The GIMP - Programmer's Reference Book

Marc Bless, bless@ai-lab.fh-furtwangen.de

Version 0.03 / 3 May 1996 - 30 Sept. 1996

This document describes the API of The GIMP v0.54/v0.6 and shows how to program plug-ins.

Contents

1 Introduction

1.1 Porting Notes (0.54 to 0.6)

Porting a plug-in from The GIMP version 0.54 to version 0.6 requires at least the correct handling of the three new image types with alpha channel (see section 2.3 for details).

1.2 About The Authors

All sections written by Marc Bless except section "How To Debug A Plug-In" by Federico Mena Quintero.

Send comments, bug reports, and other stuff to bless@ai-lab.fh-furtwangen.de (Marc Bless) or federico@nuclecu.unam.mx (Federico Mena Quintero).

1.3 Online Resources

Needs work...

2 Basic Concepts

2.1 What Is A Plug-In?

A plug-in is a (more or less) little program that enhances the functionality of The GIMP.

2.1.1 Plug-In Types

There are two different types of plug-ins:

file plug-ins

provide reading and writing of various graphic-file-formats.

"normal" plug-ins

calculate effects and filters to an image.

2.1.2 How Does The GIMP Communicate With A Plug-In?

Everytime you invoke a plug-in from the menu The GIMP executes the following steps:

1. Open two pipes: one for reading from and one for writing to the plug-in.
2. Create a child with `fork ()`.
3. Execute the plug-in with the pipe-file-descriptors as arguments.

As soon as the plug-in calls `gimp_init ()` (see section 4.2) the plug-in sets its pipe-file-descriptors to the given arguments and is ready for communication.

2.2 Images, Layers And Channels

Needs work...

2.3 Image Types

The GIMP provides three types of images: true color, grayscale and 8-bit indexed color images. An alpha-channel can be added to each of that three types so that we have six types altogether. The differences between these six types are shown in table 2.3.

Table 2.3: Differences between image types

image type	identifier	bytes per pixel	color depth
true color	RGB_IMAGE	3	16.777.216
true color with alpha	RGBA_IMAGE	4	16.777.216
grayscale	GRAY_IMAGE	1	256
grayscale with alpha	GRAYA_IMAGE	2	256
indexed	INDEXED_IMAGE	1	256
indexed with alpha	INDEXEDA_IMAGE	2	256

For it has become senseless to use indexed color images today, you will work with true color or grayscaled images almost. The main reason for programmers not to support indexed color images in their plug-ins is the impossibility of interpolating pixel-values in a color map.

2.3.1 True color images

Each pixel of a true color image consists of a RGB-triplet (see table 2.4). A RGB-triplet needs three bytes in memory, because each red, green and blue part of a color can take values between 0 and 255. These parts are known to The GIMP as color channels also.

With three color channels you can compose up to 256 * 256 * 256 = 256^3 = 16.777.216 different colors - more than enough for smooth coloring.

2.3.2 Grayscale images

In contrast to true color images a pixel of a grayscale image needs one byte in memory only. That is, because you have 256 different values for brightness: 0 represents black, 255 represents white. See table 2.4.

2.3.3 Indexed color images

The indexed color image can be called an awkwark image type for filter programming. As you see in table 2.3.3 any of the 256 available colors is defined in a color map.

Table 2.3.3: Color map of an indexed color image

col.	RGB-triplets
0	r000 g000 b000
1	r001 g001 b001
2	r002 g002 b002
...	...
254	r254 g254 b254
255	r255 g255 b255

It is not difficult to program something like brightness or gamma correction with a color map, but as soon as you need color interpolation to create smooth colors you obviously have no chance to do it in a nice way. One solution could be to convert the indexed image into a true color image, then calculate the filter and convert it back into an indexed image. But I think that doesn't look perfect.

Another disadvantage of indexed images is that although each red, green and blue value of the color map needs one byte, you actually have six bits per value. Sometimes you can set values 0...63, sometimes 0...255. It depends on the application to right-shift the full byte range by two bits to get back to six bits.

With three times six bits you can compose up to 64 * 64 * 64 = 64^3 = 262.144 different colors but you can only use 256 colors simultaneous.

2.4 Internal Representations of Images

The internal representation of the image data is shown in tables 2.4 and 2.4. The first pixel in memory is the upper-left corner of the image, the last one is the lower-right corner (remember, that every pixel needs as many bytes in memory as there are color channels in the image).

See section 4.3 for an example how to walk through the image data and handle the correct positions.

Table 2.4: Internal representation of true color images.

r g b RGB_1	r g b RGB_2	... ...	r g b RGB_x
r g b RGB_x+1	r g b RGB_x+2	... ...	r g b RGB_2x
...	...	...	...
r g b RGB_(y-1)x+1	r g b RGB_(y-1)x+2	... ...	r g b RGB_yx

Table 2.4: Internal representation of grayscale and indexed color images.

v_1	v_2	...	v_x
v_x+1	v_x+2	...	v_2x
...	...	...	...
v_(y-1)x+1	v_(y-1)x+2	...	v_yx

2.5 The `.gimprc` File

The `.gimprc` file should be placed in your home directory (~.gimprc/). It contains the following fields:

swap-path *path*

Defines the directory where swap files will appear.

brush-path *paths*

Defines the directories where The GIMP searches valid brushes at startup.

default-brush *brush-name*

Sets the default brush. The brush file has to be placed in any of the paths given in `brush-path`.

plug-in-path *paths*

Defines the directories where The GIMP searches for plug-ins when the plug-in is run.

file-plug-in *plug-in-name image-types plug-in-title extensions*

Defines available file plug-ins. *plug-in-name* is the name of the executable file plug-in. *image-types* consists of three digits where each digit is 0 or 1. It defines what image types the plug-in can save. The digits correspond to *indexed*, *grayscale* and *rgb*. *plug-in-title* is the name of the plug-in for The GIMP. This will most likely be the name of the plug-in. *extensions* defines the possible file extensions for the plug-in.

Examples: The `jpeg` filter can save grayscaled and true color images but no indexed images, the `gif` filter can save indexed and grayscaled images but no true color images. So you would define in `.gimprc`:

```
file-plug-in  jpeg    011  "jpeg"   "jpg,jpeg"
file-plug-in  gif     110  "gif"    "gif"
```

Note: In the original `.gimprc`-comments that come with The GIMP v0.54 the format of the line `file-plug-in` is defined as

`file-plug-in` *image-types plug-in-name plug-in-title extension*

which obviously is wrong.

plug-in *plug-in-name menu-location accelerator-key accelerator-text*

Defines available plug-ins. *plug-in-name* is the name of the executable plug-in. *menu-location* defines the appearance of the plug-in in the submenu *Filters*, e.g. if you define a *menu-location*

```
"Effects/Invert"
```

then `Effects` becomes a submenu of *Filters*. `Invert` is the menu item that starts the plug-in. Menus and submenus are sorted in alphabetical order. *accelerator-key* defines the keys that start the plug-in. That string has the format

```
"[Shift] [Alt] <Key> Letter"
```

accelerator-text appears in the menu entry of the plug-in.

Examples:

```
plug-in bleed "FX/Bleed"
plug-in edge  "FX/Edge Detect"   "Alt<Key>E"   "Alt+E"
plug-in gauss "Blur/Gaussian"    "Shift Alt<Key>B"   "Shft+Alt+B"
```

Note that paths are separated by "`:`".

3 Compiling A New Plug-In

The easiest way to compile a new plug-in is to use a make file. Take a look at the original `Makefile` coming with The GIMP and check it out.

If you want to compile a new plug-in called `newfilter.c` just add it to the following list in the make file:

```
FILTERSRC = blur.c relief.c to-gray.c       \
            to-color.c to-indexed.c         \
            and-so-on.c some-more-filters.c \
            just-add-the-new-one.c          \
            newfilter.c
```

That should work for many of your plug-ins. Test it in your shell and type

```
make newfilter
```

to see what happens. Everything is ok, if you get no errors. Suppose you get something like

```
gcc ... newfilter.c ...
/tmp/cca001461.o(.text+0x67f): undefined reference to `pow'
/tmp/cca001461.o(.text+0x6a8): undefined reference to `rint'
...
make: *** [newfilter] Error 1
```

then you probably have included the math-library and maybe some other libraries also that need special compiler options. In that case you have to add the following lines (depending on the used libraries) to the `Makefile`:

```
newfilter: newfilter.c $(LIBGIMP)
           -$(CC) $(CFLAGS) $(LINCLUDE) -o newfilter \
                  newfilter.c $(LIBGIMP) -lc -lm
```

That should fit for most plug-ins. If you *really* get errors after inserting these lines you should check the `Makefile` on the following stuff:

- Contains `INCLUDE` all directories to the header files?
- Contains `LINCLUDE` all directories to the libraries?
- Is your compiler able to find `libgimp.a`?

4 Programming Plug-Ins

4.1 Datatypes and Constants

First we need some type definitions and constants out of `gimp.h`. What we always have to check is the image type. Therefore exists the following enum:

```
typedef enum {
  RGB_IMAGE = 0,
  RGBA_IMAGE,
  GRAY_IMAGE,
  GRAYA_IMAGE,
  INDEXED_IMAGE,
  INDEXEDA_IMAGE,
  UNKNOWN_IMAGE,
} ImageType;
```

Before we can handle an image we need an image typedef:

```
typedef struct _Image *Image;
```

`struct _Image` is defined in `gimp.c` but we don't have to care about it. All information about an image could be determined by calling corresponding functions.

4.2 Initializing All Stuff

Right, let's start with our programming. We have to

```
#include "gimp.h"
```

for the communication with The GIMP main program. Our filter function `newfilter ()` has to be prototyped:

```
static void newfilter (Image, Image);
```

The GIMP provides a mechanism to handle both source and destination image with different pointers. We just have to define two pointers, one for the input and one for the output image:

```
static Image input,
             output;
```

That's all for the global declarations. Let's begin with the `main()` function:

```
int
main (argc, argv)
     int argc;
     char **argv;
{
  if (!gimp_init (argc, argv))
    return 0;

  /* do something */

  gimp_quit ();

  return 0;
}
```

Of course, this function does nothing but initializing the connection to The GIMP and exiting from it. The image handles `input` and `output` have to be defined (right after `gimp_init`):

```
if (!(input = gimp_get_input_image (0)))
  return 0;

if (!(output = gimp_get_input_image (0)))
{
  gimp_free_image (input);
  return 0;
}
```

As there are three types of images (RGB, gray and indexed) we have to determine whether our filter calculation may start or not. It depends on what the filter function is doing. In most cases we only operate on RGB and grayscaled images, so that we could write:

```
select (gimp_image_type (input))
{
  case RGB_IMAGE    :
  case GRAY_IMAGE   :
    newfilter (input, output);
    gimp_update_image (output);
    break;
  case INDEXED_IMAGE:
    gimp_message ("NewFilter: Can't operate on indexed image.");
    break;
  case UNKNOWN_IMAGE:
    gimp_message ("NewFilter: Can't operate on unknown image.");
    break;
}
```

See table 2.3 and section 4.1 for further explanation about image types.

Before we call `gimp_quit ()` (see above) we should free some memory:

```
gimp_free_image (input);
gimp_free_image (output);
```

You could use this code piece as a template for your own plug-ins. We expand that `main ()`-code later when we add dialogs to it. But the code structure is almost the same.

4.3 The Filter Function

We have defined a prototype of the `newfilter ()` function above (see Section 4.2). The function starts with some variable declarations:

```
static void
newfilter (linput, loutput)
    Image linput, loutput;
{
  long width, height;
  long channels, rowstride;
  unsigned char *src_row, *dest_row;
  unsigned char *src, *dest;
  short row, col;
  int x1, y1, x2, y2;
```

See table 4.3 for the meaning of an identifier.

Table 4.3: Identifiers of the filter function.

width	width of image
height	height of image
channels	color depth (bytes)
rowstride	length of a row
src_row	current row of source image
dest_row	current row of destination image
src	current column of source image
dest	current column of destination image
row	row counter
col	column counter
x1	most left ordinate of selection
y1	most upper ordinate of selection
x2	most right ordinate of selection
y2	most lower ordinate of selection

To fill the identifiers with correct values we can use some "standard" code:

```
  gimp_image_area (linput, \&x1, \&y1, \&x2, \&y2);

  width     = gimp_image_width (linput);
  height    = gimp_image_height (linput);
```

```
    channels  = gimp_image_channels (linput);
    rowstride = width * channels;

    src_row   = gimp_image_data (linput);
    dest_row  = gimp_image_data (loutput);

    x1       *= channels;
    x2       *= channels;

    src_row  += rowstride * y1 + x1;
    dest_row += rowstride * y1 + x1;
```

I think most of it is self-explaining. The reason we multiply `x1` and `x2` with `channels` is because the value of `width` gives the image width in pixel, but each pixel needs `channels` bytes of memory.

Both `src_row` and `dest_row` point at the first row of the selection. Now we can start some loops to increment the positions:

```
    for (row = y1; row < y2; row++)
    {
      src     = src_row;
      dest    = dest_row;

      for (col = x1; col < x2; )
      {
        int ch;

        for (ch = 0; ch < channels; ch++)
        {
          col++;
          *dest++ = (unsigned char)*src++;
        }
      }

      src_row  += rowstride;
      dest_row += rowstride;
    }
  }
```

The first started loop counts from the most upper row to the most lower row. The current column pointers `src` and `dest` point at the first column of the current row. The second loop counts from the most left to the most right column. Now we have to go through all channels in the next loop and just have to set and increment both `dest` and `src`.

At this point-when you are setting `dest`-begins the actual filter calculation. This is just an example that does nothing but copying the input to output. Remember to increment both `dest` and `src` in the inner loop!

After the second loop both `src_row` and `dest_row` are incremented by `rowstride`.

That's it.

4.4 Creating A New Image

It is very easy to create a new image out of a plug-in. The function

```
Image gimp_new_image (char      *TITLE,
                      long       WIDTH,
                      long       HEIGHT,
                      ImageType TYPE)
```

produces a new image with the name `TITLE`. Its geometry is `WIDTH` * `HEIGHT` pixels and its type is `TYPE`. It returns a image handle of type `Image`.

Example: If you want to create a 512 * 256 pixel true color image, you have to write:

```
...
Image newimage;
...
newimage = gimp_new_image ("New Image", 512, 256, RGB_IMAGE);
...
```

Ok, now there is a new image, but it is still hidden. To make an image visible just call:

```
gimp_display_image (newimage);
```

To get more information about an image use the `gimp_image_*` functions.

4.5 Color Functions

Needs work...

4.6 Adding Dialogs

The GIMP provides a large library of dialog functions and widgets. Every widget in a dialog needs a widget ID of type `int`. We have to declare an identifier for each ID of the dialog. First the dialog itself needs an ID:

```
int dlg_ID;
```

To create a new dialog we use the function `gimp_new_dialog ()`:

```
dlg_ID = gimp_new_dialog ("Another Plug-In");
```

There exist several function for creating new widgets, for separating and grouping widgets and for other needs (see tables 4.6, 4.6 and 4.6).

Table 4.6: Functions to create dialog widgets.

Function	Description
gimp_new_check_button ()	Create check button.
gimp_new_dialog ()	Create dialog.
gimp_new_image_menu ()	Create image choose box.
gimp_new_label ()	Create label of a widget.
gimp_new_push_button ()	Create push button.
gimp_new_radio_button ()	Create radio button.
gimp_new_scale ()	Create scale.
gimp_new_text ()	Create text widget.

Table 4.6: Functions to group dialog widgets.

Function	Description
gimp_new_column_group ()	Create group box, members are shown in a column.
gimp_new_frame ()	Create frame.
gimp_new_row_group ()	Create group box, members are shown in a row.

Table 4.6: Functions for additional dialog handling.

Function	Description
gimp_add_callback ()	Add a callback function to a widget.
gimp_cancel_item_id ()	Returns handle of cancel button.
gimp_change_item ()	Change a widgets value.
gimp_close_dialog ()	Exit the dialog.
gimp_delete_item ()	Delete an item of a dialog.
gimp_hide_item ()	Make a widget invisible.
gimp_ok_item_id ()	Returns handle of ok button.
gimp_show_dialog ()	Runs dialog and waits on exit.
gimp_show_item ()	Make a widget visible.

4.7 Groups of Radio Buttons

First some identifiers for that example:

```
int DialogID, OtherGroupID;
int RadioGroupID, FirstRadioID, SecndRadioID;
int first, second;
```

You can add a group box containing radio buttons to a dialog by simply switching the `type` parameter of either the `gimp_new_row_group ()` or the `gimp_new_column_group ()` function to `RADIO`:

```
RadioGroupID = gimp_new_row_group (DialogID, OtherGroupID, RADIO, "");

FirstRadioID = gimp_new_radio_button (DialogID, RadioGroupID, "First");
```

```
SecndRadioID = gimp_new_radio_button (DialogID, RadioGroupID, "Second");
...
```

It is recommended to initialize the radio widgets, so that the dialog represents the internal values correctly to the user. Note that only one member of a radio button group has to have a true value (1), all other members have to be zero:

```
first  = 1;
second = 0;
gimp_change_item (DialogID, FirstRadioID, sizeof (first), \&first);
gimp_change_item (DialogID, SecndRadioID, sizeof (second), \&second);
...
```

Add a callback function for every radio button:

```
gimp_add_callback (DialogID, FirstRadioID, radio_callback, \&first);
gimp_add_callback (DialogID, SecndRadioID, radio_callback, \&second);
...
```

Last you have to define the callback function (don't forget prototyping):

```
static void
radio_callback (int item_ID, void *client_data, void *call_data)
{
*((long*) client_data) = *((long*) call_data);
}
```

4.8 Showing Previews

As it would be nice to turn on and off the preview we insert a check-box into the dialog. For this we need an `int` as handle and a `long` as value:

```
static long aapply;
static int  aapply_ID;
```

To restore the original picture, we need some memory and a pointer for it:

```
static unsigned char *saved;
```

Of course there are some lines of prototypes:

```
static void toggle_callback (int, void *, void *);
static void saveimage(void);
static void freshen(void);
```

The **saveimage ()** function should be called after we checked that we are able to calculate anything (i.e. after gimp-init and image-checks):

```
saveimage ();
```

To add the check-box to the dialog, the following lines have to be at a good place between dialog-init and -exit:

```
aapply_ID = gimp_new_check_button (dialog_ID, group_ID,
                                   "Auto Apply");
aapply = 1;
gimp_change_item (dialog_ID, aapply_ID,
                  sizeof (aapply), \&aapply);
gimp_add_callback (dialog_ID, aapply_ID,
                   toggle_callback, \&aapply);
```

So we need the callback function. If we are in preview-mode the filter will be calculated with the current values. If we are *not* in preview-mode the original image will be restored.

```
static void
toggle_callback (item_ID, client_data, call_data)
     int item_ID;
     void *client_data;
     void *call_data;
{
  *((long*) client_data) = *((long*) call_data);

  if (aapply)
  {
    filter (input, output);      /* our filter function */
    gimp_update_image (output);
  }
  else
  {
    freshen();
    gimp_update_image (output);
  }
}
```

Finally, we need the **saveimage ()** and **freshen ()** functions:

```
static void saveimage(void)
{
  saved = (unsigned char *) malloc(gimp_image_width(input) *
                            gimp_image_height(input) *
                            gimp_image_channels(input));
  memcpy(saved, gimp_image_data(input),
```

```
              gimp_image_width(input)*
              gimp_image_height(input)*
              gimp_image_channels(input));
}

static void freshen(void)
{
  memcpy(gimp_image_data(output), saved,
              gimp_image_width(input)*
              gimp_image_height(input)*
              gimp_image_channels(input));
}
```

To apply the preview "on demand" for each change of values (e.g. slider-values, radio- or check-buttons) we have to extend the callbacks in the following way. If preview-mode is turned on and values had changed we have to update the preview, else we just set the new values.

```
static void
scale_callback (item_ID, client_data, call_data)
     int item_ID;
     void *client_data;
     void *call_data;
{
  if (aapply \&\&
      (*((long*) client_data) != *((long*) call_data)))
  {
    *((long*) client_data) = *((long*) call_data);

    filter (input, output);
    gimp_update_image (output);
  }

  *((long*) client_data) = *((long*) call_data);
}
```

When calculating your filter function you have to set a pointer to the source data. I have made the experience that if you use `gimp_get_input_image (0)` for that pointer you will get "ugly" results, because you get the data from the last calculated destination image.

Imagine the source pointer identifier is called `src_row`. Then you won't use

```
src_row = gimp_get_input_image (0);
```

but

```
src_row = saved;
```

so that you get the correct data for the next preview.

4.9 Progress Bars

To add a progress bar to your plug-in use the functions `gimp_init_progress ()` and `gimp_do_progress ()`. Initialize it right after `gimp_show_dialog ()`:

```
if (gimp_show_dialog (dialog_ID)
{
  gimp_init_progress ("New Filter");

  newfilter (input, output);
}
```

Define two identifiers for the current progress and the maximum progress value which is the height of the selected area almost:

```
int progress;
int max_progress;

...

progress     = 0;
max_progress = y2 - y1;
```

Increase `progress` after each calculated row and do an update of the progress bar after some rows:

```
progress++
if (progress % 5 == 0)
  gimp_do_progress (progress, max_progress);
```

The value of `max_progess` depends on what your plug-in is doing. I think the example shown here will fit for most filters that walk straight through the image.

To close the current progress dialog you have to do

```
gimp_do_progress (1, 1);
```

4.10 Creating Additional Dialogs

See the Lighting plug-in (Tom Bech and Federico Mena Quintero) and the Lensflare plug-in (Marc Bless) for examples.

Needs work...

4.11 File-Plug-Ins

Needs work...

4.12 Storing And Recovering Dialog Values

You can let the gimp executable store and recover your data of any size. The usage of the functions `gimp_get_params ()` and `gimp_set_params ()` is very easy and will be explained with the source code of the `gauss_recurse.c` plug-in (Spencer Kimbal and Peter Mattis).

This plug-in creates a dialog with an entry field for a real value (double) and two check buttons (long), so this structure is used:

```
typedef struct
{
  double radius;
  long   horizontal;
        long   vertical;
} BlurValues;

BlurValues vals;
```

As soon as all image consistency at startup is done, `vals` should be filled with either default or previously saved values:

```
void *data;

data = gimp_get_params ();
if (data)
  vals = *((BlurValues *) data);
else
{
  vals.radius     = 5.0;
  vals.horizontal = 1;
  vals.vertical   = 1;
}
```

Now the dialog widgets will be defined:

```
char buf[16];

sprintf (buf, "%0.2f", vals.radius");
text_ID = gimp_new_text (dialog_ID, temp_ID, buf);

horz_ID = gimp_new_check_button (dialog_ID, temp_ID, "Horizontal");
vert_ID = gimp_new_check_button (dialog_ID, temp_ID, "Vertical");

gimp_change_item (dialog_ID, horz_ID, sizeof (vals.horizontal), \&vals.horizontal);
gimp_change_item (dialog_ID, vert_ID, sizeof (vals.vertical), \&vals.vertical);

gimp_add_callback (dialog_ID, text_ID, text_callback, \&vals.radius);
gimp_add_callback (dialog_ID, horz_ID, toggle_callback, \&vals.horizontal);
gimp_add_callback (dialog_ID, vert_ID, toggle_callback, \&vals.vertical);
```

If the dialog ends with "Ok" the structure is given to the Gimp:

```
  if (gimp_show_dialog (dialog_ID))
  {
    gimp_set_params (sizeof (BlurValues), \&vals);
    /* do all stuff ... */
        }
```

To be a little complete, here are the callbacks:

```
static void *text_callback (int item_ID, void *client_data, void *call_data)
{
  *((double *) client_data) = atof (call_data);
}

static void *toggle_callback (int item_ID, void *client_data, void *call_data)
{
  *((long *) client_data) = *((long *) call_data);
}
```

5 How To Debug A Plug-In

by Federico Mena Quintero

Writing Gimp plug-ins is very easy, due to the simple but powerful structure of the API (applications programming interface) provided by The Gimp to plug-in developers. However, some people have had problems when attempting to debug plug-ins. This short tutorial tries to explain just how to do that, using the very fine GDB (the GNU debugger).

5.1 Why is it not obvious how to debug plug-ins?

When writing 'normal', stand-alone programs, you simply have to use the -g option with gcc (or whatever option your compiler uses to enable compilation with debugging information). Using GDB, a typical debug sequence may be started as follows:

```
$ gdb someprogram
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.14 (i486-slackware-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb)
```

You can then easily start to debug your program by using GDB's "run" command. If you want, you can set breakpoints and such before you run the program, set up signal handlers, etc.

However, plug-ins are somewhat different beasts. It is very easy to debug stand-alone programs, because you actually launch them from GDB (i.e. you know exactly when they begin execution and such). However, plug-ins get started by The Gimp, not by you. So how do you debug them?

5.2 The power of GDB

GDB is a very powerful debugger. Among many other things, it lets you debug processes which are already running, and that's just what we need for debugging our plug-ins.

Let's say your plug-in is called **foobar**. First of all, make sure you have compiled it using gcc's "-g" option, or whatever options your compiler needs to produce debugging information.

For now, let's assume that your plug-in uses a dialog box to let the user set its parameters before doing its work. This is good, because the plug-in does not do anything while the dialog box is running (i.e. it is only after you click the OK button that the plug-in actually starts its interesting work). So we'll use that 'pause' time to fire up GDB and tell it to 'attach' to our already-running plug-in.

Start the **foobar** plug-in using the normal Gimp menus. If all goes well up to the dialog box creation (we'll assume that for now; later I will explain how to debug this kind of problems), you'll be shown the dialog box, and the plug-in will happily sit idle until you click the OK button. Now we have to figure out which running process corresponds to our plug-in. That's what the "ps" command is for, and by looking at it we'll know the PID (process-ID) of our plug-in:

```
$ ps
  PID TTY STAT  TIME COMMAND
   67 v01 SW    0:00 (bash)
  342 v01 SW    0:00 (startx)
  346 v01 SW    0:00 (xinit)
  349 v01 S     0:02 /usr/openwin/bin/olvwm
  355 v01 SW    0:00 (olwmslave)
  357 pp0 SW    0:00 (bash)
  367 pp0 S     0:00 xclock -geometry 66x66+539-0 -padding 2
  368 pp0 S     5:16 procmeter
  370 pp0 S     0:00 xload -geometry 239x66+290-0 -scale 3 -update 2 -hl red -nolabel -jumpscroll 1
  371 pp0 S     0:42 emacs
  374 pp1 S     0:00 -bash
  383 pp2 S     0:00 -bash
  392 pp3 SW    0:00 (bash)
  401 pp4 SW    0:00 (bash)
  410 pp5 S     0:00 (bash)
  419 pp6 SW    0:00 (bash)
  428 pp6 S     0:00 xmixer
  430 pp0 S     0:31 netscape -install -geometry 1014x738+0+0
  628 pp5 S     0:02 ./gimp
  629 pp5 S     0:00 /home/quartic/gimp/plug-ins/foobar -gimp 6 5
  631 pp2 R     0:00 ps
```

From this output, we see that the **foobar** process' PID is number 629. Don't worry about those weird command-line parameters; they are used solely by the Gimp's API to set up the communications stuff between the plug-in and the Gimp.

Let's fire up GDB. On an xterm or similar, switch to the directory where your plug-in's source and binary reside. Now fire up GDB:

```
$ gdb foobar
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.14 (i486-slackware-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb)
```

GDB has only loaded the debugging information from the plug-in's binary; it has *not* run the plug-in. Now we will use GDB's "attach" command to do just that, to attach to a running process. This is one of the beauties of Unix and GDB; you do not have to be the parent process of another process to be able to debug it. You just have to be its owner. The plug-in's PID was 629, so we will use the following command:

```
(gdb) attach 629
Attaching to program '/home/quartic/gimp/plug-ins/plasma', process 629
0x50016ead in channels ()
```

The stuff in the last line may vary. The following things have happened. GDB has temporarily become the parent process of our plug-in, so that we can control its execution. Also, the running process of the plug-in has been stopped.

You can now set a breakpoint in your plug-in's main 'work' function or whatever. For example, if that function is called do_foobar(), you could use

```
(gdb) break do_foobar
Breakpoint 1 at 0x8000d96: file foobar.c, line 180.
```

Now we have to resume execution of the plug-in's process (remember, it was stopped by GDB). Use GDB's "cont" command for this.

```
(gdb) cont
Continuing.
```

Now return to your plug-in's dialog box, configure its parameters, and click the OK button. GDB will stop the program when it reaches the do_foobar() function, at the point where we set the breakpoint. GDB will look more or less like this:

```
Breakpoint 1, do_foobar (input=0x800a000, output=0x800c000, turb=0) at foobar.c:180
180             gimp_image_area(input, \&ix1, \&iy1, \&ix2, \&iy2);
```

After that, you can use GDB's usual commands for stepping through your program, setting watchpoints, breakpoints, etc. When it has finished execution (or if it has crashed), GDB will tell you. Note that you don't have to fire up GDB every time you want to re-run the plug-in; you just have to use the "attach" command again, to attach to the appropriate process. Remember that if the binary file has changed (say, because of a recompilation), you'll have to reload it.

5.3 Debugging plug-ins before the dialog box

The above stuff is all you need to know to start debugging plug-ins which invoke dialog boxes (and you get them right). However, some plug-ins do not have dialog boxes (so you don't have an implied pause before the plug-in starts its work), or you actually want to debug the dialog box's creation process or your widget callback functions.

There are several methods to accomplish this. The easiest one is to insert a getchar() in the main() function of your plug-in, before everything else runs. When you invoke the plug-in from the Gimp, you won't see anything. You have to switch to the xterm from where you started the Gimp, and press a key so that your plug-in will continue its execution.

You can use that 'pause' time to find about your plug-in's PID, fire up gdb, attach to the process, set up breakpoints, and use the "cont" command to let it continue. After that, switch to the xterm from where you started the Gimp and do press a key to let your plug-in continue.

Another way is to insert a sleep() in main() with a pause long enough to let you do all the GDB stuff. There are many ways to insert pauses in the plug-in's execution, but perhaps the most flexible is the getchar() one. Experiment.

5.4 Some plug-in programming tips

Here are some tips for when you are starting to develop Gimp plug-ins. Many people get surprised by some of these little details, so you have to be careful. You'll save some hours of debugging time, as well.

- By far, the most common bug in plug-ins is the incorrect calculation of offsets within the image data to locate a particular pixel. The correct byte offset inside an image to a pixel in position (x, y) is calculated by the formula

  ```
  pixeloffset = (width * y + x) * channels
  ```

  The multitplication by the number of channels is **very important!!!**b If you don't do it, your plug-in will only work with grayscale or indexed-color images (because they use exactly 8 bits per pixel). But for images with a larger number of channels (RGB, RGBA, GA, etc.), you *have* to do the multiplication.

- Remember that in your dialog boxes, every widget which accepts data **must** have a correct callback function. Callback functions simply copy (and optionally convert) the data from the way the widget represents it to whatever format your plug-in may need. If your callbacks are wrong, you will get garbage data. Most of the time you can simply yank the callbacks you need from other plug-ins, and they will work fine. If you want to do more exotic things like having a widget change its data when another widget changes (like for example, linking a slider to a numeric/text input field), or having an auto-apply function, you should carefully read the code from other people's plug-ins so that you can see how it is done.

- It is very important to call gimp_update_image() after your plug-in has done its work! Otherwise, you will get a black output image. This can get frustrating, because you may think that your plug-in has not done anything :)

- If you are in the process of creating or modifying a plug-in's dialog boxes (for example, adding and changing widgets and data structures), you may notice that sometimes the dialog starts with garbage data, or some widgets don't show on the dialog, etc. This is almost certainly caused by the fact that you are using gimp_get_params() and gimp_set_params() to let your plug-in preserve its dialog's data from one invocation to the next. However, the Gimp cannot know if the internal organization of widgets or if your data structures have changed, so you could get data that *was* valid in the previous invocation of a plug-in, but is no longer so with the newer version. So if you are still writing your dialogs and changing data structures, it is better *not* to use gimp_get_params() and gimp_set_params() until the dialog is complete and working. That way the plug-in can re-create its data correctly every time, instead of getting possible garbage from previous runs.

- You can **not** write to the source image's data, because its shared memory segment is marked read-only by the Gimp. If you try to modify it, your plug-in will get a segmentation fault and die.

- If you want to create new windows for output images, remember that you have to call gimp_new_image() to create a new output image, then operate on it, and finally call gimp_display_image() and gimp_update_image() (in that order) to show the new windows.

- Remember that output images are **black** when your plug-in starts. So you have to paint all of the output area, or your image will come up with black parts. An exception to this is when a selection is active. If you your plug-in paints outside the selection, those changes will be ignored (i.e. the Gimp only takes into account whatever is inside the selection).

- Since you are accessing an image's pixels directly, the Gimp can not provide for clipping functions and such. All clipping and coordinate checking must be done by your plug-in. If you try to access a pixel outside the image, either you'll get garbage, or your plug-in will segfault. Check your coordinates carefully and remember to do the correct offset calculations.

- Finally, and most important of all: **do not reinvent the wheel!** Most plug-ins are protected under the GNU Public License (GPL), so you can freely modify them and use parts of their code if you preserve the copyright. That's exactly the purpose of the GPL, to let programmers use other people's code with very few restrictions. Also, consider whether it is better to write a whole new plug-in or to just extend an existing plug-in's functionality. This was done with the BCG plug-in for example. If you run out of ideas for plug-ins, try to improve other people's code. Optimize the algorithms, fix bugs, etc. There are many ways to contribute!

6 Useful Routines

6.1 Coordinate Handling

Sometimes you want to get a pixel of an image by referring to it as coordinates *(x,y)*. The following functions return a relative position of a pixel that has to be added to the pointer of the image data:

```
long int XYToIndex(int x,int y)
{
  return( (long int)x * (long int)channels +
          (long int)y * (long int)rowstride);
}
```

See section 4.3 for a description of `channels` and `rowstride`.

Example of usage:

```
*(dest + XYToIndex (x, y)) =
  *(src + XYToIndex (width - x, height - y));
```

7 Complete Plug-In-Templates

The following programs represent two templates, one with and one without preview function.

7.1 A Simple Plug-In

```
#include <stdlib.h>
#include "gimp.h"

#define INVALID_IMAGE_TYPE "newfilter: cannot operate on indexed or unknown image type"

static void newfilter (Image, Image);

static char *prog_name;

Image input, output;

int main (int argc, char **argv)
{
  prog_name = argv[0];

  if (!gimp_init (argc, argv))
    exit (0);
```

```
  input = 0;
  output = 0;

  if (!(input = gimp_get_input_image (0))
  {
    gimp_quit ();
    exit (0);
  }

  if (!(output = gimp_get_output_image (0))
  {
    gimp_free_image (input);
    gimp_quit ();
    exit ();
  }

  switch (gimp_image_type (input))
  {
    case RGB_IMAGE:
    case GRAY_IMAGE:

        newfilter (input, output);
        gimp_update_image (output);
        break;

    case INDEXED_IMAGE:

        gimp_message (INVALID_IMAGE_TYPE);
        break;

    default:

        gimp_message (INVALID_IMAGE_TYPE);
        break;
  }

  gimp_free_image (input);
  gimp_free_image (output);

  gimp_quit ();

  return 0;
}
```

```
static void newfilter (Image linput, Image loutput)
{
  long           width, height, channels, rowstride;
  unsigned char *src_row, *dest_row, *src, *dest;
  short          row, col;
  int            x1, y1, x2, y2, i;

  gimp_image_area (linput, \&x1, \&y1, \&x2, \&y2);

  width     = gimp_image_width (linput);
  height    = gimp_image_height (linput);
  channels  = gimp_image_channels (linput);
  rowstride = width * channels;

  src_row   = gimp_image_data (linput);
  dest_row  = gimp_image_data (loutput);

  x1       *= channels;
  x2       *= channels;

  src_row  += rowstride * y1 + x1;
  dest_row += rowstride * y1 + x1;

  for (row = y1; row < y2; row++)
  {
    src  = src_row;
    dest = dest_row;

    for (col = x1; col < x2; col++)
    {
      for (i = 0; i < channels; i++)
        *dest++ = *src++;
    }

    src_row  += rowstride;
    dest_row += rowstride;
  }
}
```

7.2 A Plug-In With Preview

```
#include <stdlib.h>
#include "gimp.h"
```

```
#define INVALID_IMAGE_TYPE "newfilter: cannot operate on indexed or unknown image type"
#define DIALOG_TITLE       "newfilter"

static void newfilter (Image, Image);
static void cb_ok (int, void *, void *);
static void cb_cancel (int, void *, void *);
static void cb_toggle (int, void *, void *);
static void saveimage (void);
static void freshen (void);
static void init_dialog (void);

static char *prog_name;

static int  dialog_ID;
static long aapply;
static int  aapply_ID;

Image input, output;

static unsigned char *saved;

int main (int argc, char **argv)
{
  int text_ID, frame_ID, main_ID, grp_ID;

  prog_name = argv[0];

  if (!gimp_init (argc, argv))
    exit (0);

  input  = 0;
  output = 0;

  if (!(input = gimp_get_input_image (0)))
  {
    gimp_quit ();
    exit (0);
  }

  if (!(output = gimp_get_output_image (0)))
  {
    gimp_free_image (input);
    gimp_quit ();
    exit (0);
```

```
  }

  switch (gimp_image_type (input))
  {
    case RGB_IMAGE:
    case GRAY_IMAGE:

          saveimage ();
          init_dialog ();

          newfilter (input, output);
          gimp_update_image (output);

          if (!gimp_show_dialog (dialog_ID))
            freshen ();
          else
            newfilter (input, output);

          gimp_update_image (output);
          free (saved);
          break;

    case INDEXED_IMAGE:

          gimp_message (INVALID_IMAGE_TYPE);
          break;

    default:

          gimp_message (INVALID_IMAGE_TYPE);
          break;
  }

  gimp_free_image (input);
  gimp_free_image (output);

  gimp_quit ();

  return 0;
}

static void init_dialog (void)
{
  dialog_ID = gimp_new_dialog (DIALOG_TITLE);
```

```
  main_ID   = gimp_new_row_group (dialog_ID, DEFAULT, NORMAL, "");

  aapply_ID = gimp_new_check_button (dialog_ID, main_ID, "Auto Apply");
  aapply    = 1;
  gimp_change_item (dialog_ID, aapply_ID, sizeof (aapply), \&aapply);
  gimp_add_callback (dialog_ID, aapply_ID, cb_toggle, \&aapply);

  gimp_add_callback (dialog_ID, gimp_ok_item_id (dialog_ID), cb_ok, 0);
  gimp_add_callback (dialog_ID, gimp_cancel_item_id (dialog_ID), cb_cancel, 0);
}

static void cb_toggle (int item_ID, void *client_data, void *call_data)
{
  *((long*) client_data) = *((long*) call_data);

  if (aapply)
  {
    newfilter (input, output);
    gimp_update_image (output);
  }
  else
  {
    freshen ();
    gimp_update_image (output);
  }
}

static void saveimage (void)
{
  saved = (unsigned char *) malloc (gimp_image_width (input) *
                                    gimp_image_height (input) *
                                    gimp_image_channels (input));
  memcpy (saved, gimp_image_data (input),
          gimp_image_width(input) *
          gimp_image_height (input) *
          gimp_image_channels (input));
}

static void freshen (void)
{
  memcpy (gimp_image_data (output), saved,
          gimp_image_width (input) *
          gimp_image_height (input) *
          gimp_image_channels (input));
```

```
}

static void cb_ok (int item_ID, void *client_data, void *call_data)
{
  gimp_close_dialog (dialog_ID, 1);
}

static void cb_cancel (int item_ID, void *client_data, void *call_data)
{
  gimp_close_dialog (dialog_ID, 0);
}

static void newfilter (Image linput, Image loutput)
{
  long           width, height, channels, rowstride;
  unsigned char *src_row, *dest_row, *src, *dest;
  short          row, col;
  int            x1, y1, x2, y2, i;

  gimp_image_area (linput, \&x1, \&y1, \&x2, \&y2);

  width     = gimp_image_width (linput);
  height    = gimp_image_height (linput);
  channels  = gimp_image_channels (linput);
  rowstride = width * channels;

  src_row   = saved;
  dest_row  = gimp_image_data (loutput);

  x1       *= channels;
  x2       *= channels;

  src_row  += rowstride * y1 + x1;
  dest_row += rowstride * y1 + x1;

  for (row = y1; row < y2; row++)
  {
    src  = src_row;
    dest = dest_row;

    for (col = x1; col < x2; col++)
    {
      for (i = 0; i < channels; i++)
        *dest++ = *src++;
```

```
    }

    src_row  += rowstride;
    dest_row += rowstride;
  }
}
```

8 GIMP Library Reference

void gimp_add_callback (int DIALOG_ID, int ITEM_ID, GimpItemCallbackProc CALLBACK, void *CALLBACK_DATA)

This function adds `CALLBACK` as callback function of the item `ITEM_ID`. The callback function handles `CALLBACK_DATA` as data of the item.

void gimp_background_color (unsigned char *RED, unsigned char *GREEN, unsigned char *BLUE)

This function returns the current background color in the GIMP application.

int gimp_cancel_item_id (int DIALOG_ID)

This function returns the handle of the Cancel-Button of dialog `DIALOG_ID`. You need it to add a callback for the Button. *See also:* **gimp_add_callback ()**, **gimp_ok_item_id ()**

void gimp_change_item (int DIALOG_ID, int ITEM_ID, long DATA_SIZE, void *DATA)

This function sets the value of `DATA` as new value of the item `ITEM_ID`. The size of the `DATA`-type has to be stored in `DATA_SIZE`.

void gimp_close_dialog (int DIALOG_ID, int RETURN_VAL)

This function closes the dialog with the handle `DIALOG_ID`. The value of `RETURN_VAL` will be returned by `gimp_show_dialog ()`. *See also:* **gimp_show_dialog ()**

void gimp_delete_item (int DIALOG_ID, int ITEM_ID)

This function deletes the item `ITEM_ID` from the dialog `DIALOG_ID`.

void gimp_destroy_image (Image IMAGE)

This function tells the GIMP library to free its link to `IMAGE` and to tell the GIMP to also free its link. The result is that `IMAGE` will be destroyed.

void gimp_display_image (Image IMAGE)

This function displays an image. A new view is created in the process. Therefore, it is not necessary to display an image that is already displayed. (This function is used mainly by file plug-ins which must tell the GIMP to display a newly created image).

void gimp_foreground_color (unsigned char *RED, unsigned char *GREEN, unsigned char *BLUE)

This function returns the current foreground color in the GIMP application.

void gimp_free_image (Image IMAGE)

This function tells the GIMP library to free its link to `IMAGE`. The image is not actually destroyed since the GIMP will retain its link.

Image gimp_get_input_image (long ID)

This function gets an input image with the given `ID`. Every image in the main GIMP application has a unique `ID` value assigned to it at creation. Values are assigned starting from 1. This function can be used to get any valid image in the GIMP. However, this action is not recommended since it is impossible to tell which `ID`'s are valid.

The main usage is to get the default image which is specified by an `ID` value of 0. (This is used since no actual image in the GIMP will have an `ID` of 0).

Input images are defined to be read-only. That is, the programmer should not modify the image. (This is enforced by the library by only read-only to the shared memory segment).

Image gimp_get_output_image (long ID)

`gimp_get_output_image` is analogous to `gimp_get_input_image` except that it gets an output image. An output image is defined to be read-write. Meaning that data can be read to and written from. It may seem strange, but getting both the input image and output image for the same `ID` actually returns two different images and two different data segments. Writing to the input image does not affect the output image and vice-versa.

void *gimp_get_params (void)

This functions returns a pointer to a previously saved data structure.

See also: `gimp_set_params`

gimp.h

This is the standard header for GIMP plug-ins. It has to be included with

```
#include "gimp.h"
```

.

void gimp_hide_item (int DIALOG_ID, int ITEM_ID)

This function makes the item `ITEM_ID` of the dialog `DIALOG_ID` invisible.

void gimp_image_area (Image IMAGE, int *X1, int *Y1, int *X2, int *Y2)

This function returns the active area of operation for a plug-in. The area is returned as a rectangle defined by the upper-left corner (X1, Y1) and the lower-right corner (X2, Y2). This is useful for effects filters which have their results masked by the current selection in an image. By only operating within the active area a slow running plug-in can dramatically increase its speed. (Even a fast running plug-in can dramatically increase its speed since the user may have only selected a small portion of the image).

It should be noted that a plug-in should run the same whether or not it uses the information about the active image area. That is, operating on the entire image and operating on the active area should have the same end effect from the users point of view.

long gimp_image_channels (Image IMAGE)

This function returns an images channels. RGB images have 3 channels. Grayscale and indexed color images have 1 channel.

void* gimp_image_cmap (Image IMAGE)

This function returns the colormap associated with an image. It is only valid for indexed color images. The colormap is a series of RGB triplets where each triplet is 3 bytes. The size of the colormap is always 768 bytes, however, only the first `gimp_image_colors (IMAGE)` triplets are valid.

long gimp_image_colors (Image IMAGE)

This function returns the number of colors in an indexed color image. It is only valid for indexed color images.

void* gimp_image_data (Image IMAGE)

This function returns the actual pixel data for the image. For RGB images the data is laid out as a series of RGB triplets where each triplet is 3 bytes. For grayscale and indexed color images, each pixel value is a single byte corresponding to a grayscale value or pixel index respectively.

long gimp_image_height (Image IMAGE)

This function returns an images height.

char* gimp_image_name (Image IMAGE)

This function returns an images name.

ImageType gimp_image_type (Image IMAGE)

This function returns the type of an image. The type is one of `RGB_IMAGE`, `GRAY_IMAGE`, `INDEXED_IMAGE`, or `UNKNOWN_IMAGE`. If `UNKNOWN_IMAGE` is returned an error has occurred within the library. (ie. This should never happen).

long gimp_image_width (Image IMAGE)

This function returns an images width.

int gimp_init (int ARGC, char **ARGV)

This function intializes the GIMP library. The arguments `ARGC` and `ARGV` are simply the `ARGC` and `ARGV` values passed into `main()` upon invocation. Returns ¿0 (true) if initialization was ok. *See also:* `gimp_quit ()`

void gimp_install_load_save_handlers (GimpLoadSaveProc load, save)

This function installs two handlers for file plug-ins. `load` and `save` refer to the file load and save functions.

int gimp_new_check_button (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new check button. It returns a handle of the new check button.

int gimp_new_column_group (int DIALOG_ID, int PARENT_ID, int GROUP_TYPE, char *TITLE)

This function creates a new group box. Members of the new box are shown in a column. It returns a handle of the new group box.

int gimp_new_dialog (char *TITLE)

This function creates a new dialog for user interactions. It returns a handle of the new dialog window.

int gimp_new_frame (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new frame. It returns a handle of the new frame.

Image gimp_new_image (char *TITLE, long WIDTH, long HEIGHT, ImageType TYPE)

This function creates a new image with the given title, width, height and type. The new image is returned in the opaque type `Image`. Use the `gimp_image_*` functions to get information out of the image.

`TYPE` should be specified as one of `RGB_IMAGE`, `GRAY_IMAGE` or `INDEXED_IMAGE` to create a new RGB image, grayscale image or indexed color image respectively.

int gimp_new_image_menu (int DIALOG_ID, int PARENT_ID, char CONSTRAINT, char *TITLE)

This function creates an image choose box which contains all opened images. `CONSTRAINT` is composed by combining the constants `IMAGE_CONSTRAIN_RGB`, `IMAGE_CONSTRAIN_GRAY`, `IMAGE_CONSTRAIN_INDEXED` and `IMAGE_CONSTRAIN_ALL` with OR operators.

int gimp_new_label (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new label. It returns a handle of the new label.

int gimp_new_push_button (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new push button. It returns a handle of the new push button.

int gimp_new_radio_button (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new radio button. It returns a handle of the new radio button.

int gimp_new_row_group (int DIALOG_ID, int PARENT_ID, int GROUP_TYPE, char *TITLE)

This function creates a new group box. Members of the new box are shown in a row. It returns a handle of the new group box.

int gimp_new_scale (int DIALOG_ID, int PARENT_ID, long MIN, long MAX, long START_VALUE, long PRECISION)

This function creates a new scale. It returns a handle of the new scale.

int gimp_new_text (int DIALOG_ID, int PARENT_ID, char *TITLE)

This function creates a new text entry field. It returns a handle of the new text entry field.

int gimp_ok_item_id (int DIALOG_ID)

This function returns the handle of the OK-Button of dialog `DIALOG_ID`. You need it to add a callback for the Button. *See also:* `gimp_add_callback ()`, `gimp_cancel_item_id ()`

void gimp_quit (void)

This function closes all shared memory blocks and disconnects from The GIMP. *See also:* `gimp_init ()`

void gimp_set_image_colors (Image IMAGE, void *CMAP, long NCOLORS)

This function sets an images colormap and the number of colors associated with it. It is only valid for indexed color images.

void gimp_set_params (long size, void *data)

This function stores a size bytes data structure that can be recovered with `gimp_get_params ()` *See also:* `gimp_get_params ()`

int gimp_show_dialog (int DIALOG_ID)

This function starts the dialog with the handle `DIALOG_ID` and normally returns after the user pressed the Ok- or Cancel-Button. It should return *true* (¿0) if the dialog has ended with the Ok-Button. It should return *false* (0) if the dialog has ended with the Cancel-Button. *See also:* `gimp_close_dialog ()`

void gimp_show_item (int DIALOG_ID, int ITEM_ID)

This function shows the item `ITEM_ID` in the dialog `DIALOG_ID`.

void gimp_update_dialog (int)

This function redraws a dialog.

void gimp_update_image (Image IMAGE)

This function tells the GIMP to redraw an image. The image is redrawn in every view.

```
gimp_update_image (input);
```