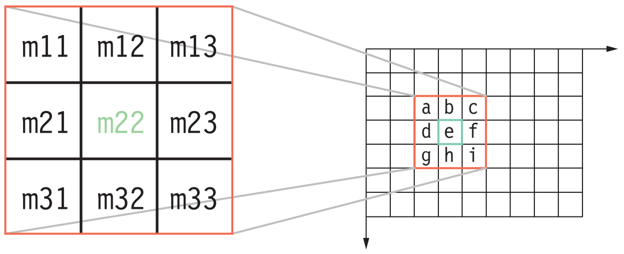


Fig. 2-3-3-a: Convolution filter

The formula for the convolution kernel goes:

$$\frac{m11 \times a + m12 \times b + m13 \times c + m21 \times d + m22 \times e + m23 \times f + m31 \times g + m32 \times h + m33 \times i}{d}$$

Examples (use them in the RGBA or ForEveryPixel handler):

```
cnv(1, 1, 1, 1, 1, 1, 1, 1, 1, 8)      // simple blur filter (also called low pass filter)
cnv(1, 2, 1, 2, 4, 2, 1, 2, 1, 16)     // simple gaussian blur
cnv(-1, -1, -1, -1, 9, -1, -1, -1, -1, 1)      // sharpen filter
cnv(-1, -1, -1, -1, 8, -1, -1, -1, -1, 1)      // LaPlace operator
```

`cnvX()` and `cnvY()` are one-dimensional convolution filters, meaning that either a horizontal or vertical convolution is processed. In the syntax `cnvX(k,off,d,pGetf,x,y,z)`, the following terms are used:

`k`	radius of the kernel (k>=0)
`off`	starting index within the anonymous array of put/get cells where the kernel coefficients are stored (the amount of coefficients is $n = k \times 2 + 1$)
`d`	denominator (d != 0), by which the convolution sum will be divided
`pGetf`	calls the function to fetch values from an image buffer at the coordinates `(x,y,z)`. `pGetf` must be either `src`, `tget`, `t2get` or `pget`
`x`	x-coordinate at which the center of the kernel will be applied
`y`	y-coordinate at which the center of the kernel will be applied
`z`	channel number to which the kernel will be applied

Both functions return an integer value, obtained by summing the products of the n kernel coefficients with n image pixels in the X respectively Y direction, and dividing the sum by the denominator d.

pset(x, y, z, a), pget(x, y, z), tset(x, y, z, a), tget(x, y, z), t2set(x, y, z, a), t2get(x, y, z)

Before we explain the functions above, you have to know that FilterMeister offers two tile buffers (the twins are called ”tbuf” and ”t2buf”) which can hold additional data. That way, it is possible to analyse the image and save the data (for example from a cnv()-function) to a buffer and then combine the original image data with the buffer data. Buffers are also interesting for iterative algorithms (e.g., for plasma clouds, fractals, etc.). All functions return an integer value.

pset(x, y, z, a)	sets at coordinate (x, y, z) the value a in the destination (or output) tile
pget(x,y,z)	fetches the pixel at (x, y, z) from the destination tile
src(x, y, z)	fetches the pixel at (x, y, z) from the source (or original) image

The tget()- and tset()-functions are analogous to pget() and pset(), with the difference that these fetch respectively set a value from/in a pixel in the buffer tile 1 or 2.

`tset(x,y,z,a)`	sets at coordinate (x,y, z) the value a in buffer 1
`tget(x,y,z)`	fetches the pixel at (x,y,z) from buffer 1
`t2set(x,y,z,a)`	sets at coordinate (x,y, z) the value a in buffer 2
`t2get(x,y,z)`	fetches the pixel at (x,y,z) from buffer 2

Note that in some cases, the tile buffers are as large as the image itself. This is when one is using functions which can have access to the whole image: src(), rad(), pset() and pget().

Examples:

```
// Example 1
// Apply a 10-pixel mosaic effect

%ffp

ForEveryTile:
{
        for (y=0; y<ymax; y++)          // rows
        for (x=0; x<xmax; x++)          // columns
        for (z=0; z<zmax; z++)          // channels
                pset(x, y, z, src(x-x%10, y-y%10, z))

return true;
}


// Example 2
// Mirror the right half of the image

%ffp

R, G, B:   pget(X-x-1, y, z)
A:         a
```

Note the curiosity in example 2. The pget()-function fetches the adjacent horizontal pixel (of the output image) on the right side and sets it to the current pixel. As soon as the function is halfway through the row, it fetches now pixels from the left side and sets them in the right side. But since the left side of the image has already been altered, the pget()-function mirrors now the left side (that has image info of the right side), thus overwriting the pixels with their original values.

FM-Knowledge question:

Is there a way to make the above filter faster, while leaving the function as it is?

Answer:

Yes, since only the left side of the image is altered, insert a condition to test if the current pixel position is left-handed:

```
%ffp

 R, G, B:   x<X/2 ? pget(X-x-1, y, z) : c
 A:         a
```

The following two-pass program first mirrors the current image horizontally, combines it with the original with the lighten mode and saves the result in buffer 1. Then it mirrors the buffer vertically, combines it with the buffer data with the darken mode and saves the result in the output image.

```
%ffp

ForEveryTile:
{

for (x=0; x<xmax; x++)
for (y=0; y<ymax; y++)
for (z=0; z<zmax; z++)
        tset(x,y,z, min( src(X-x-1,y,z), src(x,y,z) ) );

for (x=0; x<xmax; x++)
for (y=0; y<ymax; y++)
for (z=0; z<zmax; z++)
        pset(x,y,z, min( tget(x,Y-y-1,z), tget(x,y,z) ) );

return true;    // apply the filter to the image
}
```

2.3.4 Memory cell functions

put(t, i), get(i)

Talking about saving and reading data to and from buffers, FilterMeister offers 256 memory cells which can contain a 4-byte integer value each. To save a value in a cell, call the `put(t,i)`-function, where `t` is the term and `i` the index of the cell. To read the value from a cell, use the `get(i)`-function, where `i` is the index of the cell.

The index `i` must be a value between 0 and 255, inclusive. If the index number is out of that range, FilterMeister performs a modulo (%256) operation. For example, `get(260)` would give the same result as `get(4)`. One good reason to use memory cells is for example a term that is used over and over again in the filter algorithm and takes too much space. It is simpler to save the result of that term in a memory cell and exchange the term in the algorithm with the `get()`-function pointing to that memory cell.

Much easier is the definition of a variable and asigning it to the term. The `get()` and `put()` functions are relicts from Filter Factory, but were used in many cases, since variable definitions were not possible.

2.3.5 Time-based functions

There are two groups of time-based functions. The first do simple tasks such as calling the current time or putting a pause. The second group includes functions which are essential for filter programs such as aborting the filter if user pressed the escape key, update the progress bar among other things.

clock(), time(v)

The clock()-function returns a 4-byte integer which tells the amount of milliseconds since the computer has been on (the cycle lasts about 50 days). The `time(v)`-function returns the amount of seconds since January 1st, 1970 (don't ask... launch up your internet browser and look for the reason) – the term v can be any term or variable, it really does not matter what you put in it. We recommend to use `NULL`.

Examples:

```
Info("Your PC has been on for %d minutes", clock()/6000 );

Info("Since 1970, %d seconds have been counted\nThat would be about %.2f years!",
time(NULL), (double) time(NULL)/3600/24/365);
```

You can offer a time-limited shareware or demo-version of your filter with this nifty little function.

sleep(z)

The above function makes recess for z milliseconds before continuing. Tip: don't use this function within the RGBA-handler! Or else your filter will pause at each pixel...

updateProgress(p, max), updatePreview(v)

The last thing the user wants is that his or her PC is hung up. Sometimes, filters (not the FM standalone filters, of course!) tend to crash the image editing program but nobody notices it: there is no progress bar signaling the stage of the filtering and the animated cursor does not stop moving. This is the time when the user needs to stay cool and wait angrily for the filter to finish.

The function `updateProgress(p,max)` updates the progress bar (mostly in the information bar at the bottom of the screen). The variable `p` describes the current progress in the range (0,`max`). If the user presses the ESC key, a non-zero value is returned by `updateProgress()` – this is important for the functions described in the next section.

The function `updatePreview(v)` updates the preview window with the index `v` in the filter dialog.

Example:

```
%ffp

ForEveryTile:
{
for (y=0; y<Y; y++)             // row by row
{
        for (x=0; x<X; x++)     // column by column
        for (z=0; z<Z; z++)     // channel by channel
                pset(x,y,z, src(x%100,y%100,z)+rnd(-50,50) );   // apply an effect
        updateProgress(y,Y);    // update the progress bar
        updatePreview(a);       // update the preview (only in active filter dialog)
}

return true;
}
```

The above progam applies an effect to an image. On larger images, this may take a while, so the progress bar is updated every row. While the filter is being rendered, you can see now the progress bar smoothly showing you how far the filter is done. If you insert the following line before the `updateProgress()`-line, the progress bar is updated jerkily ten times:

```
                pset(x,y,z, src(x%100,y%100,z)+rnd(-50,50) );   // apply an effect
        if (y%(Y/10)==0)
                updateProgress(y,Y);    // update the progress bar ten times
```

abort(), testAbort()

Since we respect the wishes and problems of our users, we still need some functions for situations like when the user makes a settings error (or sees that his boss is coming right through the door, not expecting some crazy filter effects on the image of his wife...). The ESC key can be used often (if the filter takes too long or the above situation with the boss really comes true), so the filter has to check if this key has been (furiously) depressed with the above functions.

The function `abort()` stops the filter processing abruptly. The function `testAbort()` checks if the user requested an abortion with the ESC or the Pause key. In the first case (ESC), the user wants to stop the filtering process immediately – in the second case (Pause), FM presents the user a message window asking if filter processing should continue or be aborted. If the user wants to stop the filtering process, `testAbort()` returns a non-zero value.

Note: The functions `updateProgress()` *or* `testAbort()` *are not needed in the RGBA or ForEveryPixel handlers; these handlers automatically react to an abort request.*

The above program now includes a user abort request function:

```
%ffp

ForEveryTile:
{
for (y=0; y<Y; y++)             // row by row
{
        for (x=0; x<X; x++)     // column by column
        for (z=0; z<Z; z++)     // channel by channel
                pset(x,y,z, src(x%100,y%100,z)+rnd(-50,50) );   // apply an effect
        updateProgress(y,Y);    // update the progress bar
        if ( testAbort() )      // if the user pressed the ESC key...
                abort();        // ... stop the filtering process
}
return true;
}
```

We mentioned that the `updateProgress()`-function also tests for a ESC-key stroke, so we can neglect the `testAbort()`-function and change the above program to:

```
                pset(x,y,z, src(x%100,y%100,z)+rnd(-50,50) );   // apply an effect
        if ( updateProgress(y,Y) );     // update the progress bar and if user pressed ESC...
                abort();                // ... stop the filtering process
```

2.3.6 Message windows

With FilterMeister, you can call special message windows like error windows, warning windows, information windows and other types. These message windows are called with the following functions:

```
Info()          Warn()          YesNo()         YesNoCancel()
Error()         ErrorOk()       msgBox()
```

All message windows return values you can use further in your filter program. For example, you might ask the user in form of a message window if he/she wants to see the filter applied to the preview window. Depending on which button the user clicks, the filter is either applied to the preview window or not. The following ID values can be returned:

```
IDOK            IDCANCEL        IDABORT         IDRETRY
IDIGNORE        IDYES           IDNO
```

The text strings in the dialog text can include subtrings, escape sequences and format descriptors (see Appendix B for a complete listing of these elements). The n0, n1, ... are variables or mathematical terms which are replaced by the format descriptors in the text string and presented in the message window.

Example:

```
%ffp

ForEveryTile:
{
Info("Decimal integer: \t\t%d\nFloating point number: \t%.2f\nHexadecimal number: \t%X", 500,
3.14159*2, 12648430);
return false;
}
```

As you can see, the text string includes two escape sequences such as the tabulator \t and new line \n. Three format descriptors present an integer, a floating point number (with two figures after the decimal point) and a hexadecimal number (a number you'll need every morning).

Info(text string, n0, n1, ...)

This function displays an information box containing a text string and an OK button. Clicking the OK button returns the value IDOK. The n0, n1, ... variables are optional (see example below).

Examples:

```
Info ("Press OK to continue");

Info ("Your image measures: %d x %d", X, Y);

Info ("This is a\nmulti-line\nInfo-box.");

Info ("Your image mode is:\n!M");
```

If you need an information window with more or other pushbuttons and a different title bar text, use the msgBox() function described later in this chapter.

Warn(text string, n0, n1, ...)

This function displays a warning box containing a text string, an OK button and a Cancel button. Clicking the OK button returns the value IDOK, clicking the Cancel button returns the value IDCANCEL. The n0, n1, ... variables are optional.

Examples:

```
Warn ("This might take a while...");

Warn ("Your image is not a square");

if ( Warn("This might take a while...") == IDOK )
        //apply the following filter code...
```

If you need a warning window with other pushbuttons and a different title bar text, use the msgBox() function described later in this chapter.

YesNo(text string, n0, n1, ...)

This function displays a message box containing a text string, a YES and a NO button. Clicking the YES button returns the value IDYES, clicking on NO returns the value IDNO. The n0, n1, ... variables are optional. Examples:

```
YesNo ("Will you marry me?");

if (YesNo ("Will you marry me?") == IDYES)
        Info("Congratulations!");
else
        Info("Maybe later...");
```

If you need a YES/NO window with more or other pushbuttons and a different title bar text, use the msgBox() function described later in this chapter.

YesNoCancel(text string, n0, n1, ...)

This function displays a message box containing a text string, a YES, a NO and a CANCEL button. Clicking the YES button returns the value IDYES, clicking on NO returns the value IDNO, while clicking on CANCEL returns the value IDCANCEL. The n0, n1, ... variables are optional.

Examples:

```
YesNoCancel ("Do you want to buy %d stocks?", M+X);

if (YesNoCancel("Will you marry me?") == IDYES)
        Info("Congratulations!");
else
        Info("Go to...");
```

If you need a YES/NO/CANCEL window with more or other pushbuttons and a different title bar text, use the msgBox() function described later in this chapter.

Error(text string , n0, n1, ...)

This function displays an error box containing a text string, a CANCEL, a RETRY and an IGNORE button. Clicking the CANCEL button returns the value IDCANCEL, clicking on RETRY returns the value IDRETRY and clicking on IGNORE the value IDIGNORE. The n0, n1, ... variables are optional.

Examples:

```
Error ("The setting %d of the scrollbar %d\nis not allowed", ctl(i), i );

if (Error ("I'm not in the mood") == IDIGNORE)
        Warn("If you ignore me, I'll\ndeinstall !H");
```

If you need an Error window with more or other pushbuttons and a different title bar text, use the msgBox() function described later in this chapter.

ErrorOk(text string , n0, n1, ...)

This function displays an error box containing a text string, and an OK button. Clicking the OK button returns the value IDOK. The n0, n1, ... variables are optional.

Examples:

```
ErrorOk ("Image is too small!!");
```

If you need an ErrorOk window with more or other pushbuttons and a different title bar text, use the msgBox() function described at the bottom.

msgBox(level, title bar text, dialog text)

This function calls a message box with a user-defined title bar text, a user-defines dialog text and at least one pushbutton, which depends on the dialog level set. The following levels can be set:

MB_ICONERROR	places a error icon in the message box
MB_ICONQUESTION	places a YESNO question icon in the message box
MB_ICONWARNING	places a warning sign icon in the message box
MB_ICONINFORMATION	places an Info icon in the message box
MB_OK	places one pushbutton: OK
MB_OKCANCEL	places two pushbuttons: OK and Cancel
MB_ABORTRETRYIGNORE	places three pushbuttons: Abort, Retry and Ignore
MB_YESNOCANCEL	places three pushbuttons: Yes, No and Cancel
MB_YESNO	places two pushbuttons: Yes and No
MB_RETRYCANCEL	places two pushbuttons: Retry and Cancel
MB_APPLMODAL	default: OK button, title bar and dialog text
MB_SYSTEMMODAL	places a windows logo in the title bar

These levels can be combined by adding the symbol | between them. Note that the *title bar text* does not support escape sequences nor substrings.

Examples:

```
msgBox ( MB_YESNO, "Title bar text", "Dialog text");

msgBox ( MB_ICONQUESTION | MB_YESNOCANCEL, "Poppy's Filters Question", "Apply the filter?");

msgBox ( MB_ICONINFORMATION | MB_SYSTEMMODAL | MB_OK, "Filtermania", "This will last for
hours");
```

Tutorial section 2.3-A

So far, we have discussed the most important functions you will be using in FilterMeister programs. If you reached this part (and have not taken a look at the goodies in chapter 3), then you are not really that far away from creating stunning and professional filters. The tutorials in this section will teach you the use of the FF+-functions in 2.3.1 through 2.3.6.

GradientMeister 2

We promised in our first example in tutorial 2.2 that we would do some nifty rotating gradients and add some color to them. Recall the code for the radial gradient:

```
%ffp

ctl[0]: "Amount of gradients", val=3

R, G, B:   ( ctl(0) * m*255/M ) %256
A:         a
```

It is not yet done if m and M are substituted by d and D, respectively. The reason is that the range of the angle d is (-512, 511). We learned a cool function named scl() where a term is scaled from an old range onto a new range, in our case:

```
%ffp

ctl[0]: "Amount of gradients", val=3

R, G, B:   scl(d, -512, 511, 0, 255)    // Do one rotating gradient
A:         a
```

As you can see, the above program does one full gradient clockwise. D is not needed because the range of d is in all images (-512, 511). The output range's max value is multiplied by the user control setting and the whole scl()-function is moduled with 256:

```
%ffp

ctl[0]: "Amount of gradients", val=3

R, G, B:   scl(d, -512, 511, 0, 255*ctl(0) ) %256      // Multiple rotating gradients
A:         a
```

Before we get to the color, think about the following task: let the user offset the gradient angle. By pulling on a second user control, the gradient rotates. The offset value ranges from 0 (no rotation) to 1024/amount of gradients (one gradient section).

```
%ffp

ctl[0]: "Amount of gradients", val=3
ctl[1]: "Offset", range=(1,1024/ctl(0)), Track

R,G,B:  scl(d+ctl(1), -512, 511, 0, 255*ctl(0) ) %256
A:      a
```

If you compile the above program and pull the handle of the Offset scroll bar, you will notice that its maximum value is set to a whole gradient section. It size depends on the gradient amount set in the top scroll bar.

Adding colors is the easiest thing to do, if you have the scale function! Note that all the gradient programs return pixel values from 0 to 255. All we have to do is include six user controls (three for gradient color 1 and three for gradient color 2) as new output range values:

```
// Listing 2.3.a
// GradientMeister (Rotator)

%ffp

ctl[0]: "Amount of gradients", val=3
ctl[1]: "Offset", range=(1,1024/ctl(0)), Track

ctl[2]: "Red starting value"
ctl[3]: "Green starting value"
ctl[4]: "Blue starting value
ctl[5]: "Red ending value", val=255
ctl[6]: "Green ending value", val=255
ctl[7]: "Blue ending value", val=255

R:      scl( scl(d+ctl(1), -512, 511, 0, 255*ctl(0) ) %256, 0, 255, ctl(2), ctl(5) )
G:      scl( scl(d+ctl(1), -512, 511, 0, 255*ctl(0) ) %256, 0, 255, ctl(3), ctl(6) )
B:      scl( scl(d+ctl(1), -512, 511, 0, 255*ctl(0) ) %256, 0, 255, ctl(4), ctl(7) )
A:      a
```

OffsetMeister

When wanting to create seamless tiling for web background images, you normally have to know the image's metrics, call Photoshop's filter "Offset", click on "wrap around" and finally set the offset for x (half the image's width) and y (half the image's height). Surely, there are other programs that can do the job a lot better, but this is a tutorial ("FM for dummies")!

First, let us offset any RGB image with the following code:

```
R, G, B:   src(x-50, y+20, z)
```

Compile the program – what do you see? The image is offsetted 50 pixels to the right and 20 pixels upwards (forget WYSIWYG fornow). At the left and at the bottom, a "repeat edges" effect can be seen. But we want to wrap the image around. This means that the first 50 pixels have to be replaced with the last 50 pixels. A wrap around code would need to check if the src() function is accessing past the image's borders.

The code for the horizontal offset would look like the following:

```
R, G, B:   src(x < 50 ? X-50+x : x-50 , y, z)
A:         a
```

Imagine that the image is 100 pixels wide (X=100). If the current pixel's position is less than half the image's width, then get those pixels on the right side. If the current position is equal to or greater than the image's width, then get those pixels on the left side. But not all of our images have a width of 100, so we have to standardize the code to any size and implement the image's metrics:

```
// Listing 2.3.b
// OffsetMeister 1

%ffp

R, G, B:   src(x<(X/2) ? X-(X/2)+x : x-(X/2), y<(Y/2) ? Y-(Y/2)+y : y-(Y/2), z)
A:         a
```

Calculation (Blending) Modes

Photoshop has some built-in calculation modes which allow two images respectively a tool and the image's contents to be somehow blended into one another. The main interest for implementing calculation mode algorithms in an FM program is the creation of a texture you want to melt with the original image. For example, if you create a drop shadow filter, it is important to combine the shadow part with the background, so that the effect looks more realistic.

Note that the following algorithms contain two variables, a and b, which represent two pixels in two different source images a and b, being b on top of a.

Multiply	**Screen**	**Overlay**
`a*b/255`	`255 - (255-a)*(255-b)/255`	`a<128 ? (2*a*b/255) : 255-2*(255-a)*(255-b)/255`

Soft Light

```
a < 128 ? 2 * scl(b,0,255,64,192)*a/255 : 255-(2*(255-scl(b,0,255,64,192))*(255-a)/255)
```

Hard Light

```
b < 128 ? (2*a*b/255) : 255-2*(255-a)*(255-b)/255
```

Darken	Lighten	Difference
`min (x,y)`	`max (a,b)`	`dif (a,b)` or `abs (a-b)`

Example:

Let’s take the basic gradient code shown in tutorial 2.2 with WYSIWYG. The scroll bar defines the amount of gradients:

```
(ctl(0)* x*255/X) % 256
```

To combine the gradients with the image, we would use the multiply mode, for example (the two colors represent the two terms a and b described above):

```
%ffp

ctl[0]: "Amount of gradients"

R, G, B:   ((ctl(0)* x*255/X) % 256 ) * c /255
A:         a
```

Experiment with these terms with the other calculation modes.

This place?

The following filter will do some displacing in the current image. We will start be placing a scaled horizontal sine gradient sin(x) on the image:

```
R, G, B:   scl( sin(x),-512, 511, 0, 255 )
A:         a
```

The next step is to make the gradient WYSIWYGy by correcting the x variable:

```
R, G, B:   scl( sin(x*255/X),-512, 511, 0, 255 )
A:         a
```

What we then did was to combine a vertical and a horizontal gradient with the difference mode:

```
R, G, B:   dif( scl(sin(x*255/X),-512,511,0,255) , scl(sin(y*255/Y),-512,511,0,255) )
A:         a
```

Wow, looks great! This image will be the basis for the displacement. High values mean a positive displacement, while lower values mean a negative displacement. Remember that the OffsetMeister-filter in the beginning of this chapter also displaced the image half the width and half the height? The src()-function comes in handy for this task. All we have to do now is scale the above term to a subtler range and integrate it in the src()-function:

```
src( x + scl( dif(), 0, 255, -10, 10), y, z)
```

Although we are only displacing in horizontal direction, take a look a what the following filter does:

```
src(x + scl( dif(scl(sin(30*x*255/X),-512,511,0,255),scl(sin(30*y*255/Y),-512,511,0,255))
,0, 255, -10, 10), y, z)
```

We are not finished, though. Let’s add two scroll bars: one for the size (wavelength of the sine waves) and one for the displacement.

```
// Program 2.3.c
// Displacement 1

%ffp

ctl[0]: "Size"
ctl[1]: "Displacement"

R, G, B:
src(x + scl(
dif(scl(sin(ctl(0)*x*255/X),-512,511,0,255), scl(sin(ctl(0)*y*255/Y),-512,511,0,255))
,0, 255, -ctl(1), ctl(1)),y,z)
A:         a
```

MosaicMeister

This small tutorial will show you how to create different mosaic types. Take a look at the following ”classic mosaic” program:

```
%ffp

R, G, B:   src (x - x%5 , y - y%5 , z)
A:         255
```

Apply the above filter to a layer with transparency (a cut out object, for example). All previously transparent areas are now filled with the background color! This is a neat side effect Adobe Filter Factory had.

Let's examine what the term x-x%5 means in a number table:

x	0	1	2	3	4	5	6	7	8	9	10
x%5	0	1	2	3	4	0	1	2	3	4	0
x-x%5	0	0	0	0	0	5	5	5	5	5	10

The term tells the src()-function to fill the first five pixels (0 – 4) with the value used in the x-position 0. The next five pixels (5 – 9) are filled with the value of the pixel in the x-position 5, and so on.

WYSIWYG fans need to adjust that number in the term: x-x%(X/5). That way, the image's width is composed of five mosaic "stones". Needless to say that we need some user controls (for x and y):

```
%ffp

ctl[0]:    "Amount of tiles", range=(2,40)

R, G, B:   src(x-x%(X/ctl(0)), y-y%(Y/ctl(0)),z)
A:         a
```

The following technique is useful for Adobe Photoshop 5.0.x-users. The effect we want to reach is a realistic kitchen tile mosaic in 3D. This is accomplished by creating a transparent border around each tile and applying a layer effect to the layer. All we need to do is change the alpha channel code to:

```
// Program 2.3.d
// MosaicMeister 1

%ffp

ctl[0]: "Amount of tiles", range=(2,40)

ForEveryTile:
{
if (Z != 4)                    // is the alpha channel non-existent?
        {
        ErrorOk ("This filter does not work\non background layers!");
        enableCtl(CTL_OK, 1);   // disable OK push button
        }
else
        enableCtl(CTL_OK, -1);  // enable OK push button

return false;
}

R, G, B:   src(x-x%(X/ctl(0)), y-y%(Y/ctl(0)),z)
A:         x%(X/ctl(0))==0 || y%(Y/ctl(0))==0 ? 0 : 255
```

Let's take a look at the alpha channel code first. It basically checks if the current pixel is the first horizontal or vertical pixel (pixel number 0 in x%amount-of-tiles) of a tile. If it is, the pixel is set to full transparency or else it is set to full opacity.

The ForEveryTile handler checks if the current layer is a background layer or any other layer. Remember, in an RGB-image, the background layer has three channels and the other layers have four channels. If the filter is run on a background layer (Z=3), then an error window appears, informing the user about the problem. The program then disables the OK push button, so the user cannot apply the filter. If the filter is run on a normal layer, the Ok push button is enabled (if previously disabled by the above situation) and the filter can be applied.

This example was chosen for didactical reasons. Of course, there are some flaws in the program, such as:

- the filter will not work with other color modes such as Graysale or CMYK (because it checks if the fourth plane is available)
- the filter should be able to apply a "normal" mosaic filter on background layers (we simply wanted to show the usage of message windows based on certain conditions)
- other flaws we are too ashamed to mention (just kidding!)

We converted the image to mosaic tiles and added transparent joints. We now expect the user to add a 3D-effect to the tiles by selecting `Layer -> Effects -> Bevel and Emboss` and color the background layer (or layer below the mosaic). An information window could explain these steps, as well.

WebTileMeister

This tutorial will demonstrate the usage of applying different effects to the preview window. The objective is to select an area from within the image half the size of the image's width and height, place it on the left/top side of the image and mirror it once horizontally and mirror this top area vertically. That way, the final image can be saved and used as a seamless background image in HTML-pages. The specialty of this filter is that we can toggle between two views in the dialog window: the original area and the mirrored image.

The first thing we will do is simply mirror the top-left area horizontally and vertically:

```
R, G, B, A:     //do not forget to mirror alpha channel!
src( x<X/2 ? x : X-x-1, y<Y/2 ? y : Y-y-1, z)
```

Explanation: if the current pixel is on the left side of the image (`x < X/2`), take the current pixel (`x`), else (current pixel is on the right side of the image), take the adjacent mirrored pixel from the left side (`X-x-1`). Use the same analogy with the vertical pixel position (`y`).

The source area is set to the top/left, but what if we want to address any pixel from the right and bottom parts of the image? The best thing to do is to implement an offset to the `src()`-function elements, which is set by two scroll bars. The scroll bars' ranges are set to (0, 100), whose values represent percent values (0% means no offset, 100% means offset the area to be mirrored up to the border of the image):

```
%ffp

ctl[0]: "Horizontal offset (in %)", range=(0,100)
ctl[1]: "Vertical offset (in %)", range=(0,100)

R,G,B,A:
src(  (x<X/2 ? x : X-x-1)+ scl(ctl(0),0,100,0,X/2) ,
      (y<Y/2 ? y : Y-y-1)+ scl(ctl(1),0,100,0,Y/2) , z)
```

Note that the scroll bars are WYSIWYGged with the `scl()`-function. The "old" range (0,100) is scaled to (0, X/2) respectively (0, Y/2). We only allow a maximum offset of half an image, since the area to be mirro-red is half an image wide. That was the code for the mirroring – we need the code to show the area we are copying from: while the area to be mirrored stays at it is, the rest is grayed out.

```
%ffp

R, G, B:   ( x < 0 || x > X/2 || y < 0 || y > Y/2 ) ? i/2 : c
A:         a
```

The above program tests if the current pixel is out of an area half the width and half the height. If it is, it is grayed out with `i/2`, which represents half the pixel's current grayscale intensity, or else leave the pixel color unchanged. The next task is to integrate the offset with the range (0, X/2) by inserting a `scl()`-fun-ction (marked red):

```
%ffp

ctl[0]: "Horizontal offset (in %)"
ctl[1]: "Vertical   offset (in %)"

R, G, B:   ( x < ( 0 + scl(ctl(0),0,100,0,X/2) ) || x > ( X/2 + scl(ctl(0),0,100,0,X/2) ) ||
           y < ( 0 + scl(ctl(1),0,100,0,Y/2) ) || y > ( Y/2 + scl(ctl(1),0,100,0,Y/2) ) ) ?
           i/2 : c
A:         a
```

The final task is to combine both programs into one and use a check box to toggle between the tile source and the mirrored image.

```
// Program 2.3.e
// WebTileMeister

%ffp

ctl[0]: "Horizontal offset (in %)", range=(0,100)
ctl[1]: "Vertical offset (in %)", range=(0,100)
ctl[3]: CHECKBOX, "Show tile source"

R, G, B:
ctl(3) && doingProxy ?  // show tile source and is the filter applied to the preview window?
( x < ( 0 + scl(ctl(0),0,100,0,X/2) ) || x > ( X/2 + scl(ctl(0),0,100,0,X/2) ) ||
 y < ( 0 + scl(ctl(1),0,100,0,Y/2) ) || y > ( Y/2 + scl(ctl(1),0,100,0,Y/2) ) ) ? i/2 : c
:
src( (x<X/2 ? x:X-x-1)+scl(ctl(0),0,100,0,X/2), (y<Y/2 ? y:Y-y-1)+scl(ctl(1),0,100,0,Y/2), z)

A:
ctl(3) && doingProxy ? // show tile source and is the filter applied to the preview window?
a :
src( (x<X/2 ? x:X-x-1)+scl(ctl(0),0,100,0,X/2), (y<Y/2 ? y:Y-y-1)+scl(ctl(1),0,100,0,Y/2), z)
```

The first conditional operator checks if the check box is active and if the dialog window (filtering the preview window) is active. Can you think why the check for the dialog window is done? In general, the user will not want to apply the ”show tile source” filter to the final image – the objective of the filter is to mirror an area from within the image horizontally and vertically: background images for HTML pages.

Do you think you can create now a filter where you can define an area of any width and height? Think of this task as a homework. Tip: use the modulo operator!

2.3.7 Text string functions

FilterMeister has been implemented with some string functions.

strlen(text string)

Returns the integer length of the text string. Note that the text string does not support substrings (such as `!A` for author, etc.).

Examples:

```
dLength=strlen("Computer");     // assigns the value 8 to the variable dLength

Info("The word 'Image' is %d characters long", strlen( "Image" ));
```

strcpy(ts1, ts2) , strncpy(ts1, ts2, n)

The above functions copy the string `ts2` in `ts1` completely respecitvely only the first `n` characters. Both functions return a string.

Examples:

```
Info("%s", strcpy("One", "Two"));            // replaces "One" with "Two" and returns "Two"

Info("%s", strncpy("Coat", "Grandpa", 1));    // replaces first character of Coat with 'G'
```

strcat(ts1, ts2), strncat(ts1, ts2, n)

The concatenate functions append the text string `ts2` at the end of the string `ts1`. In the case of `strncat()`, the first `n` characters of the string `ts2` are appended at the end of `ts1`.

Examples:

```
Info("%s", strcat("Bye-", "bye"));            // Returns "Bye-bye"

Info("%s", strncat("Drop ", "method", 2));    // Returns "Drop me"
```

strcmp(ts1, ts2), strncmp(ts1, ts2, n)

These functions compare the (lexicographical) similarity between two strings. The `strncmp()`-function compares explicitly the first n characters of the strings. The functions return an integer:

0	true; both strings are identical
1	false; ts1 is greater than ts2
-1	false; ts2 is less than than ts2

Examples:

```
Info("%d", strcmp("FilterMeister", "FilterMaster"));    // returns false (1)


Info("%d", strncmp("Filtermania", "Filtercraze", 6));   // returns true

Info("%d", strcmp("Boy", "Girl"));                      // returns false (-1)
```

appendEllipsis(text string), stripEllipsis(text string)

These functions append or remove Ellipsis (...) to or from the end of the text string. While `appendEllipsis()` appends an ellipsis to the string (regardless if the string already has an ellipsis), the `stripEllipsis()` removes the ellipsis only if the string includes an ellipsis. Both functions return a string.

Examples:

```
Info("%s", appendEllipsis("Wait a minute"));    // returns "Wait a minute..."

Info("%s", stripEllipsis("Hello..."));          // returns "Hello"
```

formatString

2.3.8 Run-time functions

Run-time functions help to change current dialog window or user control settings in the standalone filter. For example, one might want to add a check box which, if checked, calls a run-time function to update a scroll bar setting while the user moves another one (e.g., a check box locks two scroll bars; if the user moves one, the other one moves automatically). Another example would be a multi-filter plug-in which can have different dialog window sizes or layouts.

There are special chapters later in this manual you can consult:

3.1.2	Dialog window run-time functions
3.4.2	User control run-time functions

2.3.9 standard i/o functions

clearerr, ferror,
fopen, freopen, fclose, fcloseall,
feof, fscanf, sscanf,
fseek, ftell,
fwrite, fread,
getc, putc, fgetc, fgets, fputc, fputs,
fflush, flushall,
remove,
rename, rewind,
sprintf, fprintf,
tmpfile, fmpnam, ungetc

2.4 Extended program elements

In this last chapter of the FF+ programming language we will explain the usage of advanced FM filter program elements such as shortcuts in math terms, different loops, special conditions, switch statements and other elements.

2.4.1 Assignment operators

In some cases, one might want to constantly change the value of a variable (e.g., a counter variable). There are various ways to do this:

```
iCount = iCount + 1;    // Adds 1 to the current value of the integer iCount
```

There are three possible shortcuts for the above term:

```
iCount++;       // Increments the integer variable iCount by 1 (postfix)
++iCount;       // Increments the integer variable iCount by 1 (prefix)
iCount += 1;    // Shortcut for iCount=iCount+1
```

If one wants to change the above counter variable by a value other than 1, one can apply one of these terms:

```
iCount = iCount + 12;   // Adds 12 to the current value of the integer variable iCount
iCount += 12;           // dito
iCount += iChange;      // iCount is added by the value of the variable iChange
```

The same applies to subtracting values:

```
iCount = iCount -1;     // iCount is subtracted by 1
iCount--;               // iCount is subtracted or decremented by 1 (postfix)
--iCount;               // iCount is subtracted or decremented by 1 (prefix)
iCount -=10;            // iCount is subtracted by 10
iCount -= iChange;      // iCount is subtracted by the value in iChange
```

The postfix and prefix terms have two different functions. The call of a value is like a function returning a value. Incrementing a variable (postfixed) returns the value of the variable and then increments it. Incrementing a variable (prefixed) increments the variable first and then returns the value.

Example:

```
%ffp

ForEveryTile:
{
int iCount = 10;                        // declare and initialize variable iCount

Info("iCount = %d", iCount++);          // returns 10 and then increments iCount
Info("iCount = %d", iCount);            // returns 11
Info("iCount = %d", --iCount);          // decrements iCount and returns 10
iCount +=12;                            // adds 12 to iCount
Info("iCount = %d", iCount);            // returns 22

return false;
}
```

There are more so-called *binary assignment operators* (as in += or -= shown above):

`a *= b;`	Multiplies a by b and puts result in a
`a /= b;`	Divides a by b and puts result in a
`a %= b;`	Modulates a by b and puts result in a
`a >>= b;`	Bit-shifts a by b bits to the right and puts result in a
`a <<= b;`	Bit-shifts a by b bits to the left and puts result in a
`a &= b;`	a is (bitwise) ANDed with b and result is put in a
`a |= b;`	a is (bitwise) ORed with b and result is put in a
`a ^= b;`	a is (bitwise) XORed with b and result is put in a

Note that the binary assigment operators return the result of the operation, which is in fact analogous to the prefixed increment/decrement operators.

2.4.2 Loops

Loops are usfeul for iterative filters. For example, one could apply the same filter a couple of times respectively different filters at once. There are various loops available for different tasks, depending for example if the filter should at least be applied once or not at all. Although rarely used in the RGBA-handler, loops are pretty valuable when working in the ForEveryTile handler. Before compiling any filter with loops, we advise you to save the program – there are often cases infinite loops are programmed just because of a small false condition or variable.

for (i_term, c_term, r_term)
{ Expression(s); }

The structure of a for() function is based on different terms:

i_term	initialize variable(s)
c_term	set conditions of the loop; if condition is true, run the expression in the following line(s)
r_term	re-initialize variable(s)

If the expression is multi-lined (do more tasks in the loop), use the brackets {} to define an expression block, or else only the line following the for()-function will be executed by the loop. Let us look at an example:

```
for (x=0; x<xmax; x=x+1)
        pset(x,10,0, 255);      // draw a horizontal line at y=10 in the red channel
```

The variable x is initialized with the value 0. The condition for this loop to execute the following line is that x is less than the current tile's (or image's) width xmax. If this condition is true, call the following line and then reinitialize the variable by incrementing it. This loop is repeated until x is equal or greater than xmax.

Take a look at the following program. All it does is mirror the image vertically, but in the ForEveryTile handler:

```
%ffp

ForEveryTile:
{
for (y=0; y<ymax; y++)                        // vertical loop
      for (x=0; x<xmax; x++)                  // horizontal loop
             for (z=0; z<Z; z++)              // channel loop
             {
                   pset(x,y,z, src(x,Y-y-1,z);  // mirror image vertically
             }
return true;
}
```

while (c_term)
{ Expression(s); }

The while() loop-function repeats the expression (in the following line or the lines between the brackets) while the conditional term c_term is true. Never forget to include a line within the brackets which can make the conditional term false, or else your filter program will be stuck in an infinite loop. The expression(s) within the loop are only executed if the expression stays true. Examine the following program:

```
%ffp

ForEveryTile:
{

x=0;                            // initialize variable x
while ( x<X )                   // repeat loop while x is less than image's width
{
        pset(x,20,1,0);         // sets pixel at y=20 in green channel to 0
        pset(x,Y-20,1,0);       // sets pixel at y=Y-20 in green channel to 0
        x++;                    // increment x!
}
return true;                    // apply filter to image
}
```

do { Expression(s); }
while (c_term);

The above loop type differs from the other two loop types for() and while(): the expression(s) is/are executed at least once before the conditional term is reached. Only then will the conditional term decide if another loop is executed. The following program applies a filter over and over again only if the user clicks on Yes in the message window.

```
// Program 2.4.2.a
// Applies a simple four-neighbour blur filter

%ffp

ForEveryTile:
{
int iCount=1;           // define and initialize the variable iCount

do {
for (y=0; y<Y; y++)
for (x=0; x<X; x++)
for (z=0; z<Z; z++)
        pset(x,y,z,
        (pget(x-1,y,z)*2+pget(x+1,y,z)*2+pget(x,y,z)*4+pget(x,y+1,z)*2+pget(x,y-1,z)*2)/12);
// the following line tells the user the filter was applied iCount times (and increments it,
// in case the user wants another filter application) and asks if another filter application
// is desired.
} while ( YesNo("Filter applied %d times!\nAgain?", iCount++)==IDYES);

return true;    // apply filter to image
}
```

The filter itself contained in the `pset()`-line is a simple convolve filter as in cnv(0,2,0,2,4,2,0,2,0,12). The reason we did not use the `cnv()`-function is that it always fetches the pixel infos from the source image. We want to apply the filter on the already filtered image in the output image buffer, so we had to use the `pget()`-function.

These were only the basics on loops. Please consult any C programming language reference for more details on loops.

2.4.3 Selection statements

It is often interesting to present the filter user the ordeal of choice: multiple filters in one single plug-in or simple conditional expression. We have learned the conditional operator (a ? b : c), which is in fact a simple and compact if-then-else statement. The two selection statements if-then-else and switch are explained in this chapter.

if-then-else

The basic structure of this statement reads as follows:

```
if (condition)          // if the condition within the parenthesis is true,
then {expression 1}     // then execute the expression 1,
else {expression 2}     // else (if the condition is false), then execute expression 2
```

Take a look at MosaicMeister 1 (program 2.3.d), where we used a simple condition to test if the current layer had three or four channels. If it did not have four channels, an error message window appeared. If it did have four channels, everything was okay (only the OK push button had to be activated). Also take a look at the tutorial section 3.4 later in this manual.

switch

While the if-then-else statement or the conditional operator are useful for selection between two choices, the switch statement can be used to select between many choices! The basic structure of the switch statement reads as the following:

```
switch (iTest)                  // the variable iTest is evaluated:
{
case 1:     expression;         // if iTest=1 execute the expression(s) following
            break;              // stop here and get out of the switch-statement

case 2:     expression;         // if iTest=2 execute the expression(s) following
            break;              // stop here

default:    expression;         // if iTest is not equal to 1 or 2, execute the expression
            break;              // stop here
}
```

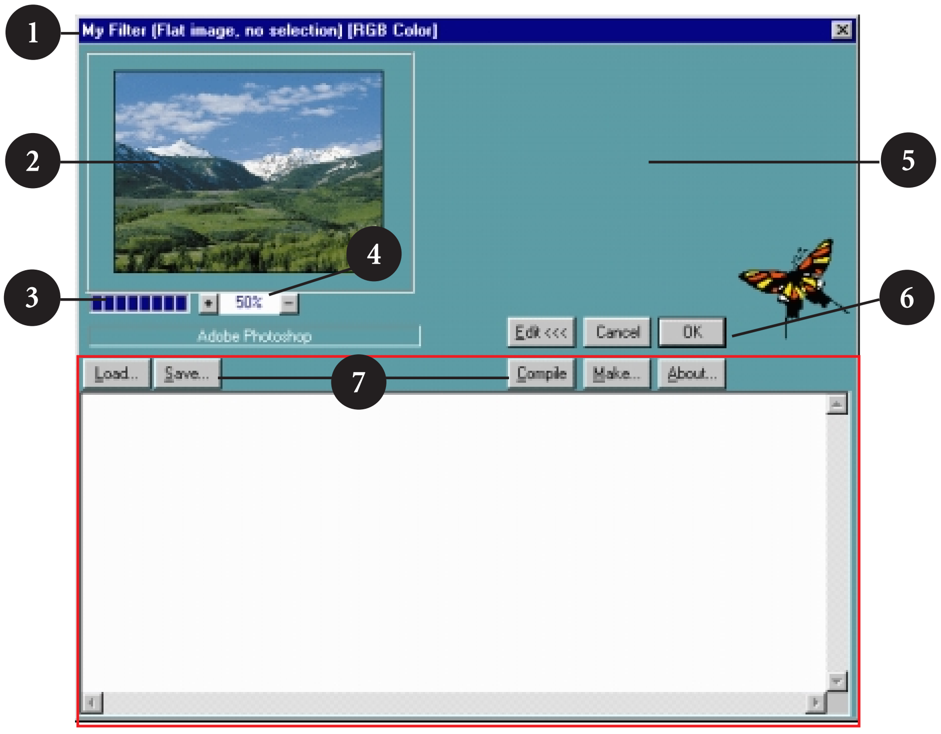
2.4.4 _eval_FFP{}

3. The User Interface

In this chapter, we will introduce you the possibilities of the user interface design. FilterMeister offers you a variety of user controls you can use. You can change the position and "looks" of each user control. We will also show you how to dynamically change user control properties with other user controls – you can even create filter animations in real-time! Finally, window background design is described.

3.1 The dialog window

When FilterMeister is invoked from the menu `Filter`, the following window appears:



1 Window Title bar, contains filter name, filter case and image mode by default
2 Preview Window
3 Progress bar; below the name of the host application
4 Zoom factor in % with zoom in (+) and zoom out (-) buttons; clicking on zoom buttons with the `SHIFT`-key depressed zooms to max. respective min. factors
5 Dialog background
6 Buttons for opening editing window (see red border), cancelling or applying (`OK`) the filter
7 Buttons for loading, saving, compiling the source code, making a standalone filter and calling the About window (see chapter 4 – File input/output)

The dialog window can be redesigned to meet your needs. You can:

– redefine the dialog window's background with a color, gradient or image,
– redefine the dialog window's size or region and position (when called from the filter menu),
– define, position and resize user controls (see chapters 3.3 and 3.4)

3.1.1 Dialog window properties

The dialog window properties are part of the FilterMeister program's head (see program structure in chapter 2.1). These property keys can change the visual appearance of the filter dialog, they also describe with which host programs, filter cases or image modes the filter will work. If these keys are not explicitly defined, FM will use the default key values. The keys are set like:

```
Key:    Property1, Property2,...
```

Dialog

This dialog key describes the visual appearance of the dialog window. You can redefine the size and background color or background image plus a couple of other nifty goodies. These are the following properties of the key `Dialog`:

Pos = relative (x, y)

The property `Pos` positions the dialog window to a new coordinate `(x,y)` in relation to either the whole screen or the host's main window (i.e., the client). The relative can be `Screen` or `Client`; instead of an `(x,y)`-coordinate, you can *center* the window.

Examples:

```
Dialog: Pos=Screen(33,38)       //coordinates are measured in DBU's!
Dialog: Pos=Client(CENTER)      //default
```

Size = (w, h)

The property `Size` determines the width and height of the dialog window in dialog box units (DBUs).

Example:

```
Dialog: Size=(200,200)
```

Drag = Background

This property lets the user drag the window either by the title bar or the background or not drag at all.

Examples:

```
Dialog: Drag = Titlebar         //default
Dialog: Drag = Background
Dialog: Drag = None
```

Titlebar/NoTitleBar

These properties set respectively delete the title bar in the dialog window. Keep in mind that deleting the title bar causes a vertical shift scroll of the dialog. This also affects the dialog appearance if you define a `Region` (see below).

Examples:

```
Dialog: NoTitlebar
```

Text

This property defines the text content which is placed in the title bar. You can use text substrings (e.g., !H to include the host application's name in the title bar), but no escape sequences.

Examples:

```
Dialog: Text="!A's Pudding Filter"      // !A is replaced by the Author key

Dialog: Text="!T (!f) [!M]"             //default title bar text
```

Region = REGION(a, b, c, d, ...)

This property defines the visual region of the dialog window. You have the possibility to combine "primitive" regions (such as circles, rectangles, polygons, etc.) to create new dialog designs. In order to create awesome dialog window designs, load a background image with the property `Background` described later in this chapter. There are five "primitive" regions you can use (all measurements in DBUs):

`RECT(x1, y1, x2, y2)`	Rectangle; (x1,y1) defines the upper-left edge and (x2,y2) the lower-right edge of the rectangle

`RRECT(x1, y1, x2, y2, w, h)`	Rounded rectangle; (x1, y1) defines the upper-left edge and (x2, y2) the lower-right edge of the rectangle. (w, h) defines the width and height of the ellipse for the rounded corners
`CIRCLE(x, y, d)`	Circle; (x, y) defines the upper-left corner of the bounding box (*not the circle's center!*) and d defines the circle's diameter
`ELLIPSE(x1, y1, x2, y2)`	Ellipse; (x1, y1) defines the upper-left corner and (x2, y2) the lower-right corner of the bounding box
`POLY(fill_mode, x1,y1, x2,y2, ...)`	Polygon; The fill_mode is either `ALTERNATE` or `WINDING`. At least three coordinate pairs must be defined.

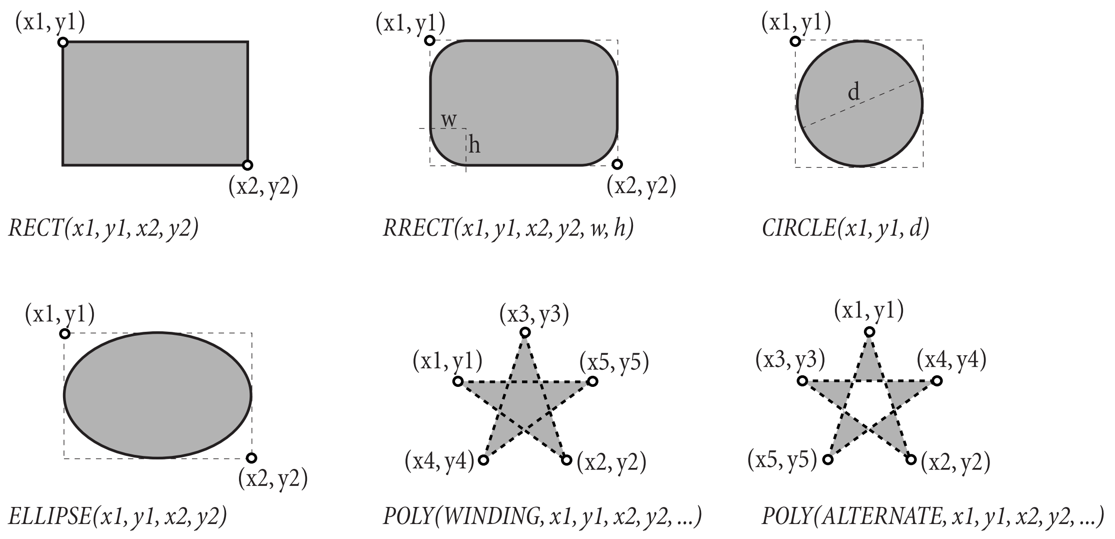


Fig. 3-1-1-a: Region primitives

The primitives can all be combined with the following operators:

Region1 - Region2	Difference operator; subtracts the second region from the first region
Region1 & Region2	Intersection operator; calculates the area where both regions intersect/overlap
Region1 ^ Region2	Disjoint union operator; calculates the area where both regions do not intersect
Region1 | Region2	Union operator; adds both region areas together

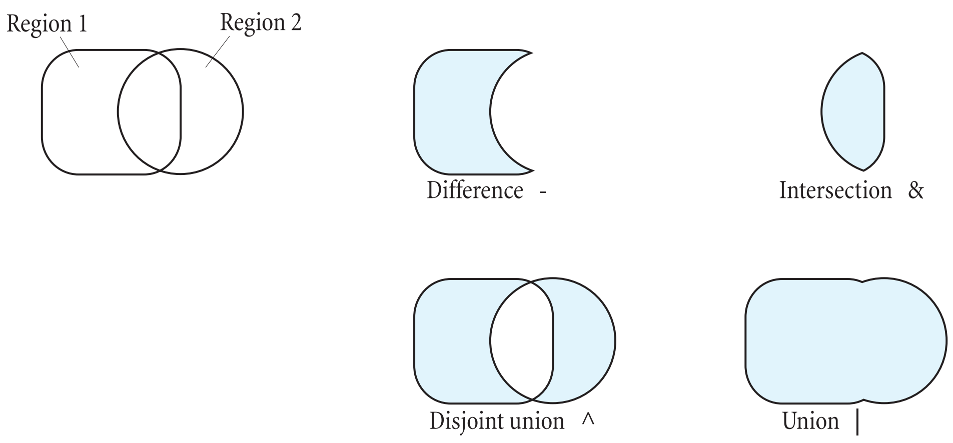


Fig. 3-1-1-b: Operator examples

It is also possible to combine more than two regions. However, keep in mind that each operator has a different priority. The difference operator has the topmost priority whereas the union operator has the leastmost priority. Of course, the usage of parentheses is possible.

Example:

```
Dialog: Region= RRECT(0, 0, 320, 240, 20, 20) - CIRCLE(130, 90, 60)

Dialog: Region= (RECT(10, 10, 100, 100) | RECT(140, 10, 230, 100)) - RECT(10, 50, 230, 60)
```

Color = RGB(255,0,127)

Choose any color from Appendix A to color the background of the dialog window. This property overrides the property `Gradient`, if defined after it.

Examples:

```
Dialog: Color = Papayawhip

Dialog: Color = Cornflowerblue
```

Gradient = (first color, second color, 'h' or 'v')

Instead of defining a colored background, you can define a horizontal or vertical gradient. This property overrides the property `Color`, if defined after it. *Tip: click on the Edit push button to close the editing window to see how the gradient will look like correctly.*

Examples:

```
Dialog: Gradient = (Brass, Semisweetchocolate)          //use vertical gradient by default

Dialog: Gradient = (DarkTurquoise, Indianred, H)
```

Image = "filename.bmp", *scaling mode*

Should you sympathize with image backgrounds, this is the property for you. A background image can be scaled in three different ways:

`Exact`	Positions the image once in the upper left corner
`Tiled`	Tiles the image vertically and horizontally (default)
`Stretched`	Stretches the image according to the current dialog window size

When defining a background image, you have to take into consideration that some users have different screen modes such as an 800 × 600 resolution or small/large fonts, etc. You will have to test your filter in different screen modes. Depending on your background image, experiment with the tiled or stretched scaling modes. The easiest method of integrating background images is the implementation of tiled images (see screenshot in the User Guide in page 22).

Embedding image files

FilterMeister does not automatically integrate the image files you specify in the final plug-in file. This procedure, called embedding, has to be done with a new key. The syntax looks like the following:

```
Embed:  Bitmap = "Filename1.ext", Bitmap = "Filename2.ext", ...
```

There is a good reason why embedding has to be done explicitly. If you think of a series of filters you programmed, all plug-ins having a common image background, then you could save disk space by defining only the image to be loaded and by including the image in your filter package.

If each filter dialog window has a different background, then you would need to embed the images in each filter.

Tutorial section 3.1.1

In this section, we will build two dialog windows. The first dialog uses the default region without any special goodies. The second dialog is more advanced and includes a background image, a non-rectangular region and special user controls.

Simple dialog

This dialog will include a tiled image background, so we will be needing a background image in BMP format (8-bit or 24-bit). In this example, the 80 × 80 RGB image in the margin of this page is used.

If you have not already done so, start up your graphics application, open up any image and call FilterMeister from the filter menu. Call the editing window by clicking on the `Edit >>>` push button and type in the following:

```
%ffp

Author: "Mark Twain"
```

```
Title:  "Tom Sawyer"
Dialog: Text="Read !T by !A!!"
```

Look what happens to the title bar when you click on the Compile push button. Besides having powerful functions, FM seems to have deep knowledge in American literature!

Let's add some color to the dialog background. Change the last line to:

```
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise
```

and compile the program. For a more clearer view, close the editing window by clicking on the `Edit <<<` pushbutton (you can also use the key shortcut `ALT-E` to switch between editing window and filter dialog window). Experiment with different colors as listed in Appendix A.

A simple color may be too boring or conservative. Let's change the color to a two-colored gradient by editing the last line of our program:

```
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise, Gradient=(Indianred, Goldenrod)
```

The gradient adapts itself to the current window size, so be sure to always close the editing window to see how the dialog will look like when created as a standalone filter. Instead of a vertical gradient, we can define a horizontal gradient by adding the gradient-property h:

```
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise, Gradient=(Indianred, Goldenrod, v)
```

There is no need, of course, to include the background color property since it is overdrawn by the gradient. You should also know that the last dialog background property overrides the ones defined before. In the following case, the color succeeds the gradient and has thus higher priority:

```
Dialog: Text="Read !T by !A!!", Gradient=(Indianred, Goldenrod, v), Color=Mediumturqouise
```

If you think that a gradient is still too conservative, then let's add a tiled background image with:

```
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise, Image="tut311a.bmp"
```

There is no need to include a directory path for the Image property if the image file is in a directory where the `PATH` or `FM_PATH` variables (defined in the `AUTOEXEC.BAT` file) point at. If this is not the case, FM will not be able to load the image – you will get an error message and the current background color or gradient is set. Colors and gradients do not have a higher priority over the Image property if defined after it.

The image loaded is tiled throughout the dialog window by default. If you want, stretch it to see horrendous results:

```
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise, Image="tut311a.bmp", Stretch
```

Call the same program in different screen sizes: 640 × 480, 800 × 600 and 1024 × 768 to see how the tiling will appear. The complete FM program should look like the following:

```
// Listing 3-1-1.a
// Simple dialog layout

%ffp

Author: "Mark Twain"
Title:  "Tom Sawyer"
Dialog: Text="Read !T by !A!!", Color=Mediumturqouise, Image="tut311a.bmp"
```

Extended dialog

The following tutorial will teach you to create the fanciest and modern dialogs. We will load a background image, replace the standard user controls `OK`, `Apply` and `About` and redefine the region of the dialog. We will add a drag option to the dialog since we are losing the title bar in the following case.

The **first step** defined the region or layout of the dialog. *These are the current settings for a standard dialog layout in 96 dpi screen resolution:*

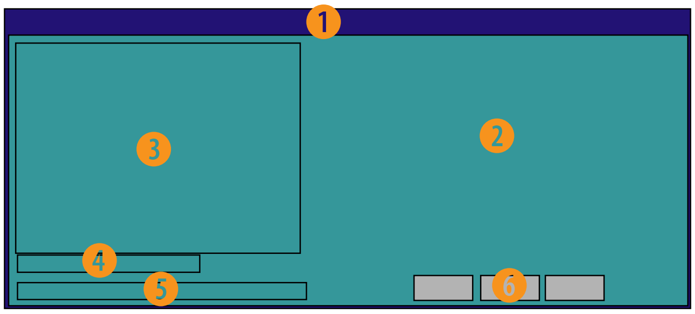


Fig. 3-1-1-c: Dialog metrics

1)	Size of the complete dialog:	392 × 158
	Title bar height:	14
	Width of side and bottom borders:	2
2)	Size of bare dialog without border and title bar:	388 × 142

3) Preview window; Left top corner at (5, 18), Right bottom corner at (169, 129)
4) Progress bar and Zoom bar; Left-top corner at (8, 130), right-bottom corner at (112, 138)
5) Host program static text (framed); Left-top corner at (7, 143), right-bottom corner at (153, 173)
6) Push buttons*:
Cancel – left-top corner at (271, 140), right-bottom corner at (306, 154)
OK – left-top corner at (308, 140), right-bottom corner at (342, 154)

All units are measured in DBUs (dialog box units) and not in pixels.
When placing user controls, the origin is not the window dialog but the bare dailog without border and title bar. This means, for example, that the OK button's position (as set in the Pos property) is (306,126).

* *There is no need to change the Edit push button's properties since it is deleted when FM creates a standalone filter.*

Figure 3-1-1-d shows our new dialog design. The construction is based on the union of one rectangle (broken line) and two rounded rectangle regions (green), from which another rectangle region is subtracted (red).

RECT(12,14,340,142)
RRECT(2,14,22,141,10,10)
RRECT(180,14,350,150,10,10)
RRECT(2,140,181,152,10,10)

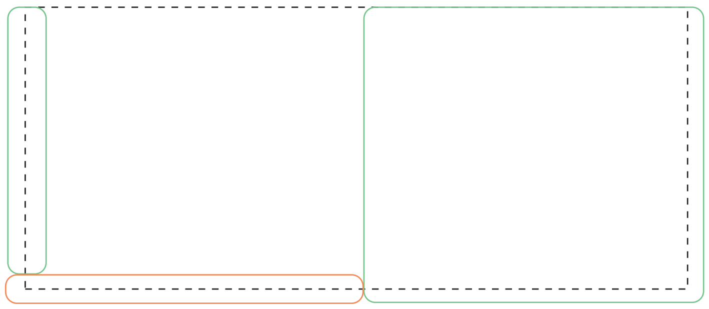


Fig. 3-1-1-d: User-defined dialog

The dialog property `Region` looks like the following:

```
Dialog:
Region=( RECT(12,14,340,142) | RRECT(2,14,22,141,10,10) | RRECT(180,14,350,150,10,10) )-
RRECT(2,140,181,152,10,10)
```

Due to the fact that the title bar is missing, the dialog window cannot be dragged around. We simply defined the background to be draggable:

```
Dialog: Drag = Background
```

In the second step we made a screenshot of the new dialog layout by pressing the keys `ALT-Print Screen` and creating a new image in Photoshop. After pasting the dialog screenshot into the new image, we used it as a template for a redesign. The new design also included three pseudo-push buttons for the actions Apply, Cancel and About. The image was saved in the BMP format.

The third step involved the loading and embeding of the image file. The FM program is added by the following commands:

```
Dialog: Image="tut331b.bmp"
Embed:  Bitmap="tut331b.bmp"
```

Remember, embedding a bitmap makes the standalone filter's size larger. If you do not wish to embed, don't forget to include the file when packing your filter files.

The fourth and last step involves the redefinition of the push buttons `OK`, `Cancel` and `About`:

```
ctl[CTL_OK]: OWNERDRAW, Pos=(186,114), Size=(48,18)
ctl[CTL_CANCEL]: OWNERDRAW, Pos=(236,114), Size=(62,18)
ctl[48]: OWNERDRAW, Pos=(300,114), Size=(40,18), Action=About
```

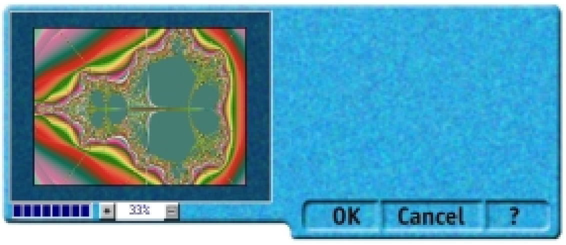
The OWNERDRAW user control is invisible to the user, but actionable. Note that we did not include the actions for OK and Cancel. By defining the control variables CTL_OK and CTL_CANCEL, all actions are preserved.

The complete FM program looks like this:

```
// Listing 3-1-1.b
// Extended dialog layout

%ffp

Category: "FilterMeister"
Title: "Extended Dialog"


Dialog: Drag=Background, Image="Tut331b.bmp", Region=( RECT(12,14,340,142) |
RRECT(2,14,22,141,10,10) | RRECT(180,14,350,150,10,10) ) - RRECT(2,140,181,152,10,10)

Embed: Bitmap="Tut331b.bmp"

ctl[CTL_OK]: OWNERDRAW, Pos=(186,114), Size=(48,18)
ctl[CTL_CANCEL]: OWNERDRAW, Pos=(236,114), Size=(62,18)
ctl[48]: OWNERDRAW, Pos=(300,114), Size=(40,18), Action=About
```

Click the button Make... to create a new file and restart your host program if necessary and go run that filter with the awesome design!

There is one problem, though. If you change the screen resolution, you will not get the same result. On a larger resolution (120 dpi), you will see that the background image is tiled. One way to get around this is to not embed the image and provide your users different background images. They would have to rename the file to the filename specified with the Image property.



3.1.2 Dialog Run-time Functions

In the past chapter, you learned how to define the dialog statically, which means that the looks of the filter will not change will setting user controls, zooming in the preview window, etc. Run-time functions help you change the appearance while the dialog window is active and are always used in handlers (e.g., in the `ForEveryTile` handler). For example, multiple filter plug-ins may have different dialog layouts or background images. Run-time functions are used in the program body's handlers.

setDialogPos (fAbs, x, y, w, h)

This function repositions and/or resizes the dialog window. `x` and `y` are the new coordinates while `w` and `h` represent the new width an height. `fAbs` is a boolean value (true or false) – if it is set to true, the `x`, `y` values represent absolute values. Otherwise, they represent absolute values but relative to the host's main window area. If `x` and `y` are set to `-1`, the dialog window is centered within the host client area or within the screen (depending on `fAbs`). If w or height is negative, the width and height of the dialog window will not be changed.

Examples:

```
setDialogPos ( true, 200, 100, 400, 300 )  //repositions and resizes dialog window
setDialogPos ( false, -1, -1, 320, 240 )   //centers dialog according to host client area
```

setDialogText ("title text")

This function changes the text in the dialog window's title bar. It is allowed to use text substrings (as in `!A` for the Author's text definition), but no escape sequences within the text string.

Examples:

```
setDialogText ("This is my filter!!");
setDialogText ("!A is the creator of this filter");
```

setDialogTextv ("title text", term1, term2, ...)

This special function allows you to use arguments as in the `printf`-function. The arguments can be any numerical or string variable. It is allowed to use text substrings (as in `!A` for the Author's text definition), but no escape sequences within the text string.

Examples:

```
setDialogTextv ("The author !A says that %d + %d equals %.2f", 4, 3, 4.0 + 3.0);
setDialogTextv ("X = %d", X);  //Zoom in and out of the preview window – watch the title bar
```

setDialogColor (color)

This function changes the background color. You can use all color specifications according to Appendix A.

Examples:

```
setDialogColor (Red);
setDialogColor (HTML.DarkTan);
```

setDialogGradient (color 1, color 2, direction)

This function sets or changes the gradient. The first two properties set the new colors and the third property sets the gradient's direction (either 0 for vertical or 1 for horizontal).

Examples:

```
setDialogGradient (Firebrick, mistyrose, 0);    //vertical gradient
setDialogGradient (Red, Blue, 1);               //horizontal gradient
```

setDialogImage ("filename.ext")

It is also possible to change the dialog window's background image by calling the above function. Currently, only the BMP file format is supported.

See example in the following function `setDialogImageMode`.

setDialogImageMode (Image Mode, *Stretch Mode*)

This function changes the image or scaling mode, which specifies if the image is tiled or stretched. If the loaded image file is stretched, a stretch mode must also be specified. These are the image modes:

0	exact (currently not working correctly in FilterMeister)
1	tiled
2	stretched

These are the stretch modes, which are explained in *chapter 3.1.1 Dialog window properties*:

BLACKONWHITE, WHITEONBLACK, COLORONCOLOR

Examples:

```
%ffp
ctl[0]: CHECKBOX, "Toggle dialog image", FontColor=Black

ForEveryTile:
{
if (ctl(0))
        {
        setDialogImage ("Image1.bmp");
        setDialogImageMode (1,0);
        }
else
        {
        setDialogImage ("Image2.bmp");
        setDialogImageMode (2, BLACKONWHITE);
        }
return false;
}
```

setDialogDragMode (DragMode)

This function changes the drag mode of the dialog window. You can choose from the following:

0	Title bar (default)
1	Title bar and dialog background area
2	None

Example:

```
setDialogDragMode(1);
```

3.2 Filter identification

Each filter differs from others by its title, category, filename, etc. Although you are free to place these identification keys anywhere in your FM program, we recommend to place them at the beginning, because they are part of the program's head. All filter identification keys are described by a string value. Look at the following example:

```
%ffp

Category:      "FM Power Filters"         //default: "FilterMeister"
Title:         "Freeze..."                //default: "My Filter..."
Author:        "Alex the Hunter"          //default: none
Copyright:     "© 1999 by AFH Systems"    //default: none
Version:       "V 1.1"                    //default: none
Filename:      "FM_Freeze.8bf"            //default: "MyFilter.8bf"
Description:   "This filter will create an ice effect on your image" //default: none
About:         "Plug-in for !H only!!"    //default: "!t Plug-in !V\n!C\n!c\n!A"*
```

- The text string size can handle up to 255 characters.
- If you do not specify any filter identification key values, the default values as commented above are used.
- Only the About key supports substrings such as:

!!	exclamation point
!A	text specified by the Author key
!C	text specified by the Category key
!F	text specified by the Filename key
!M	current Image Mode (as in ”RGB Color”)
!T	text specified by the Title key

- The keys Author, Copyright, Version, Description and About also support escape sequences such as:

\\	backslash
\n	new line
\t	move tab stop

... *(see Appendix B for more substrings and escape sequence definitions)*

The About Box

This window appears when you click on Archibald, the FilterMeister's dwarf placed next to the `OK` push button or when you select the filter in the `Help` → `About Plug-In` menu. You also have the option to define your own user control which calls the About Box.

By default, the following keys are used by FilterMeister:

```
Title:      "My Filter"
Category:   "FilterMeister"
About:      "!T Plug-in\n!C"
```

Currently, you can only change the text in the About box, the FM logo is always included. The text defined after the key `About` is centered in the window and supports escape sequences such as `\n` or `\t` and substrings such as `!T` or `!A`.See Appendix B for the escape sequence and substring definitions.

The text can be defined in one line as follows:

```
About:   "This filter is Shareware and costs 1,000 US$. Register it at http://..."
```

The About text can also be defined in multiple lines, e.g.:

```
About:  "The filter !T is available at:\n"
        "http://www.filtermeister.com\n\nCopyright © 1999 by AFH Systems"
```

Note that the substring `!T` is replaced by the text defined in the Title key and that the escape sequence `\n` inserts a new line. Also consider the following:

```
About:  "Remember to pay"
        "me in cash"
```

A new line in the editing window will not lead to a multiple line text in the About box. The above text will show you one line only. Do not forget to use escape sequences for tabs, new lines, etc. You will also note that the words *pay* and *me* will appear as *payme*, since there is no space defined in the text. The above line should correctly read:

```
About:  "Remember to pay\n"
        "me in cash"
```

or for a single line output:

```
About:  "Remember to pay "
        "me in cash"
```

3.3 User control properties

User controls are interesting when the filter designer wants to give the user the freedom to set his or her own filter settings. Each user control has a variety of properties – the most common ones are:

```
Class          Text          Val           Range         Pos           Size
Color          FontColor     Line          Page          [No]Track     In-/Visible
En-/Disabled   Action
```

The following user control *classes* are available in FM:

```
SCROLLBAR           TRACKBAR            PUSHBUTTON          CHECKBOX
RADIOBUTTON         GROUPBOX            OWNER DRAW          LISTBOX
COMBOBOX            STATIC TEXTFIELD    FRAME               RECTANGLE
BITMAP              IMAGE               ICON                METAFILE
NONE                MODIFY
```

A user control can be defined anywhere in your filter program but oustide the handlers. We recommend to place them in the program head (*see 2.1 Program structure*). Here is an example for a user control definiti-on:

```
%ffp

ctl[0]: Class=STANDARD, Text="&Grey", Range=(50,200), Val=100, Pos=(250,50), Size=(80,10),
Color=blue, FontColor=Red, Line=10, Page=15, Enabled, NoTrack

R,G,B: ctl(0)
```

Note that the `STANDARD` class is a special case for a scrollbar. It includes a text label and a numeric edit control. You don't need to specify the terms `Class=` and `Text=` and be sure to specify the user control class as the first property. Within the text string, an ampersand (&) will highlight the following letter with an underline. The control can then be accessed by a combination of the `ALT`-key and the corresponding letter. `Range` specifies the lowest and highest possible value a user control can return. `Val` will initialize the user control to that value. `Pos` and `Size` reposition respectively resize the user control within the dia-log window. You can recolor the font and user control text background with any color you like (*see Appendix A for color definitions*). `Line` defines the amount of unit jumps when you click on the user con-trol's buddies and `Page` defines the amount of unit jumps when you click within the user control (except the handle). Finally, you can define if the user control is *enabled* or *disabled* and if *tracking* (update the preview window as you drag the handle of the user control) is on or off.

All numerical values for position, size are measures in DBUs (dialog box units) – the actual pixel size of such an unit depends on the system font size. You don't have to specify all of the above properties – if a property is not defined, the default value is used. The user control classes including their individual pro-perties are described in the next pages. *Note that the activation of a property depends on the user control class. For example, you can set a range other than (0,1) to a checkbox. A checkbox can only return 0 or 1, so the range does not change the return values.*

PROPERTIES FOR ALL USER CONTROLS

These properties can be set in all user controls except for the GROUPBOX user control:

Pos=(x,y)	The user control is repositioned to the coordinate x,y
Size=(w,h)	The user control is resized with a new width w and height h
Invisible	The user control is deactivated and invisible
Tooltip="Tip me!"*	When the mouse pointer is set over a user control, a yellow tool tip appears
Enable/Disable	The user control is enabled respectively disabled (Default: Enable)
Action*=APPLY**	When the mouse is pointed to and clicked over a user control, the action defined is executed (Default: PREVIEW)

Tip: When resizing or repositioning a user control and want to keep one default value, simply use an asterisk (*), as in Pos=(100,*) *or* Size=(*,22)

* The tool tip and action properties for the user control classes STATICTEXT, FRAME, RECT, IMAGE, BITMAP, ICON and METAFILE are available when the class-specific property NOTIFY is activated.

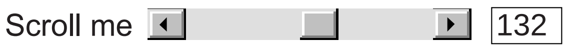
** These actions are available:

APPLY	Apply the filter to the image
NONE	No action
PREVIEW	Update the preview window
CANCEL	Cancel filter prosecution
EDIT	Open the editing window (only in programming mode, ignored in stand-alone filters)
ABOUT	Open the about window

These class-specific frame properties can be set in all user controls except for the STANDARD and SCROLLBAR user controls:

BORDER	Draws a black border around the user control
STATICEDGE	Draws a 3D-like sunken border around the user control
CLIENTEDGE	Draws a 3D-like sunken border (deeper than STATICEDGE) around the user control
MODALFRAME	Draws a 3D-button-like border around the user control

STANDARD (default)



The STANDARD user control is a simple scrollbar including a text label and a numeric edit control. You don’t need to specifiy the class property STANDARD. You can use the following properties with the class STANDARD (default values in parentheses):

`Text="&Text label"`	defines the text label next to the scrollbar (no Text)
`Range=(-15,300)`	sets the numerical range the scrollbar can return (0,255)
`Val=5`	initializes the scrollbar's value (0)
`Color=Aquamarine`	sets text background color in plain English format (CadetBlue)
`FontColor=#FF00FF`	sets font color in hexadecimal format (White)
`Line=10`	sets the left/right button jump unit (1)
`Page=6`	sets the scrollbar paging jump unit (10)
`Track/ NoTrack`	update preview window when dragging the scrollbar's handle (NoTrack)

You have the possibility of including these class-specific properties:

`HORZ`	horizontal scrollbar orientation (default)
`VERT`	vertical scrollbar orientation

Note: The background text `Color` *property also has some redraw problems. These problems will be resolved in future FM versions.*

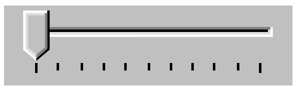
Examples:

```
ctl[0]: STANDARD(MODALFRAME, HORZ), Text="Plasma effect", Val=10, Disable

ctl[2]: STANDARD(VERT), "Pressure", Range=(-10,10), Val=0, Track

ctl[3]: "Threshold", Range=(-128,128), Val=0
```

SCROLLBAR

In contrast to the `STANDARD` class, using the property `SCROLLBAR` will invoke a scrollbar without text and edit control. All of the `STANDARD` properties except Text, Color and FontColor can be used.

TRACKBAR



This user control is a scrollbar with a different design. The filter designer can insert so-called ticks on one or two sides of the handle. You can use these additional properties with the class `TRACKBAR` (default values in parentheses):

`Range=(-15,300)`	sets the numerical range the scrollbar can return (0,255)
`Val=5`	initializes the trackbar's value (0)
`Color=Aquamarine`	sets text background color in plain English format (COLOR_SCROLLBAR)
`Page=6`	sets the trackbar paging jump unit (10)
`Track/ NoTrack`	update preview window when dragging the trackbar's handle (NoTrack)

You have the possibility of including these class-specific properties:

`NOTHUMB`	hides the thumb handle
`HORZ`	horizontal trackbar orientation (default)
`TOP/BOTTOM`	changes orientation of handle and ticks (default: `BOTTOM`)
`VERT`	vertical trackbar orientation
`LEFT/RIGHT`	changes orientation of handle and ticks (default: `RIGHT` with `VERT` active)
`BOTH`	sets ticks on adjacent sides of the handle according to orientation (`HORZ` or `VERT`)
`AUTOTICKS`	automatically sets tick marks according to range amount
`NOTICKS`	hides the tick marks of the trackbar
`TOPTIP`	displays the trackbar value beside the handle when dragging the handle

Note: Currently, trackbar values cannot be correctly read if the first item i1 of the Range=(i1,i2) property is greater than the second item i2. These problems will be resolved in future FM versions.

Examples:

```
ctl[0]: TRACKBAR, Size=(50, 20), Color = Red

ctl[5]: TRACKBAR ( VERT, BOTH, TOPTIP, AUTOTICKS), Range=(0, 10), Size=(30, 90),
        Color = Yellow
```

CHECKBOX

Sometimes, one needs a user control as a toggle for two or three values. The checkboxes come in handy for this kind of work. For example, you might want to program a blur filter where the user can select a horizontal or vertical orientation. You can use these additional properties with the class `CHECKBOX` (default values in parentheses):

`Text="Toggle"`	defines the text label next to the checkbox (no Text)
`Color=Aquamarine`	sets text background color in plain English format (transparent)
`FontColor=#FF00FF`	sets font color in hexadecimal format (White)

You have the possibility of including these class-specific properties:

`BORDER`	draws a border around the checkbox
`CLIENTEDGE`	draws a 3D-border around the checkbox
`STATICEDGE`	draws a 3D-border around the checkbox
`MODALFRAME`	draws a 3D-socket under the checkbox

`3STATE`	checkbox returns up to three values (third state is a grayed check)
`FLAT`	flattens the checkbox
`PUSHLIKE`	checkbox appears as a depressable pushbutton
`LEFTTEXT/RIGHTBUTTON`	either property sets text to the left of the checkbox
`LEFT/RIGHT/CENTER/`	aligns text within the checkbox's text area (`LEFT`)
`TOP/BOTTOM/VCENTER`	*VCENTER is a vertical center text alignment*
`MULTILINE`	allows word-wrapping within the checkbox's text area

If you are using the class-specific property `3STATE` , these are the values a function like `ctl(n)` returns (also applies if the class-specific property `PUSHLIKE` is defined):

0	not checked
1	checked (black check in the checkbox)
2	checked (gray check in the checkbox)

Examples:

```
ctl[5]: CHECKBOX, "Horizontal/Vertical Lines", FontColor=Darkblue
```

```
ctl[7]: CHECKBOX(3STATE), "White/Black/Gray"
```

PUSHBUTTON

Pushbuttons are usually involved in some dialog action. They can not only be used for general actions such as applying or cancelling a filter, but also for start/stop buttons when you animate the preview window as learned in chapter 3.2. You can use this additional property with the class `PUSHBUTTON` (default values in parentheses):

`Text="Push me!"`	defines text to be put inside the button's area (no Text)

You have the possibility of including these class-specific properties:

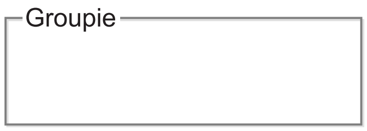
`FLAT`	flattens the pushbutton
`LEFT/RIGHT/CENTER/`	aligns text within the pushbutton's text area (`LEFT`)
`TOP/BOTTOM/VCENTER`	*VCENTER is a vertical center text alignment*
`MULTILINE`	allows word-wrapping within the pushbutton's text area

The function `ctl(n)`, where n is the user control index, returns on pushbuttons the values 0 (depressed) and 1 (pressed).

Example:

```
ctl[2]:  PUSHBUTTON (MULTILINE), "Press this button if you want to apply the filter",
         Size=(70,50), Action=APPLY
```

GROUPBOX



You may want to use the groupbox for two reasons:

- Design purposes: user controls are grouped for a main function. If you have a filter that simply draws colored lines (vertically or horizontally) on your image, you might put the orientation controls in one groupbox and the coloring controls in another groupbox.
- Radiobutton grouping: If you want to use radio buttons to give the user more options, you need the groupbox to define the group of these radio buttons. That way, you can implement different groups of radio buttons

You can use these additional properties with the class `GROUPBOX` (default values in parentheses):

`Text="Groupie"`	defines text to be put on the top line of the groupbox area (no Text)
`Val=5`	assigns a value to the groupbox (0)
`Color=X11.green`	sets text background color (transparent)
`FontColor=JAVA.pink`	sets text color (white)

Note that the groupbox is not actionable.

You have the possibility of including these class-specific properties:

`FLAT`	flattens the groupbox
`LEFT/RIGHT/CENTER`	text alignment (default: LEFT)
`GROUP`	define the end of the radio button group (see below)

Example:

```
ctl[1]:  GROUPBOX(CENTER), "Groupie", Size=(70,50), Color=CadetBlue, FontColor=Red
```

RADIOBUTTON

Sometimes the programmer wants to give the user the ability to select one item from a small group of (usually more than two) options. This can be easily done with radio buttons, list boxes and combo boxes. When implementing radio buttons, the first radio button is defined as the group start, while a groupbox sets the group's end (see example below). *Do not forget to activate one of the radio buttons with* `Val=1`.

You can use these additional properties with the class `RADIOBUTTON` (default values in parentheses):

`Text="Buttonize"`	defines the text label next to the radio button (no Text)
`Val=5`	assigns a value to the radio button (0)
`Color=Firebrick`	sets text background color (transparent)
`FontColor=tomato`	sets text color (white)

You have the possibility of including these class-specific properties:

`FLAT`	creates a border around the radio button
`PUSHLIKE`	radio button appears as a depressable pushbutton
`LEFTTEXT/RIGHTBUTTON`	either property sets text to the left of the radio button
`TOP/BOTTOM/VCENTER`	aligns radio button within the checkbox's text area (`VCENTER`) *VCENTER is a vertical center text alignment*
`MULTILINE`	allows word-wrapping within the checkbox's text area
`GROUP`	define the beginning of the radio button group

Example:

```
ctl[0]: RADIOBUTTON(GROUP), "Remove white", Val=1, Pos=(210,20)
ctl[1]: RADIOBUTTON(MULTILINE, TOP), "Adjust contrast and remove color", Pos=(210,30),
        Size=(70,20)
ctl[2]: GROUPBOX(GROUP, CENTER), "Test", Pos=(200,10), Size=(90,40), Color=CadetBlue
```

Note that the first radio button is activated (value set to one) and declared as the first radio button of the group. The second radio button is moved to the top text line. The last user control, the groupbox, defines the end of the group.

You do not need to visually "embrace" the radio buttons with the groupbox. You can create an invisible groupbox and the group items stay untouched.

LISTBOX

Listboxes are good for scrollable lists. If you need the "pull-down menu"-style, use the COMBOBOX class. The items in the listbox are specified in the text string separated with the escape sequence new line (\n) and each item has its individual value.

You can use these additional properties with the class `LISTBOX` (default values in parentheses):

`Text="Item 1\nItem2"`	defines the listbox's text contents (no Text)
`Val=1`	assigns a value to the listbox and activates the item (-1)

You have the possibility of including these class-specific properties:

`SORT`	sorts the items in alphabetical order – the values of the items are recomputed; top item is always 0 and continues with 1, 2, etc.
`MULTICOLUMN`	items are arranged in columns (also depends on Size-property)
`HSCROLL`	if necessary, a horizontal scrollbar is activated
`VSCROLL`	if necessary, a vertical scrollbar is activated
`DISABLENOSCROLL`	used in conjunction with `HSCROLL` or `VSCROLL`; if the item amount is less than needed to require scrolling, the scrollbar is disabled and not removed
`INTEGRALHEIGHT`	the height of the listbox is resized according to the items' height (default)
`NOINTEGRALHEIGHT`	the height of the listbox is resized according to the Size property, even if items are partially displayed

Example:

```
ctl[0]: LISTBOX, "Multiply\nScreen\nLighten\nDarken"   //try using LISTBOX(SORT)

ctl[1]: LISTBOX(NOINTEGRALHEIGHT), "Uno\nDos\nTres\nCuatro\nCinco", Val=2,
        Size=(40,30)
```

Note:

In the first example, the unsorted and sorted items have the following index (important when calling the listbox's value with the function ctl(0)):

Unsorted (default)		**Alphabetically sorted with class-property SORT**	
Multiply	0	Darken	0
Screen	1	Lighten	1
Lighten	2	Multiply	2
Darken	3	Screen	3

COMBOBOX

Comboboxes are good for drop-down lists, also known as "pull-down menus". The items in the listbox are specified in the text string separated with the escape sequence new line (\n) and each item has its individual value.

You can use these additional properties with the class `COMBOBOX` (default values in parentheses):

`Text="Item 1\nItem2"`	defines the combobox's text contents (no Text)
`Val=1`	assigns a value to the combobox and activates the item (-1)

You have the possibility of including these class-specific properties:

`SORT`	sorts the items in alphabetical order – the values of the items are recomputed; top item is always 0 and continues with 1, 2, etc.
`EXTENDEDUI`	list drops down when right and down arrow keys are pressed (by default, arrow keys select the next item without dropping down the list)
`VSCROLL`	if necessary, a vertical scrollbar is activated
`DISABLENOSCROLL`	used in conjunction with `HSCROLL` or `VSCROLL`; if the item amount is less than needed to require scrolling, the scrollbar is disabled and not removed
`INTEGRALHEIGHT`	the height of the listbox is resized according to the items' height (default)
`NOINTEGRALHEIGHT`	the height of the listbox is resized according to the Size property, even if items are partially displayed
`UPPERCASE/LOWERCASE`	items in the combobox are displayed in uppercase resp. lowercase characters

Example:

```
ctl[0]: COMBOBOX(VSCROLL, DISABLENOSCROLL), "U.S.A.\nGermany\nRussia\nBrazil\nSpain\nEgypt",
        Val=1, Size=(*,60)
```

OWNERDRAW

This user control is a simple rectangle you can colorize. Since it is actionable, you can use it as a simple push button. One possible usage is the definition of an image map consisting of a dialog background image and several `OWNERDRAW` user controls.

You can use this additional property with the class `OWNERDRAW` (default value in parentheses):

`Color=Aquamarine`	defines the background color of the user control (transparent)

Note: *If you wish to use an owner-drawn user control as a settings control (where you can change the user control's value), you have to disable it.*

Examples:

```
ctl[0]: OWNERDRAW, Color=Red, Size=(50,50)

ctl[1]: OWNERDRAW, Action=APPLY

ctl[2]: OWNERDRAW, Disabled, Val=5
```

STATICTEXT

This user control places a text in the dialog window. By default, this user control is not actionable.

You can use these additional properties with the class `STATICTEXT` (default values in parentheses):

`Text="Power Tools"`	defines the text contents (no Text)
`Val=1`	assigns a value to the static text (0)
`Color=Blue`	defines the background color (transparent)
`FontColor=Yellow`	defines the text color (white)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip
`LEFT/CENTER/RIGHT`	aligns the text (LEFT)
`LEFTNOWORDWRAP`	aligns the text to the left and deativates word wrapping

Note: *Once the static text user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value. For example, the action* `APPLY` *returns a value of 2.*

Examples:

```
ctl[0]: STATICTEXT(MODALFRAME, NOTIFY), "OK", Color=Red, FontColor=Yellow, Action=APPLY,
        Size=(60,30)

ctl[3]: STATICTEXT(CENTER), "Edit me"
```

FRAME

This user control class draws a frame in the dialog window. By default, this user control is not actionable.

You can use this additional property with the class `FRAME` (default values in parentheses):

`Val=5`	assigns a value to the frame, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip
`BLACK/GRAY/WHITE`	defines the border's color (BLACK)
`ETCHED`	gives the border a 3D-look
`ETCHEDHORZ/ETCHEDVERT`	makes a single horizontal resp. vertical line in 3D-look

Note: Once the frame user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.

Examples:

```
ctl[0]: FRAME

ctl[1]: FRAME(ETCHED), Val=45, Disabled
```

RECT

This user control class draws a rectangle in the dialog window. By default, this user control is not actiona-ble.

You can use this additional property with the class `RECT` (default values in parentheses):

`Val=5`	assigns a value to the frame, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip
`BLACK/GRAY/WHITE`	defines the rectangle's background color (BLACK)

Note: *Once the rectangle user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.*

Examples:

```
ctl[0]: RECT(NOTIFY), Action=ABOUT
```

```
ctl[1]: RECT(GRAY), Val=231, Disabled
```

BITMAP

The class BITMAP allows one to place a bitmap in the dialog window. By default, this user control is not actionable. You can use this additional property with the class `BITMAP` (default values in parentheses):

`Val=5`	assigns a value to the bitmap, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip
`CENTERIMAGE`	scales the image to original size and centers the image within the control

Note: *Once the bitmap is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.*

Currently, only BMP files are supported. Transparency is supported by using the class IMAGE.

When standalone filters are created, images are not embedded by default. Thus, use the Embed: function to embed the image into the standalone filter file.

The image file should be present in the active directory or in any of the directories set in the PATH *or* FM_PATH *variables (check your AUTOEXEC.BAT file).*

Examples:

```
ctl[0]: BITMAP, Image="Logo.bmp"
```

```
ctl[1]: BITMAP(CENTERIMAGE, NOTIFY), Image="C:\\Programs\\Images\\OK_Button.bmp", Action=APPLY
```

IMAGE

The class IMAGE allows one to place a bitmap in the dialog window. Further, the bitmap will be transparent according to the left- and topmost pixel color. By default, this user control is not actionable.

You can use this additional property with the class `IMAGE` (default values in parentheses):

`Val=5`	assigns a value to the image, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip

Note: *Once the user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.*

Currently, only BMP files are supported. Transparency is supported by using the class IMAGE.

When standalone filters are created, images are not embedded by default. Thus, use the Embed: function to embed the image into the standalone filter file.

The image file should be present in the active directory or in any of the directories set in the PATH *or* FM_PATH *variables (check your AUTOEXEC.BAT file).*

Examples:

```
ctl[0]: IMAGE, Image="Logo.bmp"
```

```
ctl[1]: IMAGE(MODALFRAME), Image="aa.bmp"
```

ICON

The class ICON allows one to place an icon in the dialog window. By default, this user control is not actionable. You can use this additional property with the class `IMAGE` (default values in parentheses):

`Val=5`	assigns a value to the icon, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the user control actionable and activates tooltip
`CENTERIMAGE`	scales the icon to original size and centers it within the control

Note: Once the user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.

When standalone filters are created, images are not embedded by default. Thus, use the Embed: function to embed the image into the standalone filter file.

The image file should be present in the active directory or in any of the directories set in the PATH *or* FM_PATH *variables (check your AUTOEXEC.BAT file).*

Examples:

```
ctl[0]: ICON, Image="Rubberduck.ico"

ctl[1]: IMAGE(MODALFRAME), Image="Bathtub.ico"
```

METAFILE

The class METAFILE allows one to place a metfaile in the dialog window. By default, this user control is not actionable. You can use this additional property with the class `METAFILE` (default values in parentheses):

`Val=5`	assigns a value to the metafile, but only when it is disabled (0)

You have the possibility of including these class-specific properties:

`NOTIFY`	makes the metafile actionable and activates tooltip
`CENTERIMAGE`	scales the icon to original size and centers it within the control

Note: Once the user control is actionable, its value definitions are lost. The reason is that an action returns a specific value and overwrites (once the mouse button is clicked over the user control) the user control's value.

When standalone filters are created, images are not embedded by default. Thus, use the Embed: function to embed the image into the standalone filter file.

METAFILE supports the windows metafile (.WMF) and enhanced metafile (.EMF) formats.

The image file should be present in the active directory or in any of the directories set in the PATH *or* FM_PATH *variables (check your AUTOEXEC.BAT file).*

Examples:

```
ctl[0]: METAFILE, Image="Airplane.wmf"

ctl[1]: METAFILE(MODALFRAME, NOTIFY), Image="D:\\Graphics\\Button2.emf", Action=CANCEL
```

NONE

The class NONE simply deletes the current user control. This is useful for deleting predefined user controls such as the OK, Cancel, Logo or Edit... pushbuttons.

Examples:

```
ctl[CTL_OK]: NONE       //deletes OK pushbutton
ctl[CTL_EDIT]: NONE     //deletes Edit... pushbutton
ctl[CTL_CANCEL]: NONE   //deletes Cancel pushbutton
ctl[CTL_LOGO]: NONE     //deletes FM logo
```

MODIFY

The class MODIFY helps you change certain properties of a user control without touching the other properties. This is useful for modifying predefined user controls such as the OK, Cancel, Logo or Edit... pushbuttons.

Examples:

```
ctl[CTL_OK]: MODIFY, "Apply"                        //changes Text property from OK to Apply
ctl[CTL_EDIT]: MODIFY, Pos=(200,20), Size=(30,30)   //repositions and resizes Edit button
```

3.4 User control run-time functions

This chapter will introduce you to the run-time functions which allow you to change user control properties on a compiled filter. There are various possibilities for using these functions:

- A user control setting is dependable from another user control. For example, the first user control (a scrollbar) creates a border and the second user control (checkbox) tells FM to either create a horizontal or vertical border. The range of the scrollbar is dependable on the width or height of the image (in case the filter is applied to a rectangled image).
- The filter has a Shuffle-pushbutton which changes the settings of the user controls. An advanced Shuffle-method is the usage of a Demo-pushbutton which shuffles the user control settings periodically.
- Loading filter settings from a text or .INI file.

The functions described in this chapter are normally thought to be used within the `ForEveryTile` handler. You may use these functions in the `ForEveryPixel` and `R,G,B,A:` handlers, but wrong implementation of these functions may cause an increase of the filtering time.

createCtl(n, c, t, x, y, w, h, s, sx, p, e)

This function creates a new user control. In conjunction with the `deleteCtl(n)` function, for example, you could activate and deactivate user controls. This function uses the following values:

n	index of the user control; the index ranges from 0 to 49
c	user control's class; you can choose from the following classes: `CC_STANDARD, CC_SCROLLBAR, CC_TRACKBAR, CC_CHECKBOX, CC_PUSHBUTTON, CC_GROUPBOX, CC_RADIOBUTTON, CC_LISTBOX, CC_COMBOBOX, CC_OWNERDRAW, CC_STATICTEXT, CC_FRAME, CC_RECT, CC_BITMAP, CC_IMAGE, CC_ICON`
t	user control text string (if class permits it)
x, y	position of the user control (using -1 makes FM use default values)
w, h	width and height of the user control (using -1 makes FM use default values)
s	style of the user control's class (differs from user control to user control):

CC_CHECKBOX	use s=3
CC_GROUPBOX	use s=7
CC_RADIOBUTTON	use s=4
CC_COMBOBOX	use s=2
CC_OWNERDRAW	use s=11
CC_FRAME	use s=7
CC_RECT	use s=4, 5 or 6 for black, white or grey fill
rest of the user controls	use s=0

sx	extended style of the user control's class (possibly differs from user control to user control):
	-2 sunken 3D-look
	-1 3D border
	0 normal (default)
	1 MODALFRAME-like look
p	user control's property; currently, no text string is processed here – set this value to 0 and use the other user control functions described later in this chapter
e	enable level:
	-1 enabled (default)
	0 disabled and invisible
	1 disabled and visible

Examples:

```
createCtl(0, CC_PUSHBUTTON, "Oh boy", 200, 20, 40, 30, 0, 1, 0, -1);
createCtl(4, CC_CHECKBOX, "Disrupt color", -1, -1, -1, -1, 3, 0, 0, -1);
```

setCtlColor(n, color)

This function changes the background color of the user control with the index n. The color value can be an RGB triplet (as in RGB(r,g,b)) or a COLOR function (as in `COLOR(MediumOrchid)`). The following user controls support background coloring: standard scrollbars, trackbars, checkboxes, radio buttons, group boxes, owner draw controls, static text and icons. To set the background color to transparent, set the value color to -1.

Examples:

```
setCtlColor(0, RGB(255,0,255) );        //set background color to magenta
setCtlColor(6, COLOR(X11.indianred));   //set background color to magenta
setCtlColor(23, -1);                    //set background color to transparent
```

setCtlText(n, text)

This function changes the text property of the user control with the index n. Note that not all user controls support text strings. Also keep in mind that the user control has to be resized if your text string exceeds the current size. Substrings and escape sequences are allowed.

Examples:

```
setCtlText(1, "Rewrite me");
setCtlText(CTL_OK, "Apply");            //rename OK pushbutton text to "Apply"
```

setCtlFontColor(n, color)

This function changes the text color of the user control with the index n. The color value can be an RGB triplet (as in `RGB(r,g,b)`) or a COLOR function (as in `COLOR(DarkGreenCopper)`). The following user controls support text coloring: standard scrollbars, checkboxes, radio buttons, group boxes and static text. To set the default text color (white), set the value color to -1.

Examples:

```
setCtlFontColor(0, RGB(255,255,0) );     //set text color to yellow
setCtlFontColor(0, COLOR(MoneyGreen) ); //set text color to "Money Green"
setCtlFontColor(0, -1);                 //set text color to default (white)
```

setCtlPos(n, x, y, w, h)

This function repositions and resizes the user control with the index n. The values x,y represent the horizontal and vertical position and w, h the width and height of the user control. Set the value to -1 to use the default or last defined measurement. Don't forget that all measurements are set in DBUs (dialog box units).

Examples:

```
setCtlPos(4, 230, 20, 40, 30);
setCtlPos(5, -1, -1, 50, -1);          //change only the width of user control 5
```

setCtlVal(n, v)

This function changes the current value of the user control with the index n. Note that the value set has to be one within the user control's range. Also note that not all user controls support value settings – for example, you can only save values in a `FRAME` user control when it is disabled.

Examples:

```
setCtlVal(4, 30);
setCtlPos(5, 0);
```

ctl(n)

This function retrieves the current value of the user control with the index n. Be aware of the range the user control can return. For example, a push button can only return the values 0 and 1.

Example:

```
R,G,B: c+ctl(0)           //adds the current user control value to the current pixel color
```

setCtlRange(n, lo, hi)

This function changes the range of the user control with the index n. Consider the possible range of certain user controls. For example, you cannot enlarge the range of a checkbox.

Examples:

```
setCtlRange(0, -128, 128);
setCtlRange(23, 5, -5);
```

setCtlLineSize(n, a)

This function changes the line jump unit (i.e., when clicking on the buddy controls) of the user control with the index n. Applies only to the user controls `STANDARD` and `SCROLLBAR`.

Examples:

```
setCtlLineSize(0, 12);
setCtlLineSize(5, 2);
```

setCtlPageSize(n, a)

This function changes the page jump unit (i.e., when clicking between buddies and handle) of the user control with the index n. Applies only to the user controls `STANDARD, SCROLLBAR` and `TRACKBAR`.

Examples:

```
setCtlLineSize(0, 12);
setCtlLineSize(5, 2);
```

enableCtl(n, level)

This function sets the state of the user control with the index n. The following levels can be set:

-1	visible and enabled user control
0	invisible and disabled user control
1	visible and disabled user control

Examples:

```
enableCtl(0, 1);                        //disables a visible user control
enableCtl(5, 0);                        //makes user control 5 invisible and disabled
```

setCtlToolTip(n, "Text", s)

This function changes the tool tip text and its state s of the user control with the index n. The following states can be set:

-1	disable tool tip
0	show tool tip under cursor
2	show tool tip under user control (centered)

Examples:

```
setCtlToolTip(0, "Follow me!!", 0);
setCtlToolTip(5, "Center me!!", 2);
```

deleteCtl(n)

This function simply deletes the user control with the index n.

Examples:

```
deleteCtl(5);
deleteCtl(CTL_CANCEL);
```

setCtlAction(n, a)

This function changes or sets the action of the user control with the index n. Note that some user controls need to have a `NOTIFY` class-specific property and that the `GROUPBOX` class is not actionable. The following actions are available:

`CA_APPLY`	Apply the filter to the image
`CA_NONE`	No action
`CA_PREVIEW`	Update the preview window
`CA_CANCEL`	Cancel filter prosecution
`CA_EDIT`	Open the editing window (only in programming mode, ignored in stand-alone filters)
`CA_ABOUT`	Open the about window

Examples:

```
setCtlAction(5, CA_APPLY);
setCtlAction(CTL_CANCEL, CA_APPLY);
```

Tutorial 3.4

a) Simple animation

This tutorial will teach you the usage of some run-time user control functions. Let's try to understand the following FM program:

```
// Listing 3-4.a
// When clicking on the push button, it will update the scroll bar's value and
// move the image from left-to-right and from right-to-left

%ffp

ctl[0]: PUSHBUTTON, "Start animation", Size=(60,*)
ctl[1]: "Yoyo", Range=(0,150), Pos=(*,24), Track
ctl[2]: BITMAP, Image = "C:\\Graphics\\Files\\Wizard.bmp", Pos=(180,50)


ForEveryTile:
{

setCtlPos(2,180+ctl(1),-1,-1,-1);               //at any time, reposition logo according to
                                                //the scroll bar

if (ctl(0))                                     //if user clicks the push button...
        {
        for (x=0; x<151; x++)                   //loop 1
                {
                setCtlVal(1,x);
                setCtlPos(2, 180+ctl(1), -1, -1, -1);
                setCtlFontColor(1, RGB(rnd(0,255), rnd(0,255), rnd(0,255)) );
                sleep(10);
                }

        for (x=149; x>=0; x--)                  //loop 2
                {
                setCtlVal(1,x);
                setCtlPos(2, 180+ctl(1), -1, -1, -1);
                setCtlFontColor(1, RGB(rnd(0,255), rnd(0,255), rnd(0,255)) );
                sleep(10);
                }

        setCtlVal(0,0);                         //reset push button
        }

return false;                                   //do not apply filter
}
```

The first three lines define the user controls. The push button is resized so the text can fit in the button's area. The scrollbar is repositioned so it won't conflict with the area of the push button. It will also immediately pass its current value to the filter while the user control's handle is held by the user. The third user control is an image which is repositioned.

In the `ForEveryTiler` handler, the main function in this program, two tasks are implemented.

The first task consists of one line only. It will reposition the image (user control 2) along the x-axis leaving all other properties (y-position, width and height) as-is.

The second task is only activated when the push button is depressed. When the user clicks on the push button, two loops are run. These loops consist of four commands:

– give the scroll bar a new value
– reposition the image along the x-axis
– gives the font of the scroll bar a random new color
– pauses for 10 millseconds

The effect you will see is the animation of the scroll bar and image from left-to-right (loop 1) and right-to-left (loop 2). During the animation the font color of the scroll bar changes.

The last command in the if-block resets the push button back to its default value.

The last command in the `ForEveryTile` handler tells FM to return the boolean value false. The reason for this is that we are not applying any pixel filters and we are giving control over to the `ForEveryPixel` handler. Its default action is to leave the image untouched.

b) Saving/Loading user control settings

The following tutorial shows you the possibility of saving and loading user control settings. By adding this feature to your program, the user can save his preferred user control settings. We also want to point out the possibility of the implementation of "memory dots"based on this tutorial's technique. For example, four radio buttons could save different user control settings. But let's start with the basics.

Compile and play with this simple filter:

```
//Listing 3-4.b
//Saving/Loading user control settings

%ffp
```

```
ctl[0]: "x", val=20
ctl[1]: "y", val=10
ctl[2]: "m", val=100

R,G,B:  scl(sin(x*ctl(0))*cos(y*ctl(1))/m*ctl(2),-512*512,512*512,0,255)
A:      a
```

This quick and dirty filter takes the current pixel's coordinate at (x,y), its position from the image's center (m) and computes a black and white grid with sharp focus on the image's center. The transparency information is left unchanged. The user controls are set to a value different than zero to give the user a first idea of this filter (if all settings were set to 0, the user would see only a gray preview window).

We need two push buttons which represent the functions of saving and loading user control data:

```
ctl[3]: PUSHBUTTON, "Save", Pos=(270,50)
ctl[4]: PUSHBUTTON, "Load", Pos=(320,50)
```

The ForEveryTile handler is used to check the push buttons' states and execute the save and load routines.

```
ForEveryTile:
{
if (ctl(3))                     //Did the user click on Save?
        {
        setCtlVal(3,0);         //Reset Save push button
        }

if (ctl(4))                     //Did the user click on Load?
        {
        setCtlVal(4,0);         //Reset Load push button
        }
return false;                   //Leave image as-is and let the RGBA-handler do the filtering
}
```

File functions such as fopen and fclose return integer values, if data was successfully transferred to or from disk media. For example, the function `fopen("Settings.ini", "w")` creates a ASCII text file named `Settings.ini` in the current directory and prepares it for **w**riting purposes. At the same time, it returns an integer value. After executing the function, FM has to know whether the file creation was successful or not. The following term:

```
SETTINGS_FILE=fopen("Settings.ini", "w");
```

creates the file and returns its status to the variable `SETTINGS_FILE`. If the variable is any number other than zero (which also means that this number is used as an index for the file), file creation was successful and we can continue to start writing to the file (with the help of the index saved in the variable `SETTINGS_FILE`). In an FM program, the routine would look like the following:

```
SETTINGS_FILE=fopen("Brightness_Contrast.ini", "w");

if (SETTINGS_FILE)      //if variable is any number other than zero,
//Write to the file
else
//Error occured;
```

One possible error that can occur is that the media you are saving to is write-protected. The same technique is used to close the file. With the function fclose(SETTINGS_FILE), the file with the index SETTINGS_FILE is closed and the function returns a number – any number other than zero means that the file could not be closed. Normally, you will close the file right after writing all the necessary data to it. A more complete routine could look like the following:

```
if ( SETTINGS_FILE=fopen("Settings.ini", "w") )
{
        for (i=0; i<3; i++)                                 //Read user control values
                fprintf(SETTINGS_FILE, "%d\n", ctl(i) );    //Save data to the file
        if ( fclose(SETTINGS_FILE) )                        //Close file
                Error("Can't close settings file.");
}
else
Error("Can't write to settings file.");             //If SETTINGS_FILE is zero: error!
```

The above routine opens a file we want to write to. Note that we combined the two first two lines into one. If the file is ready to receive some data, a loop writes the user control settings in the file with the function fprintf(). Note that %d is replaced by the value set in ctl(i) and the escape sequence new line (\n) is inserted, as well. If you open the file with a text editor, you will see three number values in three consecutive lines.

After writing all the data, the file is closed and tested if closing was successful. Finally, two error message window functions are implemented if any error happens.

It is always crucial to inform the user when something does not work. The worst thing that can happen is that a user cannot give you any feedback when a problem occurs, so be sure to intercept all possible errors in your FM program.

Opening a file works the same way. Take a look at the complete routine for opening a file and loading the settings back to the filter:

```
if (SETTINGS_FILE=fopen("Settings.ini", "r"))
{
        for (i=0; i<3; i++)
        {
                fscanf(SETTINGS_FILE, "%d", &iVal);     //Read data from file
                setCtlVal(i,iVal);                      //and assign it to user controls
        }
        if (fclose(SETTINGS_FILE))
                Error("Can't close settings file.");
}
else
Error("Can't open settings file.");
```

This time, the file is opened for reading purposes and checked if opening the file was successful. If this is the case, a loop reads the settings of the user controls with the function fscanf() and saves it into the variable iVal which is afterwards used to set the user control value. The file is closed and tested if it was successful. Again, two error message window functions are implemented in the case when some error happens (e.g., corrupt or missing file).

If you tried to assemble your FM program, you surely noticed that the compiler did not recognize the variables SETTINGS_FILE, iVal or i. These have to be declared in the `ForEveryPixel` handler, as well. The complete FM program looks like this:

```
// Listing 3-4.b
// Saving/Loading user control settings

%ffp

ctl[0]: "x", val=20
ctl[1]: "y", val=10
ctl[2]: "m", val=100

ctl[3]: PUSHBUTTON, "Save", Pos=(270,50)
ctl[4]: PUSHBUTTON, "Load", Pos=(320,50)


ForEveryTile:
{

int SETTINGS_FILE, i;   //Define file and counter variables

if (ctl(3))             //Did the user click on Save?
{
        if ( SETTINGS_FILE=fopen("Settings.ini", "w") )
        {
                for (i=0; i<3; i++) //Read user control values
                        fprintf(SETTINGS_FILE, "%d\n",ctl(i)); //Save data to the file
                if (fclose(SETTINGS_FILE))
                Error("Can't close settings file.");
        }
        else
        Error("Can't write to settings file.");
        setCtlVal(3,0); //Reset pushbutton
}

if (ctl(4))             //Did the user click on Load?
{
        int iVal;       //Variable where user control data is saved temporarily
        if (SETTINGS_FILE=fopen("Settings.ini", "r"))
        {
                for (i=0; i<3; i++)
                {
                        fscanf(SETTINGS_FILE, "%d", &iVal);  //Read data from file
                        setCtlVal(i,iVal);                   //and assign it to user controls
                }
                if (fclose(SETTINGS_FILE))
                Error("Can't close settings file.");
        }
        else
        Error("Can't open settings file.");
        setCtlVal(4,0);                                         //Reset Pushbutton
}

return false;           //Leave image unchanged, let the RGBA handler do the filtering
}                       //end ForEveryTile


R,G,B:  scl(sin(x*ctl(0))*cos(y*ctl(1))/m*ctl(2),-512*512,512*512,0,255)
A:      a
```

As stated in the beginning, you can implement the so-called memory dots, possibly represented by radio buttons which can save and load different user control settings. That is a homework for you, though...

Note: Future FM versions will be able to save and load filter data to and from the Windows' Registry.

4. File Input/Output

This chapter will describe the navigation in the editing window, how to save filter programs in source code and load various filter file formats and finally how to create stand-alone filters.

4.1 The editing window

When you call FilterMeister from the Menu Filter... you are first presented with a dialog window. Pressing on the **`Edit >>>`** pushbutton reveals the editing window plus a couple of pushbuttons. The editing window is where you type in your filter program. This readable program listing is also called "source code".

The FM editing window resembles the MS-Windows program Notepad in its basic functions. To start typing in the editing window, click with the mouse in the editing area and a cursor will appear. While you type, you can start a new line by pressing on the `ENTER`-key on your keyboard. If you want to indent a line, you can insert either spaces or tab spaces. Note that tab spaces are set with the key combination `CTRL-TAB` or `CTRL-I`.

You can select portions of the text by clicking and dragging with the mouse. If you press the *right* mouse button, a context menu appears containing the following commands (you can use the keyboard shortcuts listed next to the commands):

`Undo`	`CTRL-Z`
`Cut`	`CTRL-X`
`Copy`	`CTRL-C`
`Paste`	`CTRL-V`
`Select all`	*`no keyboard shortcut available`*

There is currently no drag-and-drop feature for text. After entering the program listing, you can click on the Compile button to compile your filter program. You will encounter an error dialog window if FM's parser finds errors of any kind (e.g., syntax errors, type mismatch, missing arguments, etc.). If you press on OK, the current filter program is applied to your image. If you edit the program listing and forget to click on the Compile button, FM will tell you that the compiled code is not up-to-date and ask you if you want to compile it before filter application.

Some host programs "forget" the editing window's contents, so be sure to save your program on disk. Should you forget to save the program and click on the OK button to apply the filter code, FM will remind you that the source code has been modified and ask you if you want to save it first before applying it on the image. If you decide not to save, the filter is nevertheless applied.

4.2 Saving/Loading filter source codes

During filter creation, unprecendented things like a hook-up can destroy the investment of hours. We advise you to save your code regularly by clicking on the `Save...` button. The filter source code is saved in the Filter Factory Plus source code file format (`.ffp`), a simple ASCII file.

You have the possibility of loading the following file formats:

```
Filter Factory Plus source code files      .ffp
Filter Factory source files                .8bf, .afs
Filter Factory Wizard source files         .ffw
Text files                                 .txt
```

Please note that only `.8bf` standalone filters created with Filter Factory can be imported into FM. If you import an `.8bf`-file, the fields `author, copyright, category, filter title, filename` and sliders with their corresponding preset values are correctly interpreted by FM and adds the fields `description` and `version`. Importing an `.afs` file will only interpret the filename and controls preset values. The rest of the fields are set to default values.

The Filter Factory Wizard format is a modification and extension of the `.afs`- files. This format adds the fields `category, title, copyright, author` and `filename` and specifies which sliders with their corresponding preset values are used.

4.3 Creating standalone filters

When you are completely satisfied with a filter you created, you can create a stand-alone filter by clicking the `Make...` button. This will pop-up a Save dialog window where you can change the filename and place in your disk.

If you did not specify the fields `filename, category` and `title`, FilterMeister will create the standalone filter with the following field contents:

```
filename    MyFilter.8bf
category    FilterMeister
title       My Filter
```

Note that you usually have to restart the host program so it can recognize the new filter. Needless to say: be sure to save the standalone filter in a plug-in directory which is recognized by the host program.

The field filename can include an absolute filename hierarchy. Be sure to use two backslashes as divisors:

```
filename: "C:\\Programs\\Photoshop\\Plug-Ins\\Starfield.8bf"
```

6. Appendix

A. Color Name and Value Specifications

In the FF+ language, a particular color value can be designated in one of several ways: as a decimal or hexdecimal RGB color triple value, as a CMYK or HSB numeric value, or as a "standard" color name.

There are several formats for specifying a numeric color value. Each is described in this Appendix.

There are also several name spaces from which the user may designate a color by name. The default color name space used by FF+ is the list of X11 color names which are now part of the international HTML JavaScript standards. Other name spaces available in FF+ are: An older proposed list of color names for HTML, called herein the "HTML" name space; the Java color name space, herein called "JAVA"; the Netscape "safe" color space, herein called "NS"; the Microsoft Windows 95 color name space, herein called "Win"; and the Win32 system UI color name space, herein called "UI".

To avoid naming conflicts in the various name spaces, preface the color name with a name space designator followed by a period ("."). For example, the X11 value of "green" is designated as 'X11.green' and the Netscape "safe" green is designated as 'NS.green'. There is one exception: The names in the UI name space do not conflict with any other name space, so the "UI." prefix is never required. If the color name is not qualified by a color name space designator, the name will be resolved by searching the FF+ color name spaces in the following order: X11, HTML, JAVA, NS, Win, and UI.

A.1 Numeric Color Value Specifications

Internally, FM always represents an RGB color value as a 32-bit integer, with the red component in the least signficant (lowest-addressed) byte, the green and blue components in the next two (higher-addressed) bytes, and the most significant (highest-addressed) byte set to 0.

In an FF+ program, an RGB color value may be specified numerically in one of three ways:

- by specifying the red, green, and blue values as the corresponding arguments to the "RGB" built-in function. For example, the color *bronze*, with a red value of 140, a green value of 120, and a blue value of 83, may be specified as `RGB(140,120,83)` or equivalently as `RGB(0x8C,0x78,0x53)` in hexadecimal notation.

- As an HTML-style 6-digit hexadecimal number, with the red component in the first two digits, the green component in the next two digits,and the blue component in the last two digits (schematically: #rrggbb). For example, the value of "bronze" might be specified as #8C7853.

- As an FF+ decimal, hexadecimal, or octal integer value. In this case, be aware that FM is running on a little-endian (i.e., x86) architecture machine, so the red component contributes to the *least* significant bits of the value, while the blue component contributes to the *most* signficant bits. For example, the value of "bronze" (#8C7853) might be represented as the hexadecimal integer 0x53788C, as the octal integer 024674214, or as the decimal integer 5470348. Note that the order of the components is reversed between the x86 hexadecimal integer representation (0x53788C) and the HTML hexadecimal representation (#8C7853), since HTM hex is based on a big-endian representation, while the x86 architecture is little-endian. Caution: If FM is ported to a big-endian environment, such as a 68K or PowerPC-based Macintosh, then you must reverse the bytes in the native integer representation of an RGB color value. It's more portable to use one of the previous two representations (RGB(140,120,83) or #8C7853) instead of a native integer representation when specifying RGB color values!

A.2 X11 Color Names

The color names in the X11 color name space are recognized by JavaScript 1.2, Netscape Navigator 3.0+, and Internet Explorer 3.0+.

The X11 color names and corresponding RGB triple values (in hexadecimal) are listed in *Table A-2*. To specify one of these names explicitly, prefix the name with 'X11.'; for example, 'X11.green' always specifies RGB color value #008000. The names in the X11 name space are case-insensitive.

Table A-2. The X11 Color Names

Color name	#rrggbb value	Color name	#rrggbb value
aliceblue	#F0F8FF	burlywood	#DEB887
antiquewhite	#FAEBD7	cadetblue	#5F9EA0
aqua	#00FFFF	chartreuse	#7FFF00
aquamarine	#7FFFD4	chocolate	#D2691E
azure	#F0FFFF	coral	#FF7F50
beige	#F5F5DC	cornflowerblue	#6495ED
bisque	#FFE4C4	cornsilk	#FFF8DC
black	#000000	crimson	#DC143C
blanchedalmond	#FFEBCD	cyan	#00FFFF
blue	#0000FF	darkblue	#00008B
blueviolet	#8A2BE2	darkcyan	#008B8B
brown	#A52A2A	darkgoldenrod	#B8860B

Table A-2. The X11 Color Names (cont.)

Color name	#rrggbb value	Color name	#rrggbb value
darkgray	#A9A9A9	lightyellow	#FFFFE0
darkgreen	#006400	lime	#00FF00
darkkhaki	BDB76B	limegreen	#32CD32
darkmagenta	#8B008B	linen	#FAF0E6
darkolivegreen	#556B2F	magenta	#FF00FF
darkorange	#FF8C00	maroon	#800000
darkorchid	#9932CC	mediumaquamarine	#66CDAA
darkred	#8B0000	mediumblue	#0000CD
darksalmon	#E9967A	mediumorchid	#BA55D3
darkseagreen	#8FBC8F	mediumpurple	#9370DB
darkslateblue	#483D8B	mediumseagreen	#3CB371
darkslategray	#2F4F4F	mediumslateblue	#7B68EE
darkturquoise	#00CED1	mediumspringgreen	#00FA9A
darkviolet	#9400D3	mediumturquoise	#48D1CC
deeppink	#FF1493	mediumvioletred	#C71585
deepskyblue	#00BFFF	midnightblue	#191970
dimgray	#696969	mintcream	#F5FFFA
dodgerblue	#1E90FF	mistyrose	#FFE4E1
firebrick	#B22222	moccasin	#FFE4B5
floralwhite	#FFFAF0	navajowhite	#FFDEAD
forestgreen	#228B22	navy	#000080
fuchsia	#FF00FF	oldlace	#FDF5E6
gainsboro	#DCDCDC	olive	#808000
ghostwhite	#F8F8FF	olivedrab	#6B8E23
gold	#FFD700	orange	#FFA500
goldenrod	#DAA520	orangered	#FF4500
gray	#808080	orchid	#DA70D6
green	#008000	palegoldenrod	#EEE8AA
greenyellow	#ADFF2F	palegreen	#98FB98
honeydew	#F0FFF0	paleturquoise	#AFEEEE
hotpink	#FF69B4	palevioletred	#DB7093
indianred	#CD5C5C	papayawhip	#FFEFD5
indigo	#4B0082	peachpuff	#FFDAB9
ivory	#FFFFF0	peru	#CD853F
khaki	#F0E68C	pink	#FFC0CB
lavender	#E6E6FA	plum	#DDA0DD
lavenderblush	#FFF0F5	powderblue	#B0E0E6
lawngreen	#7CFC00	purple	#800080
lemonchiffon	#FFFACD	red	#FF0000
lightblue	#ADD8E6	rosybrown	#BC8F8F
lightcoral	#F08080	royalblue	#4169E1
lightcyan	#E0FFFF	saddlebrown	#8B4513
lightgoldenrodyellow	#FAFAD2	salmon	#FA8072
lightgreen	#90EE90	sandybrown	#F4A460
lightgrey	#D3D3D3	seagreen	#2E8B57
lightpink	#FFB6C1	seashell	#FFF5EE
lightsalmon	#FFA07A	sienna	#A0522D
lightseagreen	#20B2AA	silver	#C0C0C0
lightskyblue	#87CEFA	skyblue	#87CEEB
lightslategray	#778899	slateblue	#6A5ACD
lightsteelblue	#B0C4DE	slategray	#708090

Table A-2. The X11 Color Names (cont.)

Color name	#rrggbb value	Color name	#rrggbb value
snow	#FFFAFA	turquoise	#40E0D0
springgreen	#00FF7F	violet	#EE82EE
steelblue	#4682B4	wheat	#F5DEB3
tan	#D2B48C	white	#FFFFFF
teal	#008080	whitesmoke	#F5F5F5
thistle	#D8BFD8	yellow	#FFFF00
tomato	#FF6347	yellowgreen	#9ACD32

A.3 Older Proposed HTML Color Names

The following color names and values are taken from p. 148 of "The Web Programming Desktop Reference." Apparently these colors were part of an earlier HMTL color standard proposal that was rejected in favor of the X11 colors.

Note that there are several conflicts between these color names and the X11 colornames (e.g., X11 "green" is #008000, while the "green" in this proposal is #00FF00), so these colors cannot share the same namespace with the X11 colors. However, I find many of the colors below to be more apt than the X11 colors (example: "orange" here looks more like orange to me than X11 "orange") and they seem to cover a more useful range of colors than X11, so I have added these colors to the set of names recognized by FF+.

These older color names and corresponding RGB triple values (in hexadecimal) are listed in *Table A-3*. To specify one of these names explicitly, prefix the name with 'HTML.'; for example, 'HTML.green' always specifies RGB color value #00FF00. The names in this name space are case-insensitive.

Table A-3: Older Proposed HTML Color Names

Color name	#rrggbb value	Color name	#rrggbb value
Aquamarine[2]	#70DB93	CornFlowerBlue[2]	#42426F
BakersChocolate	#5C3317	Cyan[1,3]	#00FFFF
BlueViolet[2]	#9F5F9F	DarkBrown	#5C4033
Black[1,3]	#000000	DarkGreen[2]	#2F4F2F
Blue[1,3]	#0000FF	DarkGreenCopper	#4A766E
Brass	#B5A642	DarkOliveGreen[2]	#4F4F2F
BrightGold	#D9D919	DarkOrchid[2]	#9932CD
Brown[2]	#A62A2A	DarkPurple	#871F78
Bronze	#8C7853	DarkSlateBlue[2]	#6B238E
BronzeII	#A67D3D	DarkSlateGray[1]	#2F4F4F
CadetBlue[2]	#5F9F9F	DarkTan	#97694F
CoolCopper	#D98719	DarkTurquoise[2]	#7093DB
Copper	#B87333	DarkWood	#855E42
Coral[2]	#FF7F00	DimGray[2]	#545454

Table A-3: Older Proposed HTML Color Names (cont.)

Color name	#rrggbb value	Color name	#rrggbb value
DustyRose	#856363	NewMidnightBlue	#00009C
Feldspar	#D19275	NewTan	#EBC79E
Firebrick[2]	#8E2323	OldGold	#CFB53B
ForestGreen[2]	#238E23	Orange[2]	#FF7F00
Gold[2]	#CD7F32	OrangeRed[2]	#FF2400
Goldenrod[2]	#DBDB70	Orchid[2]	#DB70DB
Gray	#C0C0C0	PaleGreen[2]	#8FBC8F
Green[2] [3]	#00FF00	Pink[2]	#BC8F8F
GreenCopper	#527F76	Plum[2]	#EAADEA
GreenYellow[2]	#93DB70	Quartz	#D9D9F3
HunterGreen	#215E21	Red[1] [3]	#FF0000
IndianRed	#4E2F2F	RichBlue	#5959AB
Khaki[2]	#9F9F5F	Salmon[2]	#6F4242
LightBlue[2]	#C0D9D9	Scarlet	#8C1717
LightGray	#A8A8A8	SeaGreen[2]	#238E68
LightSteelBlue[2]	#8F8FBD	SemiSweetChocolate	#6B4226
LightWood	#E9C2A6	Sienna[2]	#8E6B23
LimeGreen[1]	#32CD32	Silver[2]	#E6E8FA
Magenta[1] [3]	#FF00FF	SkyBlue[2]	#3299CC
MandarinOrange	#E47833	SlateBlue[2]	#007FFF
Maroon[2]	#8E236B	SpicyPink	#FF1CAE
MediumAquamarine[2]	#32CD99	SpringGreen[1]	#00FF7F
MediumBlue[2]	#3232CD	SteelBlue[2]	#236B8E
MediumForestGreen	#6B8E23	SummerSky	#38B0DE
MediumGoldenrod	#EAEAAE	Tan[2]	#DB9370
MediumOrchid[2]	#9370DB	Thistle[1]	#D8BFD8
MediumSeaGreen[2]	#426F42	Turquoise[2]	#ADEAEA
MediumSlateBlue[2]	#7F00FF	VeryDarkBrown	#5C4033
MediumSpringGreen[2]	#7FFF00	VeryLightGray	#CDCDCD
MediumTurquoise[2]	#70DBDB	Violet[2]	#4F2F4F
MediumVioletRed[2]	#DB7093	VioletRed	#CC3299
MediumWood	#A68064	Wheat[2]	#D8D8BF
MidnightBlue[2]	#2F2F4F	White[1] [3]	#FFFFFF
NavyBlue	#23238E	Yellow[3]	#FFFF00
NeonBlue	#4D4DFF	YellowGreen[2]	#99CC32
NeonPink	#FF6EC7		

Legend:
[1] *same as X11*
[2] *conflict with X11*
[3] *Netscape "safe color"*

A.4 The Java Color Name Space

The color names in the JAVA color name space are recognized by the Java JDK 1.0 and later (specifically, they are defined in the java.awt.Color class).

The JAVA color names are listed in *Table A-4.* To specify one of these names explicitly, prefix the name with 'JAVA.'; for example, 'JAVA.green' always specifies the RGB color value defined as 'green' by the java.awt package. The names in the JAVA name space are case-insensitive.

Table A-4. The Java Color Names

Color name	Description	Color name	Description
black	The color black.	magenta	The color magenta.
blue	The color blue.	orange	The color orange.
cyan	The color cyan.	pink	The color pink.
darkGray	The color dark gray.	red	The color red.
gray	The color gray.	white	The color white.
green	The color green.	yellow	The color yellow.
lightGray	The color light gray.		

A.5 Netscape Safe Colors

On a 256-color display, Netscape reserves 216 entries in the color palette for the equally-spaced nodes of a 6x6x6 color cube. These 216 colors will not be dithered, so are considered "safe" to use in Web pages that will be viewed on 256-color systems. The 216 colors have the form #rrggbb where rr, gg, and bb are each one of 00, 33, 66, 99, CC, or FF.

Some of these 216 colors have commonly-used names (e.g., "Netscape Gray" is #CCCCCC) and are listed in the *Table A-5* below. I also propose a few other color names for Netscape colors. If anyone wants to assign names to the remaining Nestcape "safe" colors, please be my guest.

To specify one of the Netscape "safe" color names unambiguously, prefix the name with 'NS.'; for example, 'NS.green' always specifies RGB color value #00CC00. The names in the NS name space are case-insensitive.

Table A-5: Netscape ”Safe” Colors

Color name	#rrggbb value	Color name	#rrggbb value
Black[1]	#000000	MediumBlue[2] [3]	#000099
Blue[2] [3]	#0000CC	MediumCyan	#009999
BrightBlue	#0000FF	MediumGray	#999999
BrightCyan	#00FFFF	MediumGreen	#009900
BrightGreen	#00FF00	MediumMagenta	#990099
BrightMagenta	#FF00FF	MediumRed	#990000
BrightRed	#FF0000	NetscapeGray	#CCCCCC
Cyan	#00CCCC	Red[2] [3]	#CC0000
DarkBlue[2]	#000066	VeryDarkBlue	#000033
DarkCyan[2]	#006666	VeryDarkCyan	#003333
DarkGray[2]	#666666	VeryDarkGray	#333333
DarkGreen[2] [3]	#006600	VeryDarkGreen	#003300
DarkMagenta[2]	#660066	VeryDarkMagenta	#330033
DarkRed[2]	#660000	VeryDarkRed	#330000
Green[2] [3]	#00CC00	White[1]	#FFFFFF
Magenta[2] [3]	#CC00CC	Yellow[1]	#FFFF00

Legend:
[1] *same as X11*
[2] *conflict with X11*
[3] *conflict with HTML*

A.6 The 20 Static Win95 Colors

Table A-6 lists the 20 static colors that are always present in the Win95 SVGA palette (the standard VGA palette has only 16 colors and does not include Cream, MediumGray, MoneyGreen, and SkyBlue). Again, there are some naming conflicts with the previous lists of colors. To specify one of the Win95 SVGA color names unambiguously, prefix the name with 'Win.'; for example, 'Win.green' always specifies RGB color value #00FF00. The names in the 'Win' color name space are case-insensitive.

Table A-6: The 20 Static Win95 Colors

Color name	#rrggbb value	Color name	#rrggbb value
Black[1] *	#000000	Cyan[1] *	#00FFFF
Blue[1] *	#0000FF	DarkBlue[2]	#000080
Cream	#FFFBF0	DarkCyan[2]	#008080

Table A-6: The 20 Static Win95 Colors (cont.)

Color name	#rrggbb value	Color name	#rrggbb value
DarkGray[2]	#808080	Magenta[1] *	#FF00FF
DarkGreen[2] [3]	#008000	MediumGray	#A0A0A4
DarkMagenta[2]	#800080	MoneyGreen	#C0DCC0
DarkRed[2]	#800000	Red[1] *	#FF0000
DarkYellow	#808000	SkyBlue[2] [3]	#A6CAF0
Green[2] *	#00FF00	White[1] *	#FFFFFF
LightGray[3]	#C0C0C0	Yellow[1] *	#FFFF00

Legend:

[1] *same as X11*
[2] *conflict with X11*
[3] *conflict with HTML*
* *Netscape "safe" color*

A.7 Windows UI System Colors

Windows defines 25 system colors for painting various parts of the User Interface. Most (but not all) of these system colors can be changed by the end user with the Display applet in the Windows Control Panel.

The filter designer can use these color names to match portions of the filter dialog color scheme to the color scheme on the end user's system. For example, when generating tool tips in text control n, the filter designer can code: `setCtlColor(n, COLOR_INFOBK)`,`setCtlFontColor(n, COLOR_INFOTEXT)` in order to match the user/system settings for tool tip background and text colors, respectively.

Table A-7 lists the names (and some synonyms) by which an FF+ program can refer to these system colors, along with a brief description of how each color is used. RGB triple values are not listed for these colors, since they are system- and/or user-defined. There are no naming conflicts between these color names and the other color name spaces supported by FF+, so it is never necessary to prefix these names; however, the prefix 'UI.' is allowed for consistency with the other name spaces.

Table A-7: Windows UI System Colors

Color name	Description
COLOR_3DDKSHADOW	Dark shadow for three-dimensional display elements.
COLOR_3DFACE, COLOR_BTNFACE	Face color for three-dimensional display elements.

Table A-7: Windows UI System Colors (cont.)

Color name	Description
COLOR_3DHILIGHT, COLOR_3DHIGHLIGHT, COLOR_BTNHILIGHT, COLOR_BTNHIGHLIGHT	Highlight color for three-dimensional display elements (for edges facing the light source.)
COLOR_3DLIGHT	Light color for three-dimensional display elements (for edges facing the light source.)
COLOR_3DSHADOW, COLOR_BTNSHADOW	Shadow color for three-dimensional display elements (for edges facing away from the light source).
COLOR_ACTIVEBORDER	Active window border.
COLOR_ACTIVECAPTION	Active window caption.
COLOR_APPWORKSPACE	Background color of multiple document interface (MDI) applications.
COLOR_BACKGROUND, COLOR_DESKTOP	Desktop.
COLOR_BTNTEXT	Text on push buttons.
COLOR_CAPTIONTEXT	Text in caption, size box, and scroll bar arrow box.
COLOR_GRAYTEXT	Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	Item(s) selected in a control.
COLOR_HIGHLIGHTTEXT	Text of item(s) selected in a control.
COLOR_INACTIVEBORDER	Inactive window border.
COLOR_INACTIVECAPTION	Inactive window caption.
COLOR_INACTIVECAPTIONTEXT	Color of text in an inactive caption.
COLOR_INFOBK	Background color for tooltip controls.
COLOR_INFOTEXT	Text color for tooltip controls.
COLOR_MENU	Menu background.
COLOR_MENUTEXT	Text in menus.
COLOR_SCROLLBAR	Scroll bar gray area.
COLOR_WINDOW	Window background.
COLOR_WINDOWFRAME	Window frame.
COLOR_WINDOWTEXT	Text in windows.

B. Substrings, Escape Sequences and Format Descriptors

Substrings and escape sequences are useful for formatting strings. While substrings are used in FilterMeister programs only, escape sequences are derived from the C programming language.

B.1 Substrings

Substrings are special variables used in strings. Whenever a string is presented to the user, substrings are exchanged by their original string values. If a substring has not defined, it is replaced by the character ”” (NULL). The following substrings can be used in FilterMeister:

!!	exclamation point
!A	text specified by the Author key (by default not defined)
!C	text specified by the Category key (by default `FilterMeister`)
!c	text specified by the Copyright key (by default not defined)
!D	text specified by the Description key (by default not defined)
!F	text specified by the Filename key (by default not defined)
!f	current Filter Case (as in ”Flat image, no selection”)
!H	Host application (as in ”Adobe Photoshop®”)
!M	current Image Mode (as in ”RGB Color”)
!T	text specified by the Title key (by default `My Filter`)
!t	text specified by the Title key (with ellipsis ”...” removed)
!V	text specified by the Version key (by default not defined)

Substrings are supported by:
– message windows (see chapter 2.7)
– user control properties `Text` and `ToolTip` (depending on user control) (see chapter 3.3)
– filter identification key `About` (see chapter 3.2)
– dialog run-time functions `setDialogText` and `setDialogTextv` (see chapter 3.1.2)
– user control run-time functions `setCtlText` and `setCtlTextv` (see chapter 3.4)

Substrings are not supported by:
– filenames
– filter identification keys `Author`, `Title`, `Category`, `Description`, `Copyright`, `Version` and `Filename`

Examples:

```
Info ("The author !A told you not to click that button!!");

About:  "!T by !A\nCopyright: !c\n!D";
```

B.2 Escape Sequences

Escape sequences replace special characters or keyboard keys which cannot be typed correctly in the string (e.g., tabulator or `ENTER` key). Escape sequences also help format a string in a proper way and are used for example in message windows, user control text or when outputting a text to a file. The following escape sequences can be used in FilterMeister:

\\	backslash
\n	new line
\t	horizontal tabulator (8 character jump)
\r	carriage return (has same effect as new line)
\’	apostrophe
\”	quotation marks

Escape sequences are supported by:
– message windows (see chapter 2.7)
– user control property `Text` (see chapter 3.3)*
– user control properties Image/Bitmap/Icon (only backslash as directory separator)
– filter identification keys `Author`, `Title`, `Category`, `Description`, `Copyright`, `Version`, and `About`
– user control run-time functions `setCtlText` and `setCtlTextv` (see chapter 3.4)

** user control classes LISTBOX and COMBOBOX need \n to separate items*

Escape sequences are not supported by:
– dialog run-time functions `setDialogText` and `setDialogTextv` (see chapter 3.1.2)
– filter identification key `Filename` (except for the obligatory backslash sequence as directory separator)
– user control property `ToolTip` (see chapter 3.3)

Examples:

```
Filename: "D:\\Photoshop 5.0\\Plug-Ins\\FilterMeister\\MyFilter.8bf"

Info ("Image width =\t%d\Image Height =\t%d", X, Y);

ctl[4]: LISTBOX, Text="Millimeters\nCentimeters\nInches\nYards"
```

B.3 Format Descriptors

Format descriptors (or format specifications) are similar to the substrings mentioned in B.1. Used within text strings, the format descriptors are placeholders for variables noted outside the string. A typical example for the usage of format descriptors:

```
Info("Your image is %d pixels wide and %d pixels high", X, Y);
```

In the above case, two *decimal* format descriptors (%d) are used in the string. These are replaced by the integer values returned by the variables X and Y. If the image's metrics are 300 × 250, the above code would pop up a window with the following text:

```
Your image is 300 pixels wide and 250 pixels high
```

Most of the FM functions except filter identification keys, tooltips and filenames support format descriptors. The *dialog title bar text* and the *user control text* allow format descriptors only with the functions `setDialogTextv` and `setCtlTextv`.

These is an extract of the format descriptors used in FilterMeister:

Descriptor	Description	Type
%d	signed decimal integer	int
%u	unsigned decimal integer	int
%o	unsigned octal integer	int
%x	unsigned hexadecimal integer (lowercase lettering)	int
%X	unsigned hexadecimal integer (upper case lettering)	int
%c	single-byte ASCII character	char
%s	ASCII string	string
%f	signed floating-point value	double
%e	signed floating-point value in exponential form (lower case 'e')	double
%E	signed floating-point value in exponential form (upper case 'E')	double

In addition to present the user with variable values, it is also possible to format the output value. For instance, it is sometimes desirable to show floating point values with only two figures after the decimal point. In other cases, it is desirable to align the values of a list.

The syntax is simple – a floating point value is set between the percent sign and descriptor character.

Example:

```
ForEveryTile:
{
Info("Five times ten equals %8.5d\nFive times ten equals %08.3d", 5 * 10, 5 * 10);
Info("Five times ten equals %6.2f\nFive times ten equals %06.3f", 5.0 * 10.0, 5.0 * 10.0);
return false;
}
```

This is what you'll get:

```
Five times ten equals    00050
Five times ten equals     050

Five times ten equals  50.00
Five times ten equals 00050.00
```

Note: For further format descriptor information,consult any book or world-wide-web site dealing with the programming languages C or C++.