Application Programming Using the GNOME Libraries

George Lebl
The GNOME Project

jirka@5z.com

Application Programming Using the GNOME Libraries

by George Lebl

In this tutorial, you will receive an overview of the GNOME libraries. You will learn how to speed up development of applications by using the many utility routines and objects available through the GNOME libraries, and how to make the GUI more consistent by using standard GNOME UI components. Focus will also be given to C applications using the GTK+ toolkit.

Table of Contents

List of Tables

List of Figures

Credits, Copyrights and Other Such Informations

All of the code given in this tutorial is under the GNU General Public License or the GNU Library General Public License. It was written by me or the rest of the Gnome core team.

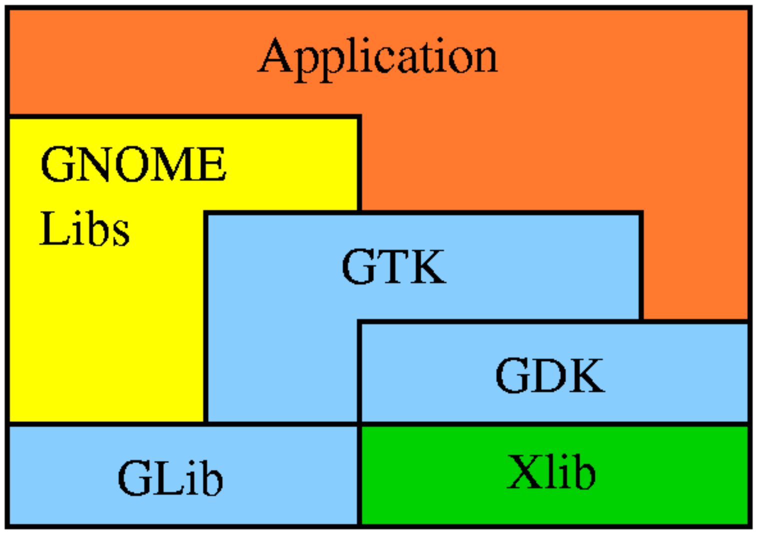
Also I would like to apologize if some of my English is not correct, as English is not my first language. I hope that my C is better then my English.

Chapter 1. GNOME Libraries Overview

1.1. Where Do GNOME Libraries Fit

Before going into the specifics of the GNOME libraries, it is important to see where do they fit in the picture of all the different libraries that are used in a GNOME application. The GNOME libraries are the most high level. GTK+ with it's two parts, GTK and GDK, comes next. GTK level provides an object model for C and a UI toolkit with the basic widgets to provide the generic basis for a GUI. GTK depends on GDK, which is a low-level wrapper around Xlib, the library directly talking to the X server. Everything (except for Xlib) depends on GLib which is a very useful C library with many utility and portability functions as well as a range of easy to use containers for C.

Figure 1-1. GNOME Application Library Hierarchy



1.2. Structure of GNOME Libraries

We now look at the structure of the GNOME libraries to see what they can offer. Here is a listing of the different libraries that are present in the gnome-libs package:

libgnome

Toolkit independent utility library

libgnomeui

Toolkit dependent library

libgnorba

Library for using ORBit corba implementation with gnome

gtk-xmhtml

xmHTML widget ported to gtk, used in the help browser

zvt

A very lean and mean terminal emulator widget

libvfs

A virtual file-system library used in Midnight Commander

libart_lgpl

A library used for nice anti-aliased graphics

We will not cover gtk-xmhtml, zvt, libvfs, libart_lgpl and libgnorba as they are mostly specialty libraries and some, notably the libvfs and gtk-xmhtml will most likely be phased out and replaced by better components.

We can see a clear division between the *libgnome* and *libgnomeui* libraries. The former is used in a toolkit independent fashion and could even be used for command line programs that never use X. The latter is the library which supplies the standard widgets and an application framework for applications written using *GTK+*. It is conceivable to write applications with other toolkits, but nobody as of yet has written a *libgnomeui* with a different toolkit, and I doubt it will happen soon, as GTK+ is a really great toolkit.

Chapter 2. GTK+ Programming

2.1. Overview

GTK+ is a C based toolkit for programming graphical applications in X windows. It is highly object oriented and has bindings to many popular languages, such as C++, Objective C, Perl, TOM, Guile, Python, etc ... GTK+ also uses GLib, which is a very useful C library, and includes things to help porting to different architectures, and containers such as a linked list or a hash. If you are already familiar with GTK+, you are now free to get bored.

2.2. GLib

2.2.1. Naming Conventions

GLib is a utility library which is heavily used in GTK+ and most of GNOME. GLib's functions are named starting with *g_* (such as *g_strdup*), GLib's typedefs for common types are just prefixed with a *g* (such as *gint32*), and GLib's structures are capitalized and start with *G* (such as *GHashTable*).

2.2.2. Typedefs

GLib provides some typedefs for portability, simplification of code, clarity of code and yet others just to keep consistent. The following table lists these typedefs. If the *Equivalent*, field is blank, there is no platform independent equivalent.

Table 2-1. GLib Typedefs

Name	Equivalent	Description
gint8		8bit wide signed integer
guint8		8bit wide unsigned integer
gint16		16bit wide signed integer
guint16		16bit wide unsigned integer
gint32		32bit wide signed integer
guint32		32bit wide unsigned integer

Name	**Equivalent**	**Description**
gint64		64bit wide signed integer (see note below)
guint64		64bit wide unsigned integer (see note below)
gchar	char	Standard character value
guchar	unsigned char	Standard unsigned character value
gshort	short	Standard short integer
gushort	unsigned short	Standard unsigned short integer
glong	long	Standard long integer
gulong	unsigned long	Standard unsigned long integer
gint	int	Standard integer
guint	unsigned int	Standard unsigned integer
gfloat	float	Standard float number type
gdouble	double	Standard float number type
gboolean	int	Type for storing TRUE/FALSE values
gpointer	void *	Type for storing pointers to arbitrary objects
gconstpointer	const void *	Type for storing pointers to arbitrary immutable objects

It should be noted that *gint64* and *guint64* might not be available on all platforms. You can check for this in your code by checking to see if the macro *G_HAVE_GINT64* is defined.

As you can see, some of the typedefs such as *gint* seem to have no other meaning in life then that of having a 'g' prefix and looking the same as the other typedefs. The logic behind this is to make the code look more consistent and clear. While it is no crime not to use these typedefs, you should really be consistent in your code. Some of the typedefs such as *gboolean* are only for improving code clarity and you could just as well use *int* to do exactly the same thing, but the former method clearly indicates that you are talking about a value that can only take TRUE or FALSE.

2.2.3. Portability and Utility Functions

There are some functions that have different implementations across different systems or are not extremely safe, or don't exist at all on some systems, so GLib provides it's own implementations or wrappers that have a constant behavior and usually check their arguments.

Here are some of the more useful functions that fit this category. Note that the prototype is more of an informative one, as some of these might be macros in reality.

Table 2-2. Few GLib Portability Functions

Prototype	Description
gchar * g_strdup (const gchar *)	Returns a newly allocated string which is a copy of the argument, if the argument is NULL, NULL is returned
gpointer g_malloc (int size)	Returns a newly region of memory with ’size’ bytes
void g_free (gpointer p)	Frees memory pointed to by ’p’, and only returns if ’p’ is NULL
gint g_snprintf (gchar *string, gulong n, gchar const *format, ...)	Works just like sprintf by printing the arguments according to the ’format’ into string, however it will only use ’n’ bytes of the string and will thus truncate the result if it needed more. It returns the number of bytes actually printed into ’string’
void g_usleep (gulong count)	Suspend execution for at least ’count’ microseconds
gint g_strcasecmp (const gchar *s1, const gchar *s2)	Compare string s1 and s2 in a case insensitive manner
gint g_strncasecmp (const gchar *s1, const gchar *s2, guint n)	Compare the first n characters of string s1 and s2 in a case insensitive manner

And there are also some utility functions and macros that are not really found in the normal c library. Here is a very short list of some of the more important and useful ones.

Table 2-3. Few GLib Utility Functions

Prototype	Description
g_new (type,count)	A macro which will allocate new memory for ’count’ items of type ’type’ and cast the result to ’type’. It is equivalent to ’(type) g_malloc(count * sizeof(type))’

Prototype	Description
g_new0 (type,count)	Same semantics as g_new, except that the returned memory will be set to all zeros. Note that you should not assume that setting the memory to zeros will zero out floating point types
gchar * g_strconcat (const gchar *str, ...)	When passed any number of arguments of the type (const char *) and a NULL after the last argument, it will return a newly allocated string that results by concatenation of all the arguments.
gchar * g_strdup_printf (const gchar *format, ...)	A printf like function that will return a newly allocated string with the result of the printf operation
gchar * g_strstrip (gchar *string)	Will strip leading and trailing whitespace from the string. It will not allocate new memory, but will modify the original string and return a pointer to it. If you wish to allocate new memory use a construction such as: 'string2 = g_strstrip(g_strdup(string1));'

There are many other useful methods in GLib, and I urge you to study GLib documentation and the GLib header file (*glib.h*), and you may be able to save a lot of time by not re-implementing some basic functionality.

2.2.4. Containers

Probably the best part of GLib are its containers. Here's a list of GLib's containers.

Table 2-4. Common GLib Containers

Name	Description
GList	Doubly linked list
GSList	Singly linked list
GHashTable	Hash table
GCache	Cache
GTree	Balanced binary tree
GNode	n-ary tree

Name	Description
GString	Dynamically sized string
GArray	Dynamically sized array
GPtrArray	Dynamically sized array of pointers
GByteArray	Dynamically sized array of bytes (guint8)

2.2.4.1. GList, Doubly Linked List

The easiest to use are *GList*'s. The basic *GList* structure is just a single node of the linked list and you can put your data into the data pointer in the *GList* structure. To store a linked list you just store a pointer to the first node of the list. Here is the list of functions that operate on a GList. The functions usually take in a pointer and return the new pointer of the list, since the first node could now be a different one.

Table 2-5. Most Important GList Functions

Prototype	Description
GList* g_list_append (GList *list, gpointer data)	Append 'data' to a list. 'list' can be NULL to make a new list.
GList* g_list_prepend (GList *list, gpointer data)	Prepend 'data' to a list. 'list' can be NULL to make a new list.
GList* g_list_remove (GList *list, gpointer data)	Remove the node containing 'data' from the list.
GList* g_list_find (GList *list, gpointer data)	Find the GList node that contains the 'data'.
GList* g_list_next (GList *list)	A macro that returns a pointer to the next node.
GList* g_list_previous (GList *list)	A macro that returns a pointer to the previous node.
void g_list_free(GList *list)	Free the entire list.

To access the data from a particular *GList* node You look at the *data* member in the *GList* structure. So code that would create a linked list of two elements which are strdup'ed strings, and later free that list and the strings would look like:

```
GList *list = NULL; /*the actual list pointer*/
GList *li; /*just a temporary pointer to a node used for iterating
             over the list*/
...
/*here we add two strings to the list*/
```

```
list = g_list_append(list,g_strdup("String 1"));
list = g_list_append(list,g_strdup("String 2"));
...
/*here we loop though the list, freeing all the strings and then
  we free the list itself*/
for(li = list; li!= NULL; li = g_list_next(li)) {
        char *string = li->data;
        g_free(string);
}
g_list_free(list);
```

2.2.4.2. GString, Dynamically Sized String Type

Another simple to use and useful container is the *GString* container. It's a dynamically sized string container for the times when you don't know how large the string you will need will be. Here's a list of the most important functions.

Table 2-6. Most Important GString Functions

Prototype	Description
GString* g_string_new (const gchar *init)	Create a new GString with initial value of 'init'
void g_string_free (GString *string, int free_segment)	Free the GString structure and optionally also the string data segment
GString* g_string_append (GString *string, const gchar *val)	Append 'val' to 'string'
GString* g_string_prepend (GString *string, const gchar *val)	Prepend 'val' to 'string'
void g_string_sprintf (GString *string, const gchar *format, ...)	A sprintf like function for GString
void g_string_sprintfa (GString *string, const gchar *format, ...)	A sprintf like function for GString, but appends the string instead of overwriting it

To access the string data for use as a *char **, just access the *str* element of the *GString* structure. You can actually free the *GString* structure without freeing this data segment. This is useful if you want to create a normal C string. The following example is a function that takes an array of integers and sprintfs them into a string and returns a *char **.

```
char *
create_number_list(int array[], int array_len)
{
        int i;          /* the array iterator */
        GString *string; /* the GString */
        char *ret;      /* the return value */

        /* make a new GString that is empty */
        string = g_string_new("");

        /* iterate over the integer array */
        for(i=0; i<array_len; i++) {
                /* append the number to the string in parenthesis */
                g_string_sprintfa(string, "(%d)", array[i]);
        }

        /* setup the return value */
        ret = string->str;

        /* free the GString structure, but not the data */
        g_string_free(string,FALSE);

        /* return the string */
        return ret;
}
```

2.2.4.3. GHashTable

Though less often used then GList's and GString's. The hash table container is a very useful one. Usually by a hash table one would mean an object (in GLib's terms a *gpointer*) would have a string key, by which we could recall the object at a later time. GLib takes this a step further, making the key a *gpointer* as well, and letting you provide a hashing and a comparison function yourself. While this makes *GHashTable* much more flexible, it can lead to some confusion with respect to memory allocation for the keys. Let's give some important functions and deal with the details later:

Table 2-7. Most Important GHashTable Functions

Prototype	Description
GHashTable* g_hash_table_new (GHashFunc hash_func, GCompareFunc key_compare_func)	Creates a new hash table using the specified hash function and comparison function
void g_hash_table_destroy (GHashTable *hash_table)	Destroy the hash table and free memory. This does not however free neither the data, nor the keys, you have to do this yourself

Prototype	Description
void g_hash_table_insert (GHashTable *hash_table, gpointer key, gpointer value)	Insert a new 'value' with a key of 'key'.
void g_hash_table_remove (GHashTable *hash_table, gconstpointer key)	Remove the value with the key of 'key' from the table. Doesn't free neither the key nor the value.
gpointer g_hash_table_lookup (GHashTable *hash_table, gconstpointer key)	Fetch the pointer of the value, with the key of 'key'. Returns NULL if it isn't found
gboolean g_hash_table_lookup_extended (GHashTable *hash_table, gconstpointer lookup_key, gpointer *orig_key, gpointer *value)	Lookup the data with the key of 'value_key', store the original key pointer in 'orig_key' and the value in 'value'. Returns TRUE if the lookup was successful else it returns FALSE. You should use this function when removing an item to get rid of the original key in memory.
void g_hash_table_foreach (GHashTable *hash_table, GHFunc func, gpointer user_data)	Run a function for each data stored in the hash table. The 'user_data' will be passed to the function as the last argument. The GHFunc prototype follows.
void (*GHFunc) (gpointer key, gpointer value, gpointer user_data)	This is the function prototype that you will use for the function that is passed to g_hash_table_foreach. It gets passed the key, the value and the user_data specified in the g_hash_table_foreach call.
guint g_str_hash (gconstpointer v)	A standard string hash function for string hash tables
gint g_str_equal (gconstpointer v, gconstpointer v2)	A standard string compare function for string hash tables

To create a hash table, you pass the hash and key compare functions to *g_hash_table_new*. There are standard functions defined for strings (*g_str_hash* and *g_str_equal*) and others. However if you pass NULL as the hash and compare functions, you will get a direct pointer hash, where pointers will be actually themselves used as keys.

The problem of memory allocation becomes apparent when we start using string hashes. *GHashTable* doesn't store the string, all it stores is a pointer. Therefore, when inserting a value into the hash, you have to create a new copy of the key for that value. This is an important thing to remember as otherwise things are not going to behave really nice for you. The other problem is how to then get rid of the key. If you do a *g_hash_table_remove*, you give as a key a string with the same contents as the original key, but not the same memory location. After then a pointer to the original key would be lost and unless you stored a pointer to it somewhere, you just created a memory leak. What you need to do instead is to do a *g_hash_table_lookup_extended* first to get both the value and the original key pointer and then do the *g_hash_table_remove*.

The following example will make a new string hash, insert a couple of strings into it, retrieve them, and then destroy the hash and the values stored in it:

```
/* function we use for freeing the keys and data in the hash before
   we destroy the hash */
static void
free_key_value(gpointer key, gpointer value, gpointer user_data)
{
        g_free(key);
        g_free(value);
}

...

/* somewhere else in the code */

GHashTable *ht;

/* create a new hash table with strings as keys */
ht = g_hash_table_new(g_str_hash, g_str_equal);

/* insert a couple of strings (say colors keyed by shape) */
g_hash_table_insert(ht, g_strdup("triangle"), g_strdup("green"));
g_hash_table_insert(ht, g_strdup("square"), g_strdup("red"));
g_hash_table_insert(ht, g_strdup("circle"), g_strdup("blue"));

/* again, somewhere else in the code */
...
/* now here we wish to print out the color of a square */
char *color;

/* get the color of a square */
color = g_hash_table_lookup(ht, "square");

printf("The color of a square is: %s\n",color);

/* yet again somewhere else */
...
/* Now here we just want to destroy the hash table and free all the
 * memory associated with it. We use the free_key_value function and
 * have it run over all the values in the hash table. */
g_hash_foreach(ht, free_key_value, NULL);

/* now we can destroy the actual hash table */
```

```
g_hash_table_destroy(ht);
```

2.2.5. More GLib Information

For more information look at the *glib.h* header file and at the documentation on the www.gtk.org (http://www.gtk.org/) web site.

2.3. GTK+

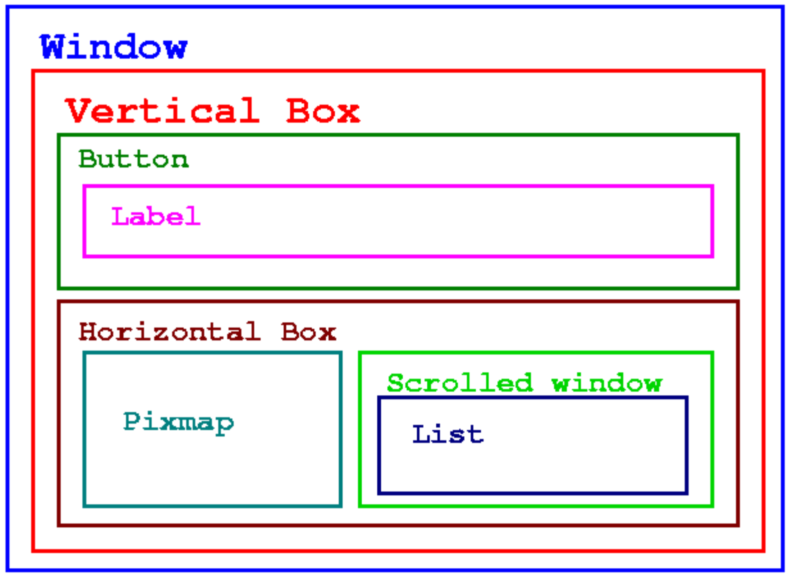
2.3.1. GUI Basics

Writing a GTK+ based GUI application is in essence extremely simple, and we'll use the Hello World example from Ian Main's excellent GTK+ tutorial, which is a very useful guide to writing gnome applications. But first we'll talk about the basic philosophy behind GTK+.

GTK+ is a container based toolkit, meaning you don't specify where the widget is, but you specify in what container it is. Some widgets, such as a window or a frame or a button, are containers that hold only one other widget. For example a button with a label is actually a button into which we added a label widget. If you need to put more widgets into that container, you will need to add another container into them, one that holds more then one widget such as a horizontal box.

In fact most layout of windows is usually done with containers such as horizontal boxes, vertical boxes and tables, those are the most important to learn. A horizontal box is a widget that you can add several widgets into and they will be added in a horizontal row. The height of the horizontal box is the height of the highest widget added, and the length is the length of all widgets combined. Vertical box behaves exactly the same, except that it's vertical instead of horizontal. A table can take in widgets at different rows and columns.

Figure 2-1. Example Window Hierarchy



2.3.2. GTK+ Object Model

Gtk's object model is an object oriented framework for C. It includes singular object inheritance, virtual methods, signals, runtime object modification, runtime type checking, and other goodies. While writing a GTK+ object is more involved then say writing an object in something like Java, it does have many advantages. GTK+ is an object model which doesn't require inheritance for most things you do with objects. For one, since methods are just functions that take the pointer to the object as the first argument, it's easy to write more methods in your own code, which are missing in the original object.

Objects in GTK+ are just C structures and inheritance is done by just including the parent structure as the first item in the child structure. This means, we can cast beween types with no problem. In addition to the object structure which is a specific instance, there is also a class structure for each class which contains things like pointers to virtual functionsand default signal handlers.

2.3.3. GTK+ Method Types

GTK+ object model uses 3 types of methods, a method can be a simple C function which just happens to take the pointer to an object instance as the first argument. This method type is the fastest in execution time, but cannot be overriden. The second type of a method is a virtual method. A virtual method is a function pointer int he class structure, with usually a wrapper function that takes care of calling the virtual method. This type of a method can be overriden in derived objects, but that's where it's advantages stop. The third and most flexible (and slowest to call) is the signal method. The signal is sort of like a virtual method, but the user can connect handlers to be run before or after the default method body. Sometimes objects have a singal method for the sole purpose of having users connect handlers to it andthus has the body of the method empty.

2.3.4. Data on Objects

There is a way to store arbitrary named data in objects to extend the object. This is done with the method, *gtk_object_set_data* (or *gtk_object_set_user_data* for a single unnamed pointer). To retrieve data, one uses *gtk_object_get_data*. Example:

```
GtkObject *obj;
void *some_pointer;
...
/*here we set "some_data" data on obj to point to some_pointer*/
gtk_object_set_data(obj,"some_data",some_pointer);
...
/*retrieve pointer to some_data from obj and store it in
  some_pointer*/
some_pointer = gtk_object_get_data(obj,"some_data");
```

The pointer can be a pointer to anything since it's manipulated as a (void *).

2.3.5. GTK+/GNOME Naming Conventions

Both GTK+ and GNOME use the same naming convention when naming objects and functions. GTK+ uses a prefix of *gtk_* for functions, and *Gtk* for objects, and GNOME uses *gnome_* and *Gnome*. When a function is a method for an object, the name (lower case) is appended to the prefix. For example the button object is named *GtkButton* (that is the name of the C struct holding the data for the object), and say the "*new*" method for *GtkButton* is then called *gtk_button_new*. Macros associated with objects use the same naming convention as functions, but are all capitalized. For example a macro that casts an object to a *GtkButton* is called *GTK_BUTTON*. There are exceptions, notably the type checking macro, which is called *GTK_IS_BUTTON* for *GtkButton*.

2.3.6. Using GTK+ Methods

Since GTK+ is object oriented, it uses inheritance for it’s widgets. For example *GtkHBox* and *GtkVBox* are derived from *GtkBox*. And thus you can use any *GtkBox* method on a *GtkVBox* or *GtkHBox*. However you need to cast the *GtkVBox* object to *GtkBox* before you call the function. This could be done with standard C casts such as:

```
GtkVBox *vbox;
...
gtk_box_pack_start((GtkBox *)vbox, ...);
...
```

This would work, however it is unsafe. GTK+ provides a mechanism of checking the types, so that it can warn you if you are casting an object which does not derive from the object you are casting to, or if you try to cast a NULL pointer. The macro is all capital name of the widget. For example the above code snippet would be

```
GtkVBox *vbox;
...
gtk_box_pack_start(GTK_BOX(vbox), ...);
...
```

GNOME uses the exact same form so anything you learn about GTK+ can be used for GNOME widgets, you just replace the GTK prefix with GNOME.

2.3.7. Example Hello World Program

Here is the promised example code for the hello world program. It doesn’t use any advanced containers, just a window and a button onto which a label is added. It illustrates the basic workings of a GUI program written in GTK+. Don’t be scared by it’s size, it’s mostly comments.

```
/* example-start helloworld helloworld.c */

#include <gtk/gtk.h>

/* this is a callback function. the data arguments are ignored in
 * this example.. More on callbacks below. */
void
hello (GtkWidget *widget, gpointer data)
{
        g_print ("Hello World\n");
```

```
}

gint
delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
        g_print ("delete event occurred\n");
        /* if you return FALSE in the "delete_event" signal
         * handler, GTK will emit the "destroy" signal.
         * Returning TRUE means you don’t want the window
         * to be destroyed. This is useful for popping up
         * ’are you sure you want to quit ?’ type dialogs. */

        /* Change TRUE to FALSE and the main window will
         * be destroyed with a "delete_event". */

        return (TRUE);
}

/* another callback */
void
destroy (GtkWidget *widget, gpointer data)
{
        gtk_main_quit ();
}

int
main (int argc, char *argv[])
{
        /* GtkWidget is the storage type for widgets */
        GtkWidget *window;
        GtkWidget *button;

        /* this is called in all GTK applications.
         * arguments are parsed from the command line and
         * are returned to the application. */
        gtk_init (&argc, &argv);

        /* create a new window */
        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

        /* when the window is given the "delete_event" signal
         * (this is given by the window manager, usually by
         * the ’close’ option, or on the titlebar), we ask
         * it to call the delete_event () function as defined
         * above.  The data passed to the callback function
         * is NULL and is ignored in the callback function. */
```

```
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (delete_event), NULL);

/* here we connect the "destroy" event to a signal
 * handler. This event occurs when we call
 * gtk_widget_destroy() on the window, or if we
 * return ’FALSE’ in the "delete_event" callback. */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
                    GTK_SIGNAL_FUNC (destroy), NULL);

/* sets the border width of the window. */
gtk_container_border_width (GTK_CONTAINER (window), 10);

/* creates a new button with the label "Hello World". */
button = gtk_button_new_with_label ("Hello World");

/* When the button receives the "clicked" signal, it
 * will call the function hello() passing it NULL as
 * it’s argument.  The hello() function is defined
 * above. */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (hello), NULL);

/* This will cause the window to be destroyed by
 * calling gtk_widget_destroy(window) when "clicked".
 * Again, the destroy signal could come from here,
 * or the window manager. */
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (window));

/* this packs the button into the window
 * (a gtk container). */
gtk_container_add (GTK_CONTAINER (window), button);

/* the final step is to display this newly created
 * widget... */
gtk_widget_show (button);

/* and the window */
gtk_widget_show (window);

/* all GTK applications must have a gtk_main().
 * Control ends here and waits for an event to occur
 * (like a key press or mouse event). */
gtk_main ();
```

```
        return 0;
}
/* example-end */
```

For more information look at the header files in <prefix>/include/gtk/ and <prefix>/include/gdk/ and at the documentation on the www.gtk.org (http://www.gtk.org/) web site.

Chapter 3. GNOME Programming

3.1. Introduction

3.1.1. What Is a GNOME Program

A GNOME program is a GTK+ GUI application, which makes use of the GNOME libraries. The GNOME libraries make it possible to have similar look and feel among applications, and to make simple things, simple to program. Plus the GNOME libraries add a whole bunch of widgets that simply don’t fit into GTK+.

3.1.2. Very Basic GNOME Program

The following program creates a basic gnome window and adds a horizontal box into which it packs two buttons, which (when pressed) print a string onto the stdout of the terminal you started the application from. The semantics and structure of a GNOME program is very similar to a pure GTK+ program.

You will need to compile this code, so you should look at the chapter on building gnome apps for more info how to build it. Here is a very simple Makefile that will compile a buttons.c for you. To compile other examples, just replace the buttons with the appropriate name.

```
CFLAGS=-g -Wall ‘gnome-config -cflags gnome gnomeui‘
LDFLAGS=‘gnome-config -libs gnome gnomeui‘
all: buttons
```

And here is buttons.c:

```
/*
 * A simple Gnome program, outside of GNOME tree, not using i18n
 * buttons.c
 */
/* the very basic gnome include */
#include <gnome.h>

/* a callback for the buttons */
static void
button_clicked(GtkWidget *button, gpointer data)
{
        /* the string to print is passed though the data field
           (which is a void *) */
        char *string = data;
        /* print a string on the standard output */
        g_print(string);
}

/* called when the user closes the window */
static gint
delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
        /* signal the main loop to quit */
        gtk_main_quit();
        /* return FALSE to continue closing the window */
        return FALSE;
}

int
main(int argc, char *argv[])
{
        GtkWidget *app;
        GtkWidget *button;
        GtkWidget *hbox;

        /* Initialize GNOME, this is very similar to gtk_init */
        gnome_init ("buttons-basic-example", "0.1", argc, argv);

        /* Create a Gnome app widget, which sets up a basic window
           for your application */
        app = gnome_app_new ("buttons-basic-example",
                             "Basic GNOME Application");

        /* bind "delete_event", which is the event we get when
           the user closes the window with the window manager,
           to gtk_main_quit, which is a function that causes
           the gtk_main loop to exit, and consequently to quit
           the application */
        gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                            GTK_SIGNAL_FUNC (delete_event),
                            NULL);

        /* create a horizontal box for the buttons and add it
           into the app widget */
        hbox = gtk_hbox_new (FALSE,5);
        gnome_app_set_contents (GNOME_APP (app), hbox);
        /* make a button and add it into the horizontal box,
           and bind the clicked event to call button_clicked */
        button = gtk_button_new_with_label("Button 1");
        gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (button_clicked),
                            "Button 1\n");

        /* and another button */
        button = gtk_button_new_with_label("Button 2");
        gtk_box_pack_start (GTK_BOX(hbox), button, FALSE, FALSE, 0);
        gtk_signal_connect (GTK_OBJECT (button), "clicked",
                            GTK_SIGNAL_FUNC (button_clicked),
                            "Button 2\n");

        /* show everything inside this app widget and the app
           widget itself */
        gtk_widget_show_all(app);

        /* enter the main loop */
        gtk_main ();

        return 0;
}
```

Please note the use of *gnome_init* instead of *gtk_init*, and *GnomeApp* widget instead of just a regular *GtkWindow*. We will go into detail of these later.

3.2. Overview

OK now we look at what the different libraries that we are going to cover do. First we look at the *libgnome* library and it's functionality, then we'll cover *libgnomeui* to complete all the basic functionality of a gnome program. We will take a look at the *gnome-canvas* in detail, as it is extremely powerful and useful widget. We'll also say a few things about drag and drop.

3.3. Using the libgnome Library

The *libgnome* library is the non-toolkit specific utility library for GNOME applications, and includes things like configuration file reading, .desktop file handling, special GLib like utility routines, getting the standard file locations for GNOME, handling mime types, handling meta-data on files, sound, "triggers", and other useful things one could want to use. Also say that you are writing an application in say motif, but you want your app to be more GNOME friendly. Then you could use this library to make your application work well with other GNOME programs.

3.3.1. Configuration Files

The *gnome-config* routines provide an easy way to store configuration info in files. To see a full list of the routines, look in the *libgnome/gnome-config.h* header file.

The routines all working with a path. The path is a Unix like path, but the root is set to the *~/.gnome/* directory. So */some/config/path/file/sectionA/keyB*, refers to the file *~/.gnome/some/config/path/file*, and inside the file using section *sectionA* and key *keyB*.

3.3.1.1. Reading Configuration Info

To read configuration information *gnome_config_get_** functions are used. the * is replaced by the type of the data, it can be *int*, *float*, *string*, *bool* and *vector*. The *int* functions work with *gint*, *float* functions work with *gdouble*, *string* functions work with *gchar **, *bool* functions work with *gboolean* and *vector* work with an argc like array of strings (*gint* and *gchar ***). For the *gnome_config_get_** functions, the default to be returned if the file section or key are not found can be appended to the path after an equals sign. If you need to know if the default was used, you can append *_with_default* to the function name and add a parameter which is a *gboolean **, though which the function returns whether it used the default or if it actually found a real value. Example follows:

```
int counter;
char *text;
gboolean def;
...
counter = gnome_config_get_int_with_default("/example/section/counter=1",
                                            &def);
if(def) g_print("Default used for counter!\n");
text = gnome_config_get_string("/example/section/text=TEXT");
...
g_free(text);
```

Note that the string returned by *gnome_config_get_string* should be freed with *g_free*, the vector from *gnome_config_get_vector* should also be freed with *g_free*.

3.3.1.2. Writing Configuration Info

To write configuration info to files, the *gnome_config_set_** functions are used. The use is very similar to above to the *gnome_config_get_** functions. The types used are exactly the same. Except with the "set" functions, you pass the data you want to store after the path, and there is no default inside the path. If the directory in the path doesn't exist it will get created when the functions are written to disk. After you set all your data, you need to call *gnome_config_sync* to actually write your data to file. The library will not write the data to file immediately for efficiency reasons. Example follows:

```
char *text;
int counter;
...
/*after we have set text and counter to some values we can
  write them to our config location*/
gnome_config_set_int("/example/section/counter",counter);
gnome_config_set_string("/example/section/text",text);
gnome_config_sync();
```

3.3.1.3. Privacy Functions

If you want to store sensitive data, that other users should not read, use the *gnome_config_private_** functions, which have exactly the same behavior as the above functions, with the exception of *gnome_config_sync* (and a few others) which doesn't have a private equivalent since it works for both. The difference is that these functions write to a directory called *~/.gnome_private* on which 0700 permissions are enforced. This is not extremely secure, but because of the highly brain-dead US export restrictions, we can't really use encryption.

3.3.1.4. Using gnome-config for Arbitrary Files

If you wish to use *gnome-config* for reading and writing of arbitrary files on the file-system (as long as those files are in the *gnome-config* format), you can just prepend '=' to the beginning of the path and another '=' to the end of the file name. Example follows:

```
char buf[256];
...
/*write some bogus data to a temporary file*/
g_snprintf(buf,256,"=%s=/section/key",tmpnam(tmpnam));
gnome_config_set_int(buf,999);
gnome_config_sync();
```

Note that it doesn't really make sense to use the private versions when using an arbitrary absolute path, as there will be absolutely no difference between the two.

3.3.1.5. Automatic Prefixes

Sometime, especially if you have a long path, would be much easier to say have the config automatically prefix the path with a given string. This is what *gnome_config_push_prefix* and *gnome_config_pop_prefix* are for. You pass the string you want to prefix to *gnome_config_push_prefix* and call *gnome_config_pop_prefix* when you are done. Note that these functions are common between private and normal config functions. Example:

```
gnome_config_push_prefix("/file/section/");
gnome_config_set_int("key1",1);
gnome_config_set_int("key2",2);
gnome_config_set_int("key3",-88);
gnome_config_pop_prefix();
```

3.3.1.6. Misc gnome-config Stuff

If you need to remove a file in your configuration file, you will use *gnome_config_clean_file*. This function will schedule that file to be deleted on the next *gnome_config_sync*. You can do a *gnome_config_clean_file* and then use the file and then do *gnome_config_sync*, and it will have the expected behavior.

If you have written to a file or read from a file and want *gnome-config* to drop it from memory, use *gnome_config_drop_file*. This is used if you want to forget changes done to that file, or to simply conserve memory, since *gnome-config* will otherwise keep a copy of the data in memory for faster access.

3.3.2. .desktop Files

The *.desktop* files are the files that contain information about programs. The files are in the *gnome-config* format and are internally read using *gnome-config*. Your app definitely needs one of these files installed in the system menu paths if it wants to be added to the menu.

You can use *gnome_desktop_entry_** functions to manipulate these files. These functions work with a structure called *GnomeDesktopEntry* and you should look at the *libgnome/gnome-dentry.h* header file for the format of this structure.

The basic functions that you use to manipulate these files are *gnome_desktop_entry_load* which returns a newly allocated *GnomeDesktopEntry* structure, *gnome_desktop_entry_launch* which takes the *GnomeDesktopEntry* structure as an argument and launches the program it describes and *gnome_desktop_entry_free* which frees the allocated memory with the structure.

An example .desktop file for your app might look like:

```
[Desktop Entry]
Name=Clock
Name[cz]=Hodiny
Comment=Digital Clock
Comment[cz]=Digitalni Hodiny
Exec=digital-clock
Icon=clock.png
Terminal=0
Type=Application
```

You will notice that there are translations for Name and Comment fields in Czech. For gnome programs to notice your .desktop file, it is usually located somewhere under *<prefix>/share/apps/*, which contains the hierarchy of the system menus. For the system to find your icon, your icon should be placed inside the *<prefix>/share/pixmaps* directory. Note that the prefix refers to the location where GNOME was installed.

3.3.3. Utility and Files

3.3.3.1. Files

There is a standard way to find files that belong to gnome installation, you shouldn’t really be using your own logic to find them and you should use these functions to get filenames for icons, sounds or other data. Also these functions are only for finding files that were installed with the GNOME libraries. There is not at this time functions to deal with data installed by your application. The functions are:

Table 3-1. File Finding Functions

Prototype	Description
char *gnome_libdir_file (const char *filename)	Get a full path of a file in the library directory or NULL if the file doesn’t exist
char *gnome_unconditional_libdir_file (const char *filename)	Get a full path of a file in the library directory
char *gnome_datadir_file (const char *filename)	Get a full path of a file in the data director or NULL if the file doesn’t exist
char *gnome_unconditional_datadir_file (const char *filename)	Get a full path of a file in the data director
char *gnome_sound_file (const char *filename)	Get a full path of a file in the sound directory or NULL if the file doesn’t exist

Prototype	Description
char *gnome_unconditional_sound_file (const char *filename)	Get a full path of a file in the sound directory
char *gnome_pixmap_file (const char *filename)	Get a full path of a file in the pixmap directory or NULL if the file doesn’t exist
char *gnome_unconditional_pixmap_file (const char *filename)	Get a full path of a file in the pixmap directory
char *gnome_config_file (const char *filename)	Get a full path of a file in the config directory or NULL if the file doesn’t exist
char *gnome_unconditional_config_file (const char *filename)	Get a full path of a file in the config directory

These functions return a newly *g_malloc*ed string and you should use *g_free* on the string when you are done. The *gnome_unconditional_** functions don’t check if the file actually exist and will just return a file name. The normal functions will check and return *NULL* if the file doesn’t exist. So you shouldn’t use those functions when you will do saving. As an example we want to get a pixmap from the standard pixmap directory, for example we need to get the "gnome-help.png" icon:

```
char *name;
...
name = gnome_pixmap_file("gnome-help.png");
if(!name) {
        g_warning("gnome-help.png doesn’t exist!");
} else {
        /*here we use the file*/
        ...
        g_free(name);
}
```

Also of interest are the functions (actually macros) *gnome_util_home_file* and *gnome_util_user_home*. *gnome_util_home_file* takes one argument (string) and returns a newly allocated string with the home directory and .gnome prepended to the file. So for example if you pass it say *someconfigfile*, it would return */home/jirka/.gnome/someconfigfile*. Similar is the *gnome_util_user_home*, it takes one argument and returns the file with just the home directory added. So if you pass it *.dotfile*, it would return */home/jirka/.dotfile*.

3.3.3.2. Utility

There are also a number of GLib like named functions to make your life easier, of note would be *g_file_exists* which takes a filename and returns *TRUE* if it exists or *FALSE* if it doesn’t, or *g_concat_dir_and_file* which takes a directory name and a file name, and takes care of the *'/'* issue, this is useful when working with strings where you don't want to check for the '/', you just want to append a directory to some file, or another directory. Note that you should *g_free* the string you get as usual. For more utility functions, look into *libgnome/gnome-util.h*, it is well commented.

3.3.4. Mime Types

Sometimes it's useful to know the mime-type of a file. You can do this by using the *gnome_mime_type_or_default* function, which takes two arguments, the filename and a default mime-type string which it will return if it can't figure out the mime type from the filename. This call doesn't actually look into the file, it tries to guess the type by looking at the filename itself. Also the string it returns is a pointer to it's internal database and you should not free it as that would likely result in a segfault later on. You can also use *gnome_mime_type* which will return NULL if it can't guess the mime-type.

It is also possible to work with URI lists, such as the ones used sometimes in Drag and Drop. Usually from an URI list you want to extract a list of filenames that you just received. For that you use the *gnome_uri_list_extract_filenames* function, which takes the URI list as a string argument, and returns a *GList ** of newly allocated strings. Once you are done with the files, you should free the strings and the list. You can use the utility routine *gnome_uri_list_free_strings* to do this for you.

In the following example I write a drag and drop handler that takes the files and finds out their mime information, then you could just write code that can do things based on the mime type of the files.

```
/*this is the handler for the drag_data_receive signal, assuming our
  widget only accepts the "text/uri-list" mime type of data, drag and
  drop is a more complicated topic and you should read up on GTK+
  documentation for better treatment*/
static void
dnd_drop_internal (GtkWidget        *widget,
                   GdkDragContext   *context,
                   gint              x,
                   gint              y,
                   GtkSelectionData *selection_data,
                   guint             info,
                   guint             time)
{
        GList *files, *li;
        /*here we extract the filenames from the URI-list we received*/
        files = gnome_uri_list_extract_filenames(selection_data->data);

        /*here we loop though the files and get their mime-type*/
        for(li = files; li!=NULL ; li = g_list_next(li)) {
                  char *mimetype;
                  char *filename = li->data;

                  /*guess the mime type of the file*/
                  mimetype = gnome_mime_type(filename);

                  /*if we can’t guess it, just loop to the
                    next filename*/
                  if(!mimetype) continue;

                  /*here comes code that can actually do something
                    based on the mime-type of the file we received*/
                  ...
        }
        /*free the list of files we got*/
        gnome_uri_list_free_strings (files);
}
```

Note how easy it is to find out what files you got, and what type they are. You would just need to add some code instead of the three dots that actually compares the mime strings you got to some you have to figure out what you can do with the files.

3.3.5. Meta Data

Sometimes it is useful to store some information along with a filename, this can be done easily with the *gnome-metadata*. It is a set of functions to manage this data. Since Unix doesn’t natively support meta-data, you have to help it yourself. For example if your app copies, renames or deletes files, use the following functions.

Table 3-2. Metadata Functions

Prototype	Description
int gnome_metadata_rename (const char *from, const char *to)	Notify the metadata database that a file has been renamed
int gnome_metadata_copy (const char *from, const char *to)	Notify the metadata database that a file has been copied

Prototype	Description
int gnome_metadata_delete (const char *file)	Notify the metadata database that a file has been deleted
int gnome_metadata_set (const char *file, const char *name, int size, const char *data)	Set data associated with the file ’file’, and key ’name’. The data is pointed to by ’data’ and is ’size’ bytes long. GNOME_METADATA_OK is returned on success.
int gnome_metadata_get (const char *file, const char *name, int *size, char **buffer)	Get data associated with file ’file’ and key ’name’. Data will be copied to a buffer and ’buffer’ will be set to point to it, and ’size’ will be set to the size of the buffer. GNOME_METADATA_OK is returned on success.
char **gnome_metadata_list (const char *file)	Get a list of the keys for which there is some data set on ’file’. The list will be a newly allocated, NULL terminated string vector and should be freed with g_strfreev

Table 3-3. Metadata Return Values

Name	Description
GNOME_METADATA_OK	No error (this is actually 0)
GNOME_METADATA_IO_ERROR	IO or other low-level communications/storage error.
GNOME_METADATA_NOT_FOUND	Information not found.

These functions don’t actually do the operations on the files, they just change the meta-data accordingly. So if your app does any of these operations, it is nicer towards other apps, that it notifies the meta-data database of the changes. You shouldn’t rely on the data being stored. Only non-critical data should be stored in the meta-data, since apps that do not notify the database with these functions will make you loose your data for the file. These functions will return 0 or *GNOME_METADATA_OK* if there was no error, or an error-code (described above).

Now if you actually want to use the meta-data to store information about files, you will most likely use the functions *gnome_metadata_set*, *gnome_metadata_remove* and *gnome_metadata_get*. Again these functions return an integer, which is *GNOME_METADATA_OK* in case there was no error, or they use the same error codes as the previous functions.

The functions work with a a key string for which they store a piece of data. The data is represented by a size integer and a character pointer. *gnome_metadata_set* takes the filename as the first argument, the name or key of the data as the second argument, the size as the third and the pointer to the actual data as the forth argument. This function just sets that data for that file and key. *gnome_metadata_remove* will clear a particular data item on a file, so it takes a file and then the key name as the second argument. *gnome_metadata_get* takes the filename as the first argument and the key name as the second, then it returns data size though an integer pointer you pass though the third argument and the actual data though a pointer to a pointer you pass as the fourth argument. The data returned is newly allocated and should be freed after use. Small example follows (in real life you should also check the return of the functions for errors):

```
int size;
char *data;
...
/*set some bogus data on a file*/
gnome_metadata_set("/some/file/name","bogus",5,"BLAH");
...
/*retrieve the data back*/
gnome_metadata_get("/some/file/name","bogus",&size,&data);
```

3.4. Using the GnomeApp Framework

The GnomeApp application framework is part of the libgnomeui library, but it really deserves sesparate treatment. This is the part which really makes a GNOME application a GNOME application. It also makes programming apps very simple and straightforward, and makes the apps featurefull, consistent and user customizable. With plain GTK+ you have to do a lot of things by yourself, reinventing the wheel every time, but GnomeApp takes care of the UI setup for you and still allows the user to configure that behavior and have it be consistent over different applications.

3.4.1. GnomeApp Overview

GnomeApp is the basic widget behind each app. It is the main window of the application, containing the document being worked on and the applications menus, tool-bars and status bars. It also remembers the docked positions of menu bars and tool-bars and such for you so that the user gets the window the way he left it when he left the application last time.

3.4.2. Creating a GnomeApp Window

Creating a new *GnomeApp* widget is as easy as calling *gnome_app_new* with the application name, which is usually the name of the executable or something else that is unique to your application and the title of the main window. Then you create the content of the main window and add it to the *GnomeApp* widget by calling *gnome_app_set_contents* with your contents as the argument.

Adding menu-bars, tool-bars and status-bars is equally easy, you call *gnome_app_set_toolbar*, *gnome_app_set_menus* or *gnome_app_set_statusbar*. *gnome_app_set_toolbar* is for simple applications that have only one tool-bar, for more complicated applications you need to use *gnome_app_add_toolbar*, which allows you to add as many docked tool-bars as you need.

3.4.3. Menu and Tool-bar Creation

3.4.3.1. Automatic Menu and Tool-bar Creation

Most of the time, you don’t really want to create your menu-bars and tool-bars by yourself. You can use functions from *libgnomeui/gnome-app-helper.h* to construct menus and tool-bars for you. All you need is to fill in a couple of structures with the your information, and call *gnome_app_create_menus* or *gnome_app_create_toolbar* with that structure and voila, your application has menus and tool-bars. Sometimes you wish to pass a data pointer to all the callbacks from those structures to work with, then you’d use the *gnome_app_create_toolbar_with_data* and *gnome_app_create_menus_with_data*, and pass an extra parameter which will be passed in the data field of the callbacks.

3.4.3.2. GnomeUIInfo Structure Definition

Here is the definition of the structure you need to fill (actually you fill in an array of such structures). Also note I included the enums that you will need to fill that structure.

```
/* These values identify the type of pixmap used in an item */
typedef enum {
        GNOME_APP_PIXMAP_NONE,          /* No pixmap specified */
        GNOME_APP_PIXMAP_STOCK,         /* Use a stock pixmap
                                           (GnomeStock) */
        GNOME_APP_PIXMAP_DATA,          /* Use a pixmap from inline
                                           xpm data */
        GNOME_APP_PIXMAP_FILENAME       /* Use a pixmap from the
                                           specified filename */
} GnomeUIPixmapType;
```

```
/* This is the structure that defines an item in a menu bar
 * or tool-bar.  The idea is to create an array of such
 * structures with the information needed to create menus or
 * tool-bars.  The most convenient way to create such a structure
 * is to use the GNOMEUIINFO_* macros provided below. */
typedef struct {
        GnomeUIInfoType type;     /* Type of item */
        gchar *label;             /* String to use in the label */
        gchar *hint;              /* For tool-bar items, the
                                     tool-tip. For menu items, the
                                     status bar message */
        gpointer moreinfo;        /* For an item, toggle-item, or
                                     radio-item, this is a pointer
                                     to the function to call when
                                     the item is activated. For
                                     a subtree, a pointer to
                                     another array of GnomeUIInfo
                                     structures. For a radio-item
                                     lead entry, a pointer to an
                                     array of GnomeUIInfo
                                     structures for the radio
                                     item group. For a help item,
                                     specifies the help node to
                                     load (i.e. the application’s
                                     identifier) or NULL for the
                                     main program’s name.  For
                                     builder data, points to the
                                     GnomeUIBuilderData structure
                                     for the following items */
        gpointer user_data;       /* Data pointer to pass to
                                     callbacks */
        gpointer unused_data;     /* Reserved for future expansion,
                                     should be NULL */
        GnomeUIPixmapType pixmap_type;  /* Type of pixmap for
                                           the item */
        gpointer pixmap_info;     /* Pointer to the pixmap
                                     information:

                                     For GNOME_APP_PIXMAP_STOCK, a
                                     pointer to the stock icon name.

                                     For GNOME_APP_PIXMAP_DATA, a
                                     pointer to the inline xpm data.

                                     For GNOME_APP_PIXMAP_FILENAME, a
```

```
                                   pointer to the filename
                                   string. */
        guint accelerator_key;    /* Accelerator key, or 0 for none */
        GdkModifierType ac_mods;  /* Mask of modifier keys for the
                                   accelerator */

        GtkWidget *widget;        /* Filled in by gnome_app_create*,
                                   you can use this to tweak the
                                   widgets once they have been
                                   created */
} GnomeUIInfo;
```

Don't worry if you don't know all the items or what they mean. If you don't know what it is, just leave it NULL or 0. However, most of the time, it's easiest to copy the menu's from another app and just modify them for your needs, that way you will also know much better what does what then by just looking at the structure.

3.4.3.3. Helper Macros

Most of the time, menu entries are very simple, so one can just use one of the simple macros provided. For example, for the end of a menu, one would use the *GNOMEUIINFO_END* macro, for a separator one uses the *GNOMEUIINFO_SEPARATOR* macro. Now for the actual items there are also macros, which require you to fill in less info. For example if you have an item that you provide an xpm format data for, you can use the *GNOMEUIINFO_ITEM(label, tooltip, callback, xpm_data)* macro, where label is the text of the label, tool-tip is the tool-tip that the user gets when he goes over that item (or it can be *NULL*), callback is the function that gets called when the user presses that item, and the xpm_data is a pointer to an xpm data you want to use as the icon. If you have no icon you can just use the *GNOMEUIINFO_ITEM_NONE(label, tooltip, callback)* macro. If what you are adding is a standard item for which there is a stock icon (we'll talk about those next), you can use the *GNOMEUIINFO_ITEM_STOCK(label, tooltip, callback, stock_id)* macro where the stock_id is the id of the stock icon you want to use. Then for your main menu bar, or to put sub-menus inside your menus, you can use *GNOMEUIINFO_SUBTREE(label, tree)* and *GNOMEUIINFO_SUBTREE_STOCK(label, tree, stock_id)*, where the tree is the array of *GnomeUIInfo* structures that you want to use as that sub-menu. There are a few other macros, but most of the time you will get by with just these macros, so you don't need to learn the entire structure of the *GnomeUIInfo*.

3.4.3.4. Standard Menu Item Macros

Just about all application contain a couple of standard menu items, so to keep things more consistent there are a bunch of macros that fill in everything for you except for the callback function and the data. The advantage of using the macros is consistency across applications, user customization, and translation.

3.4.3.4.1. Menu Items

Most of these macros have the form: *GNOMEUIINFO_MENU_<name>_ITEM (callback, data)*. However, there is an exception, the "New xxx" item. The GNOME style guide Requires that you put what the "New" thing is into the item name. Not to mention that it will have a different hint as well. So the "New xxx" item has the structure of: *GNOMEUIINFO_MENU_NEW_ITEM(label, hint, callback, data)*. The "label" should start with "New ". Also note that if you have more new items, you need to use the "New" subtree macro, which is explained later.

Table 3-4. The File Menu

Macro	Description
GNOMEUIINFO_MENU_NEW_ITEM(label, hint, cb, data)	"New" menu item (you need to provide label and hint yourself here)
GNOMEUIINFO_MENU_OPEN_ITEM(cb, data)	"Open" menu item
GNOMEUIINFO_MENU_SAVE_ITEM(cb, data)	"Save" menu item
GNOMEUI-INFO_MENU_SAVE_AS_ITEM(cb, data)	"Save as..." menu item
GNOMEUIINFO_MENU_REVERT_ITEM(cb, data)	"Revert" menu item
GNOMEUIINFO_MENU_PRINT_ITEM(cb, data)	"Print" menu item
GNOMEUI-INFO_MENU_PRINT_SETUP_ITEM(cb, data)	"Print Setup" menu item
GNOMEUIINFO_MENU_CLOSE_ITEM(cb, data)	"Close" menu item
GNOMEUIINFO_MENU_EXIT_ITEM(cb, data)	"Exit" menu item

Table 3-5. The Edit Menu

Macro	Description
GNOMEUIINFO_MENU_CUT_ITEM(cb, data)	"Cut" menu item
GNOMEUIINFO_MENU_COPY_ITEM(cb, data)	"Copy" menu item
GNOMEUIINFO_MENU_PASTE_ITEM(cb, data)	"Paste" menu item
GNOMEUIINFO_MENU_SELECT_ALL_ITEM(cb, data)	"Select" menu item
GNOMEUIINFO_MENU_CLEAR_ITEM(cb, data)	"Clear" menu item
GNOMEUIINFO_MENU_UNDO_ITEM(cb, data)	"Undo" menu item
GNOMEUIINFO_MENU_REDO_ITEM(cb, data)	"Redo" menu item
GNOMEUIINFO_MENU_FIND_ITEM(cb, data)	"Find" menu item
GNOMEUIINFO_MENU_FIND_AGAIN_ITEM(cb, data)	"Find Again" menu item
GNOMEUIINFO_MENU_REPLACE_ITEM(cb, data)	"Replace" menu item
GNOMEUIINFO_MENU_PROPERTIES_ITEM(cb, data)	"Properties" menu item, for properties of a manipulated object

Table 3-6. The Settings Menu

Macro	Description
GNOMEUIINFO_MENU_PREFERENCES_ITEM(cb, data)	"Preferences" menu item, for application preferences

Table 3-7. The Windows Menu

Macro	Description

Macro	Description
GNOMEUI-INFO_MENU_NEW_WINDOW_ITEM(cb, data)	"New window" menu item
GNOMEUI-INFO_MENU_CLOSE_WINDOW_ITEM(cb, data)	"Close window" menu item

Table 3-8. The Help Menu

Macro	Description
GNOMEUIINFO_MENU_ABOUT_ITEM(cb, data)	"About" menu item

Table 3-9. The Game Menu

Macro	Description
GNOMEUI-INFO_MENU_NEW_GAME_ITEM(cb, data)	"New game" menu item
GNOMEUI-INFO_MENU_PAUSE_GAME_ITEM(cb, data)	"Pause game" menu item
GNOMEUI-INFO_MENU_RESTART_GAME_ITEM(cb, data)	"Restart game" menu item
GNOMEUI-INFO_MENU_UNDO_MOVE_ITEM(cb, data)	"Undo move" menu item
GNOMEUI-INFO_MENU_REDO_MOVE_ITEM(cb, data)	"Redo move" menu item
GNOMEUIINFO_MENU_HINT_ITEM(cb, data)	"Hint" menu item
GNOMEUIINFO_MENU_SCORES_ITEM(cb, data)	"Scores" menu item
GNOMEUI-INFO_MENU_END_GAME_ITEM(cb, data)	"End game" menu item

3.4.3.4.2. Menu Trees and Subtrees

We have already mentioned a "New" subtree. For this you should use the *GNOMEUIINFO_MENU_NEW_SUBTREE (tree)* macro, where the tree argument is another GnomeUIInfo structure array of the different new items.

There are also the standard top level menus. Again you pass the array of GnomeUIInfo structures to the macro.

Table 3-10. The Toplevel Menu Macros

Macro	Description
GNOMEUIINFO_MENU_FILE_TREE (tree)	"File" menu
GNOMEUIINFO_MENU_EDIT_TREE (tree)	"Edit" menu
GNOMEUIINFO_MENU_VIEW_TREE (tree)	"View" menu
GNOMEUIINFO_MENU_SETTINGS_TREE (tree)	"Settings" menu
GNOMEUIINFO_MENU_FILES_TREE (tree)	"Files" menu
GNOMEUIINFO_MENU_WINDOWS_TREE (tree)	"Windows" menu
GNOMEUIINFO_MENU_HELP_TREE (tree)	"Help" menu
GNOMEUIINFO_MENU_GAME_TREE (tree)	"Game" menu

Sometimes you may want to refer to menu path of these menus, such as for adding items to a "Windows" menu. For this you should use the macros of the form *GNOME_MENU_<name>_STRING* and *GNOME_MENU_<name>_PATH*. These will expand to the appropriate string. The macro ending with *_STRING* will expand to just the menu name, and the macro ending with *_PATH* to the menu name followed by a "/". The <name> can be one of the following: FILE, EDIT, VIEW, SETTINGS, NEW, FILES or WINDOWS.

3.4.3.5. Help Menu

Your application should contain a help menu, the help menu can be defined as:

```
GNOMEUIINFO_HELP("app_name"),
GNOMEUIINFO_MENU_ABOUT_ITEM(callback, data),
GNOMEUIINFO_END
```

The GNOMEUIINFO_HELP macro takes the name of your application and expects the help files to be installed as per normal gnome procedures. *FIXME: we need to add some section on help files and stuff*

3.4.3.6. Example

Here is a very simple application that makes use of these:

```
/*
 * A simple Gnome program, outside of GNOME tree, not using i18n
 * uiinfo.c
 */
/* the very basic gnome include */
#include <gnome.h>

/* a callback for the buttons */
static void
a_callback(GtkWidget *button, gpointer data)
{
        /*just print a string so that we know we got there*/
        g_print("Inside Callback\n");
}

GnomeUIInfo file_menu[] = {
        GNOMEUIINFO_MENU_EXIT_ITEM(gtk_main_quit,NULL),
        GNOMEUIINFO_END
};

GnomeUIInfo some_menu[] = {
        GNOMEUIINFO_ITEM_NONE("_Menuitem","Just a menuitem",
                              a_callback),
        GNOMEUIINFO_SEPARATOR,
        GNOMEUIINFO_ITEM_NONE("M_enuitem2","Just a menuitem",
                              a_callback),
        GNOMEUIINFO_END
};

GnomeUIInfo menubar[] = {
        GNOMEUIINFO_MENU_FILE_TREE(file_menu),
        GNOMEUIINFO_SUBTREE("_Some menu",some_menu),
        GNOMEUIINFO_END
};

GnomeUIInfo toolbar[] = {
        GNOMEUIINFO_ITEM_STOCK("Exit","Exit the application",
                               gtk_main_quit,
                               GNOME_STOCK_PIXMAP_EXIT),
        GNOMEUIINFO_END
};

int
main(int argc, char *argv[])
{
        GtkWidget *app;
        GtkWidget *button;
        GtkWidget *hbox;
        GtkWidget *label;

        /* Initialize GNOME, this is very similar to gtk_init */
        gnome_init ("menu-basic-example", "0.1", argc, argv);

        /* Create a Gnome app widget, which sets up a basic
           window for your application */
        app = gnome_app_new ("menu-basic-example",
                             "Basic GNOME Application");

        /* bind "delete_event", which is the event we get when
           the user closes the window with the window manager,
           to gtk_main_quit, which is a function that causes
           the gtk_main loop to exit, and consequently to quit
           the application */
        gtk_signal_connect (GTK_OBJECT (app), "delete_event",
                            GTK_SIGNAL_FUNC (gtk_main_quit),
                            NULL);

        /*make a label as the contents*/
        label = gtk_label_new("BLAH BLAH BLAH BLAH BLAH");

        /*add the label as contents of the window*/
        gnome_app_set_contents (GNOME_APP (app), label);

        /*create the menus for the application*/
        gnome_app_create_menus (GNOME_APP (app), menubar);

        /*create the tool-bar for the application*/
        gnome_app_create_toolbar (GNOME_APP (app), toolbar);

        /* show everything inside this app widget and the app
           widget itself */
        gtk_widget_show_all(app);
        /* enter the main loop */
        gtk_main ();

        return 0;
}
```

Voila, an application with a menu and a tool-bar. As you see, adding extra menu items is just adding extra definitions to the GnomeUIInfo structure array.

3.4.3.7. Accelerator Keys

You have probably noticed the underlines in the labels for the menu items, these specify the accelerators for that menu. That's really all you need to do to add accelerators for menu items. The way accelerators work is very similar to the other windowing systems out there, *alt-<key>* if you are not browsing the menus or just the *<key>* if you have the menu open.

3.4.4. GnomeAppBar, The Status Bar

Every app should also include an application status bar on the bottom of the window. This status bar should give flashes and warnings about the application status, this is done according to the preferences as long as you use the GnomeApp message functions (described in Talking to the user section). The status bar also gives hints about menuitems and can provide a progress bar. Here are the important appbar functions.

Table 3-11. Important GnomeAppBar Methods

Prototype	Description
GtkWidget * gnome_appbar_new (gboolean has_progress, gboolean has_status, GnomePreferencesType interactivity)	Creates a new appbar widget, optionally with a progress bar if 'has_progress' is TRUE, and a statusbar if 'has_status' is true. The 'interactivity' tells us if when we want to be interactive, it can be never, always, or depending on user preferences, GNOME_PREFERENCES_USER, and that's what should probably be used. Though note that an interactive status bar is not finished yet.
void gnome_appbar_set_status (GnomeAppBar * appbar, const gchar * status)	Set the text of the status bar, without changing AppBar state.

Prototype	**Description**
void gnome_appbar_set_default (GnomeAppBar * appbar, const gchar * default_status)	Set tthe default text of the statusbar when there is nothing to display
void gnome_appbar_push (GnomeAppBar * appbar, const gchar * status)	Push a message onto the appbar stack of status messages.
void gnome_appbar_pop (GnomeAppBar * appbar)	Pop a message from the appbar status message stack.
void gnome_appbar_clear_stack (GnomeAppBar * appbar)	Clear the status message stack
void gnome_appbar_refresh (GnomeAppBar * appbar)	Refresh the status bar and set it to the current value of the stack or to the default. Useful for clearing a message put on by the *gnome_appbar_set_status method.*
void gnome_appbar_set_progress (GnomeAppBar *appbar, gfloat percentage)	Sets the percantage on the progress bar
GtkProgress * gnome_appbar_get_progress (GnomeAppBar * appbar)	Get the GtkProgress widget that is the progress bar so that you can manipulate it.

To get the appbar onto the GnomeApp window, you use the *gnome_app_set_statusbar* method. To also get the menu hints onto the status bar, you need to call *gnome_app_install_menu_hints* with the pointer of your GnomeUIInfo definition for the main menu bar. For the above example, we could add such functionality by adding:

```
GtkWidget *w;

w = gnome_appbar_new(FALSE, TRUE, GNOME_PREFERENCES_USER);
gnome_app_set_statusbar(GNOME_APP(app), w);

gnome_app_install_menu_hints(GNOME_APP(app), menubar);
```

Assume that ’app’ is your GnomeApp application window and ’menubar’ is your GnomeUIInfo pointer to your menu bar definition.

The way the status bar works, is that you have a stack of status messages and the most recent one is displayed in the status bar. This is useful since a lot of times your status will be hierachial anyway, and you might have a long term message that you want to be there unless some short term message needs to be displayed for a while, after this short term one is taken down the one that is ont he stack below it (the long term one) will be displayed.

You should also note that a lot of things are taken care of automatically without you having to actually touch the appbar at all. For example displaying messages on the status bar can be achieved by say *gnome_app_flash* described later in the Talking to the user section.

3.4.5. Talking To The User

A lot of times you will need to cummunicate something to the user, such as that an error orccured or inform him of something. There are preferences the user can set as to where he sees such data, so you should use the following functions rather then just making a new dialog box. This will save you time and will make the application more consistent and flexible.

Table 3-12. GnomeApp Message Functions

Prototype	Description
GtkWidget * gnome_app_message (GnomeApp * app, const gchar * message)	Display a message to the user and require confirmation from the user. In case it was a dialog, it returns the pointer to the dialog, but you can ignore that unless you wish to do something with it.
void gnome_app_flash (GnomeApp * app, const gchar * flash)	Flash a quick message to the user on the status bar for a couple of seconds. Usefull for quick, non-crtical status updates.
GtkWidget * gnome_app_error (GnomeApp * app, const gchar * error)	Similiar to gnome_app_message, but for displaying errors to the user.
GtkWidget * gnome_app_warning (GnomeApp * app, const gchar * warning)	Also similiar to gnome_app_message, but for a warning.

An example might be when you say want to open a file and you fail.

```
FILE *fp;

fp = fopen(filename,"r");
if(!fp) {
        char *err = g_strdup_printf(_("Cannot open file %s"),
                                    filename);
        gnome_app_error(GNOME_APP(app),err);
        g_free(err);
        ...
}
```

Sometimes you might want to get some feedback from the user about a specific situation, instead of just telling him something. One of the following functions might be appropriate. These function take a function pointer, either GnomeReplyCallback or GnomeStringCalback. These callbacks take an integer reply number or a character string and a data pointer. However don't count on the callback ever being called. All of these functions return the widget of the dialog or NULL if the status line was used. Another thing to note is that these are nonblocking, which means that you cannot immediately act as if they are done.

Table 3-13. GnomeApp Query Functions

Prototype	Description
GtkWidget * gnome_app_question (GnomeApp * app, const gchar * question, GnomeReplyCallback callback, gpointer data)	A yes or no type question, the reply callback gets 0 or 1.
GtkWidget * gnome_app_question_modal (GnomeApp * app, const gchar * question, GnomeReplyCallback callback, gpointer data)	Yes or no type of question, but it will not allow the user to interact with the rest of the application until he answers or closes the dialog.
GtkWidget * gnome_app_ok_cancel (GnomeApp * app, const gchar * message, GnomeReplyCallback callback, gpointer data)	An OK or Cancel type of a question.
GtkWidget * gnome_app_ok_cancel_modal (GnomeApp * app, const gchar * message, GnomeReplyCallback callback, gpointer data)	Modal version of the gnome_app_ok_cancel function.
GtkWidget * gnome_app_request_string (GnomeApp * app, const gchar * prompt, GnomeStringCallback callback, gpointer data)	Ask the user for a string with 'prompt'. The callback may get NULL if the user cancels. The string passed to the callback is newly allocated so free it after use.
GtkWidget * gnome_app_request_password (GnomeApp * app, const gchar * prompt, GnomeStringCallback callback, gpointer data)	Ask the user for a password with 'prompt'. Like gnome_app_request_string, but the text is not displayed as it's typed.

For example say we want to ask if one wants to really delete an object.

```
static void
really_delete_handler(int reply, gpointer data)
{
        GtkWidget *some_widget;

        some_widget = GTK_WIDGET(data);

        if(reply == 0) {
                ... /* yes was selected */
        }
}

...

/* ask a yes no question, passing some_widget as the data
   argument to the handler */
gnome_app_question(GNOME_APP(app),_("Really delete object?"),
                   really_delete_handler,
                   some widget);
```

3.5. Using the libgnomeui Library

Besides GnomeApp, the libgnomeui library contains a whole number of other objects and widgets, including standard dialogs, session managment, MDI, and other utility widgets. It also contains the GnomeCanvas widget, which deserves separate treatment. This is the library that makes the programmers life easy. It is important that you use these widgets rather then plain GTK+ ones as these widgets give you consistency and features that a user of a GNOME application expects.

3.5.1. Stock Icons

Since most of the time you will want to use standard buttons and menu items (such as *Open* or *Save as...*), and you want to provide icons with the menu items or tool-bar buttons or just dialog buttons, to make it easier to navigate, you can use some of the predefined icons from *gnome-libs*. These are called *Stock Icons*. You have already seen an example of how to use stock menu icons and regular stock icons in menus and tool-bars (you just use the proper define from *libgnomeui/gnome-stock.h*). There are also stock buttons, where you can get back a button widget based on a stock description.

Here is a list of the *normal* gnome stock icons, these are regular sized for use in tool-bars and other places where you need a normal sized icon. They are given as defines of string constants and their meaning should be obvious.

```
#define GNOME_STOCK_PIXMAP_NEW         "New"
#define GNOME_STOCK_PIXMAP_OPEN        "Open"
#define GNOME_STOCK_PIXMAP_CLOSE       "Close"
#define GNOME_STOCK_PIXMAP_REVERT      "Revert"
#define GNOME_STOCK_PIXMAP_SAVE        "Save"
#define GNOME_STOCK_PIXMAP_SAVE_AS     "Save As"
#define GNOME_STOCK_PIXMAP_CUT         "Cut"
```

```
#define GNOME_STOCK_PIXMAP_COPY        "Copy"
#define GNOME_STOCK_PIXMAP_PASTE       "Paste"
#define GNOME_STOCK_PIXMAP_PROPERTIES  "Properties"
#define GNOME_STOCK_PIXMAP_PREFERENCES "Preferences"
#define GNOME_STOCK_PIXMAP_HELP        "Help"
#define GNOME_STOCK_PIXMAP_SCORES      "Scores"
#define GNOME_STOCK_PIXMAP_PRINT       "Print"
#define GNOME_STOCK_PIXMAP_SEARCH      "Search"
#define GNOME_STOCK_PIXMAP_SRCHRPL     "Search/Replace"
#define GNOME_STOCK_PIXMAP_BACK        "Back"
#define GNOME_STOCK_PIXMAP_FORWARD     "Forward"
#define GNOME_STOCK_PIXMAP_FIRST       "First"
#define GNOME_STOCK_PIXMAP_LAST        "Last"
#define GNOME_STOCK_PIXMAP_HOME        "Home"
#define GNOME_STOCK_PIXMAP_STOP        "Stop"
#define GNOME_STOCK_PIXMAP_REFRESH     "Refresh"
#define GNOME_STOCK_PIXMAP_UNDO        "Undo"
#define GNOME_STOCK_PIXMAP_REDO        "Redo"
#define GNOME_STOCK_PIXMAP_TIMER       "Timer"
#define GNOME_STOCK_PIXMAP_TIMER_STOP  "Timer Stopped"
#define GNOME_STOCK_PIXMAP_MAIL        "Mail"
#define GNOME_STOCK_PIXMAP_MAIL_RCV    "Receive Mail"
#define GNOME_STOCK_PIXMAP_MAIL_SND    "Send Mail"
#define GNOME_STOCK_PIXMAP_MAIL_RPL    "Reply to Mail"
#define GNOME_STOCK_PIXMAP_MAIL_FWD    "Forward Mail"
#define GNOME_STOCK_PIXMAP_MAIL_NEW    "New Mail"
#define GNOME_STOCK_PIXMAP_TRASH       "Trash"
#define GNOME_STOCK_PIXMAP_TRASH_FULL  "Trash Full"
#define GNOME_STOCK_PIXMAP_UNDELETE    "Undelete"
#define GNOME_STOCK_PIXMAP_SPELLCHECK  "Spellchecker"
#define GNOME_STOCK_PIXMAP_MIC         "Microphone"
#define GNOME_STOCK_PIXMAP_LINE_IN     "Line In"
#define GNOME_STOCK_PIXMAP_CDROM       "Cdrom"
#define GNOME_STOCK_PIXMAP_VOLUME      "Volume"
#define GNOME_STOCK_PIXMAP_BOOK_RED    "Book Red"
#define GNOME_STOCK_PIXMAP_BOOK_GREEN  "Book Green"
#define GNOME_STOCK_PIXMAP_BOOK_BLUE   "Book Blue"
#define GNOME_STOCK_PIXMAP_BOOK_YELLOW "Book Yellow"
#define GNOME_STOCK_PIXMAP_BOOK_OPEN   "Book Open"
#define GNOME_STOCK_PIXMAP_ABOUT       "About"
#define GNOME_STOCK_PIXMAP_QUIT        "Quit"
#define GNOME_STOCK_PIXMAP_MULTIPLE    "Multiple"
#define GNOME_STOCK_PIXMAP_NOT         "Not"
#define GNOME_STOCK_PIXMAP_CONVERT     "Convert"
#define GNOME_STOCK_PIXMAP_JUMP_TO     "Jump To"
#define GNOME_STOCK_PIXMAP_UP          "Up"
```

```
#define GNOME_STOCK_PIXMAP_DOWN        "Down"
#define GNOME_STOCK_PIXMAP_TOP         "Top"
#define GNOME_STOCK_PIXMAP_BOTTOM      "Bottom"
#define GNOME_STOCK_PIXMAP_ATTACH      "Attach"
#define GNOME_STOCK_PIXMAP_INDEX       "Index"
#define GNOME_STOCK_PIXMAP_FONT        "Font"
#define GNOME_STOCK_PIXMAP_EXEC        "Exec"

#define GNOME_STOCK_PIXMAP_ALIGN_LEFT  "Left"
#define GNOME_STOCK_PIXMAP_ALIGN_RIGHT "Right"
#define GNOME_STOCK_PIXMAP_ALIGN_CENTER "Center"
#define GNOME_STOCK_PIXMAP_ALIGN_JUSTIFY "Justify"

#define GNOME_STOCK_PIXMAP_TEXT_BOLD    "Bold"
#define GNOME_STOCK_PIXMAP_TEXT_ITALIC  "Italic"
#define GNOME_STOCK_PIXMAP_TEXT_UNDERLINE "Underline"
#define GNOME_STOCK_PIXMAP_TEXT_STRIKEOUT "Strikeout"

#define GNOME_STOCK_PIXMAP_EXIT        GNOME_STOCK_PIXMAP_QUIT
```

If you need to use these outside of *GnomeUIInfo*, you need to get the widget with the pixmap. What you do is you call the *gnome_stock_pixmap_widget* function with your main window as the first argument (so that it can copy it's style) and the icon name (one of the above defines) as the second argument. It returns a new widget which you can just use as a pixmap.

For menus you want to use the *_MENU_* variety of the stock pixmaps. These are smaller and these should be the ones you use for the stock menu items in your *GnomeUIInfo* definitions.

```
#define GNOME_STOCK_MENU_BLANK        "Menu_"
#define GNOME_STOCK_MENU_NEW          "Menu_New"
#define GNOME_STOCK_MENU_SAVE         "Menu_Save"
#define GNOME_STOCK_MENU_SAVE_AS      "Menu_Save As"
#define GNOME_STOCK_MENU_REVERT       "Menu_Revert"
#define GNOME_STOCK_MENU_OPEN         "Menu_Open"
#define GNOME_STOCK_MENU_CLOSE        "Menu_Close"
#define GNOME_STOCK_MENU_QUIT         "Menu_Quit"
#define GNOME_STOCK_MENU_CUT          "Menu_Cut"
#define GNOME_STOCK_MENU_COPY         "Menu_Copy"
#define GNOME_STOCK_MENU_PASTE        "Menu_Paste"
#define GNOME_STOCK_MENU_PROP         "Menu_Properties"
#define GNOME_STOCK_MENU_PREF         "Menu_Preferences"
#define GNOME_STOCK_MENU_ABOUT        "Menu_About"
#define GNOME_STOCK_MENU_SCORES       "Menu_Scores"
#define GNOME_STOCK_MENU_UNDO         "Menu_Undo"
#define GNOME_STOCK_MENU_REDO         "Menu_Redo"
#define GNOME_STOCK_MENU_PRINT        "Menu_Print"
```

```
#define GNOME_STOCK_MENU_SEARCH       "Menu_Search"
#define GNOME_STOCK_MENU_SRCHRPL      "Menu_Search/Replace"
#define GNOME_STOCK_MENU_BACK         "Menu_Back"
#define GNOME_STOCK_MENU_FORWARD      "Menu_Forward"
#define GNOME_STOCK_MENU_FIRST        "Menu_First"
#define GNOME_STOCK_MENU_LAST         "Menu_Last"
#define GNOME_STOCK_MENU_HOME         "Menu_Home"
#define GNOME_STOCK_MENU_STOP         "Menu_Stop"
#define GNOME_STOCK_MENU_REFRESH      "Menu_Refresh"
#define GNOME_STOCK_MENU_MAIL         "Menu_Mail"
#define GNOME_STOCK_MENU_MAIL_RCV     "Menu_Receive Mail"
#define GNOME_STOCK_MENU_MAIL_SND     "Menu_Send Mail"
#define GNOME_STOCK_MENU_MAIL_RPL     "Menu_Reply to Mail"
#define GNOME_STOCK_MENU_MAIL_FWD     "Menu_Forward Mail"
#define GNOME_STOCK_MENU_MAIL_NEW     "Menu_New Mail"
#define GNOME_STOCK_MENU_TRASH        "Menu_Trash"
#define GNOME_STOCK_MENU_TRASH_FULL   "Menu_Trash Full"
#define GNOME_STOCK_MENU_UNDELETE     "Menu_Undelete"
#define GNOME_STOCK_MENU_TIMER        "Menu_Timer"
#define GNOME_STOCK_MENU_TIMER_STOP   "Menu_Timer Stopped"
#define GNOME_STOCK_MENU_SPELLCHECK   "Menu_Spellchecker"
#define GNOME_STOCK_MENU_MIC          "Menu_Microphone"
#define GNOME_STOCK_MENU_LINE_IN      "Menu_Line In"
#define GNOME_STOCK_MENU_CDROM        "Menu_Cdrom"
#define GNOME_STOCK_MENU_VOLUME       "Menu_Volume"
#define GNOME_STOCK_MENU_BOOK_RED     "Menu_Book Red"
#define GNOME_STOCK_MENU_BOOK_GREEN   "Menu_Book Green"
#define GNOME_STOCK_MENU_BOOK_BLUE    "Menu_Book Blue"
#define GNOME_STOCK_MENU_BOOK_YELLOW  "Menu_Book Yellow"
#define GNOME_STOCK_MENU_BOOK_OPEN    "Menu_Book Open"
#define GNOME_STOCK_MENU_CONVERT      "Menu_Convert"
#define GNOME_STOCK_MENU_JUMP_TO      "Menu_Jump To"
#define GNOME_STOCK_MENU_UP           "Menu_Up"
#define GNOME_STOCK_MENU_DOWN         "Menu_Down"
#define GNOME_STOCK_MENU_TOP          "Menu_Top"
#define GNOME_STOCK_MENU_BOTTOM       "Menu_Bottom"
#define GNOME_STOCK_MENU_ATTACH       "Menu_Attach"
#define GNOME_STOCK_MENU_INDEX        "Menu_Index"
#define GNOME_STOCK_MENU_FONT         "Menu_Font"
#define GNOME_STOCK_MENU_EXEC         "Menu_Exec"

#define GNOME_STOCK_MENU_ALIGN_LEFT     "Menu_Left"
#define GNOME_STOCK_MENU_ALIGN_RIGHT    "Menu_Right"
#define GNOME_STOCK_MENU_ALIGN_CENTER   "Menu_Center"
#define GNOME_STOCK_MENU_ALIGN_JUSTIFY  "Menu_Justify"
```

```
#define GNOME_STOCK_MENU_TEXT_BOLD      "Menu_Bold"
#define GNOME_STOCK_MENU_TEXT_ITALIC    "Menu_Italic"
#define GNOME_STOCK_MENU_TEXT_UNDERLINE "Menu_Underline"
#define GNOME_STOCK_MENU_TEXT_STRIKEOUT "Menu_Strikeout"

#define GNOME_STOCK_MENU_EXIT     GNOME_STOCK_MENU_QUIT
```

If you are building the menu yourself and just want to get a menu-item that's built with the stock icon and a label, you can use the *gnome_stock_menu_item* convenience routine. It takes the stock icon type (one of the defines above) as the first argument, and the menu text as the second argument, and it returns a newly created menu-item widget.

Then there are stock buttons. These are for use in your dialogs (see the next section).

```
#define GNOME_STOCK_BUTTON_OK     "Button_Ok"
#define GNOME_STOCK_BUTTON_CANCEL "Button_Cancel"
#define GNOME_STOCK_BUTTON_YES    "Button_Yes"
#define GNOME_STOCK_BUTTON_NO     "Button_No"
#define GNOME_STOCK_BUTTON_CLOSE  "Button_Close"
#define GNOME_STOCK_BUTTON_APPLY  "Button_Apply"
#define GNOME_STOCK_BUTTON_HELP   "Button_Help"
#define GNOME_STOCK_BUTTON_NEXT   "Button_Next"
#define GNOME_STOCK_BUTTON_PREV   "Button_Prev"
#define GNOME_STOCK_BUTTON_UP     "Button_Up"
#define GNOME_STOCK_BUTTON_DOWN   "Button_Down"
#define GNOME_STOCK_BUTTON_FONT   "Button_Font"
```

To get a button widget with the stock icon and text, you can just use the function *gnome_stock_button* with the button type (one of the above defines) as the argument. Now sometimes you want to create a mixture of stock or ordinary buttons, what you can do is call the *gnome_stock_or_ordinary_button* function with either the type of a stock button or just a text for the button label. The function checks if it is one of the above strings, and if it's not it creates an ordinary button widget with the text as the label. Here is an example, it creates three buttons packing them into a box 'box' (I don't include code to make the box), two of the buttons are stock and one is normal:

```
GtkWidget *w;
GtkWidget *box;
int i;
char *buttons[]={
        GNOME_STOCK_BUTTON_OK,
        GNOME_STOCK_BUTTON_CANCEL,
        "Foo",
        NULL
};
...
/* loop through all strings in buttons array */
for(i = 0; buttons[i] != NULL; i++) {

        /* create the button, stock or ordinary */
        w = gnome_stock_or_ordinary_button(buttons[i]);

        /* show and pack it */
        gtk_widget_show(w);
        gtk_box_pack_start(GTK_BOX(box),w,FALSE,FALSE,0);
        /* we should bind signals and other stuff here */
        ...
}
```

3.5.2. Dialogs

3.5.2.1. Generic Dialogs

If you need to create you own custom dialog, *gnome-dialog* is the way to do it. It can handle both modal and non-modal dialogs, although, it's definitely much more friendly to the users of your program if you use a non-modal dialog box, if at all possible, although non-modal dialog boxes tend to have problems associated with them, and sometimes can cause strange bugs, for example if a non-modal dialog box is associated with a window, you'd better bind the *destroy* signal of the window and set it to destroy the dialog box as well, since otherwise it could hang around even though the window or document it was supposed to act on is already dead. However modal dialogs (while definitely easier to program) are usually pretty annoying to use, so avoid them if you at all can.

To make a new *GnomeDialog* widget, just use the *gnome_dialog_new* function. You pass the title of the dialog as the first argument, and then multiple arguments as the button titles terminated by a NULL. The button titles can also be the *GNOME_STOCK_BUTTON_** definitions if you want stock buttons on your dialog. Then you need to add content to the dialog, the dialog is created with a vertical box (*GtkVBox*) for you to use, just by using *GNOME_DIALOG(dialog)->vbox*. Into that you add your content.

You should also set the parent of the dialog to be your main application window (your GnomeApp). This allows the windowmanager to handle the window more appropriately rather then just like a generic window. You accomplish it with the following call:

```
gnome_dialog_set_parent(GNOME_DIALOG(dialog), GTK_WINDOW(app));
```

At this point you have to decide if you want to do a modal dialog or a non-modal dialog. In case you want to do a modal dialog, all you need to do is to call *gnome_dialog_run_and_close* function and it will run the dialog, wait for a user to press a button or close the dialog, and then close the dialog. This function will return the number of the button that was pressed or -1 if the dialog was just closed. In case you don't want to close the dialog when just any button is pressed, you use the *gnome_dialog_run* function, and after you get a result, do what you need to do for that particular button press. Then if you want to run the dialog more, you just loop back to *gnome_dialog_run*, and if you want to close, you run *gnome_dialog_close*. Here's an example of the second scheme.

```
GtkWidget *dlg;
GtkWidget *label;
int i;
...
/*create a new dialog, DON'T forget the NULL on the end,
  it is very important!*/
dlg = gnome_dialog_new("A Dialog",
                       GNOME_STOCK_BUTTON_OK,
                       GNOME_STOCK_BUTTON_APPLY,
                       GNOME_STOCK_BUTTON_CLOSE,
                       NULL);
/* here we assume that app is a pointer to our GnomeApp window */
gnome_dialog_set_parent(GNOME_DIALOG(dlg), GTK_WINDOW(app));
...
/*add some content to the dialog here*/
label = gtk_label_new("Some random content");
gtk_box_pack_start(GTK_BOX(GNOME_DIALOG(dlg)->vbox),label,
                   FALSE,FALSE,0);
...
/*set up an infinite loop*/
for(;;) {
        i = gnome_dialog_run(GNOME_DIALOG(dlg));
        if(i == 0 || i == 2) {
                /*the user pressed OK or close, so we will get
                  out of the loop and close the dialog, or the
                  user pressed */
                gnome_dialog_close(GNOME_DIALOG(dlg));
                break;
        } else if(i < 0) {
                /*the user closed the dialog from the window
                  manager*/
                break;
        } else if(i == 1) {
                /*user pressed apply we don't want to close*/
                ...
        }
}
```

By default the dialog is destroyed when closed, so you don't have to worry about it's destruction. You can change this behavior if you wish though.

If you are doing a non-modal dialog box, things get a little more complicated. You create the dialog as above, but then you bind the *clicked* signal of the *GnomeDialog* widget. That signal has as it's second argument the button number that was pressed. After that you should use the *gnome_dialog_set_close* function to tell *GnomeDialog* that we want to close the dialog when the user first presses any button, if you want that behavior, otherwise you'll have to do *gnome_dialog_close* in the *clicked* signal handler for the buttons you want to close on. After that is set up you just *gtk_widget_show* the dialog. An example follows:

```
/*the clicked signal handler*/
static void
dialog_clicked(GnomeDialog *dlg, int button, gpointer data)
{
        switch(button) {
        case 1:
                /*user pressed apply*/
                ...
                return;
        case 0:
                /*user pressed OK*/
                ...
                /*fall though to close*/
        case 2:
                /*user pressed close*/
                gnome_dialog_close(dlg);
                break;
        }
}

/*somewhere else in the source file*/
...
GtkWidget *dlg;
...
/*create a new dialog, DON'T forget the NULL on the end, it
  is very important!*/
dlg = gnome_dialog_new("A Dialog",
                       GNOME_STOCK_BUTTON_OK,
                       GNOME_STOCK_BUTTON_APPLY,
                       GNOME_STOCK_BUTTON_CLOSE,
                       NULL);
```

```
/* here we assume that app is a pointer to our GnomeApp window */
gnome_dialog_set_parent(GNOME_DIALOG(dlg), GTK_WINDOW(app));
...
/*add some content to the dialog here*/
...
/*bind the clicked handler*/
gtk_signal_connect(GTK_OBJECT(dlg),"clicked",
                   GTK_SIGNAL_FUNC(dialog_clicked),
                   NULL);
/*show the dialog, note that this is not a modal dialog,
  so the program doesn’t block here, but continues*/
gtk_widget_show(dlg);
```

This implements the same dialog as the modal example above, only non modal. Make sure that you have some way of destruction of the dialog in case it’s no longer relevant, for example if a dialog is to modify some object, it should be destroyed when that object is destroyed.

3.5.2.2. Message Box

GnomeMessageBox is an object derived from *GnomeDialog*. As such you use it in the exact same manner, the only difference here is that it automatically sets up the insides of the dialog to be a single label and an icon of the selected message box type. The message box types are as follows:

```
#define GNOME_MESSAGE_BOX_INFO      "info"
#define GNOME_MESSAGE_BOX_WARNING   "warning"
#define GNOME_MESSAGE_BOX_ERROR     "error"
#define GNOME_MESSAGE_BOX_QUESTION  "question"
#define GNOME_MESSAGE_BOX_GENERIC   "generic"
```

To create a message box, you use the function *gnome_message_box_new* with the first argument being the message text, the second argument being the type of the message box (one of the defines above), and then any number of buttons terminated by a NULL exactly as in the *GnomeDialog*’s case. After created it is again used exactly the same as *GnomeDialog*.

3.5.2.3. Property Dialogs

If you have some properties to set in your application, you should use a *GnomePropertyBox* dialog for the preferences to make the applications more consistent. Again this object is derived from *GnomeDialog* so it’s use is similar. But *GnomePropertyBox* defines some new signals, namely *apply* and *help*. They both get passed the page number as the second argument. For help you should use this to display the proper help page, however for apply, this was created for adding a per-page apply button, which was not realized yet, so you should ignore any *apply* signal with the page number other then -1, which is the *global* apply. This can be done with a simple if statement at the top of your apply routine. You can choose to be per-page apply *ready*, by doing a per-page apply in your code, but it is not sure if this code will ever get completed. It should be safe to do just the global apply as that is the only thing implemented in *gnome-libs 1.0*.

To use property dialogs, you call *gnome_property_box_new*, which will create a completely new dialog for you with a notebook and the four buttons. *OK*, which will call your apply handler for all pages and then for the -1 page, and then it will close the dialog, *Apply*, which will call the apply handler for all pages and then for the -1 page, *Close*, which will just close the dialog, and *Help* which will call your help handler if you bound it. You then connect the *apply* signal to your apply handler, and most likely the *destroy* signal on the property box to destroy the data associated with the property box when it closes. You then create the different pages for your property box and add them with, *gnome_property_box_append_page*, which takes your page as the second argument and a label as the third (usually this will be just a *GtkLabel*). You also want to connect the different signals for the widgets on your pages, to mark the property box as changed (otherwise the Apply and OK buttons will not be sensitive). You do this by calling *gnome_property_box_changed* every time the user changed something with the widgets. For example on entry (and derived) widgets you connect to the *changed* signal. Example follows:

```
/*apply handler*/
static void
property_apply(GnomePropertyBox *box, int page_num, gpointer data)
{
        /*ignore page numbers other then -1*/
        if(page_num!=-1)
                return;
        /*do your apply routine here*/
        ...
}
...
/*somewhere else in the source file*/
GtkWidget *pbox;
GtkWidget *widget;
...
pbox = gnome_property_box_new();
gtk_signal_connect(GTK_OBJECT(pbox),"apply",
                   GTK_SIGNAL_FUNC(property_apply),NULL);
...
/*you create a page for the property box and added it to the
  container called widget*/
gnome_property_box_append_page(GNOME_PROPERTY_BOX(pbox),
                               widget,
                               gtk_label_new("SomePage"));
/*then add other pages in similar manner*/
```

```
...
/*we show the dialog box*/
gtk_widget_show_all(pbox);
```

3.5.2.4. File Picking Dialog

Gnome doesn’t have it’s own file picking dialog, although this is planned for the future, for now you need to use the regular *GTK+* file dialog.

Use of the file dialog is very simple. You create the dialog with *gtk_file_selection_new*, passing it the title of the dialog box as the argument. After this you bind the clicked signal on the *OK* and *Cancel* buttons. For example for a loading dialog box, you could check that the file is of the correct type when the user presses OK and if so then close the dialog (usually with *gtk_widget_destroy*). Or for saving dialog, you could ask if the file exists. File selection dialog boxes are usually safe and simple to do non-modal. Just make sure you’d destroy the file dialog box when the object or window it’s supposed to work with. Here’s the routine that invokes the save as dialog for *Achtung*, which is a presentation program we’re working on.

```
void
presentation_save_as (AchtungPresentation *p)
{
        GtkFileSelection *fsel;

        g_return_if_fail (p != NULL);
        g_return_if_fail (p->doc != NULL);

        /* create a new file selection widget */
        fsel = GTK_FILE_SELECTION
                (gtk_file_selection_new (_("Save presentation as")));
        if (p->real_file && p->filename)
                gtk_file_selection_set_filename (fsel, p->filename);

        gtk_object_set_data(GTK_OBJECT(fsel),"p",p);

        /* Connect the signals for Ok and Cancel */
        gtk_signal_connect (GTK_OBJECT (fsel->ok_button), "clicked",
                            GTK_SIGNAL_FUNC (save_ok), fsel);
        gtk_signal_connect_object
                (GTK_OBJECT (fsel->cancel_button), "clicked",
                 GTK_SIGNAL_FUNC (gtk_widget_destroy),
                 GTK_OBJECT(fsel));

        /* set the position at the mouse cursor, we do this because this
```

```
           is not a gnome dialog and thus it isn’t set for us.  You
           shouldn’t do this for normal gnome dialogs though */
        gtk_window_position (GTK_WINDOW (fsel), GTK_WIN_POS_MOUSE);

        /*if the presentation dies so do it’s dialogs*/
        gtk_signal_connect_object_while_alive
                (GTK_OBJECT (p), "destroy",
                 GTK_SIGNAL_FUNC (gtk_widget_destroy),
                 GTK_OBJECT(fsel));

        gtk_widget_show (GTK_WIDGET (fsel));
}
```

This is actually a save_as method for *AchtungPresentation* object in object oriented speak. *AchtungPresentation* is a GtkObject we use for storing all the presentation data (This is a nice example of how to use GtkObject for things not directly related to widgets or GUI programming). First we check the arguments to the function with *g_return_if_fail* which is for debugging purposes. Then we create a new *GtkFileSelection* with a title of "Save presentation as". Ignore the *_()* macro around the string for now, it’s used for internationalization. Afterwards we check if the presentation already has a filename associated with it, and if so we set the filename on the file selection dialog to that. After that we connect the the *OK* button to a routine called *save_ok* defined elsewhere in the file and pass the file selection dialog as a data argument. Then we use *connect_object* to bind the *Cancel* button to destroying the file selection dialog. The *connect_object* method is similar to regular *connect* but when it calls the function itself it will pass the object from the data field as the first argument of the function. So connecting to *gtk_widget_destroy* will destroy the object passed in the data field, which is the file selection dialog. Then we position the dialog near the mouse button. In the future when this dialog is derived from *GnomeDialog*, you will not need to and actually should not do that, as that will be done according to use preferences as for all the other gnome dialogs. After this we use yet another signal connection method ... this time *gtk_signal_connect_object_while_alive*, which is similar to *connect_object*, but has a nice twist to it. The signal will be disconnected when the object passed in the data field dies. This needs to happen as the file dialog will most likely be destroyed before the the presentation itself is, then when the presentation is destroyed itself, it would try to destroy an already non-existent file selection dialog and most likely cause a segmentation fault and crash. This way it is safe and if the file selection dialog is still around when the presentation is destroyed, it is destroyed with it.

3.5.2.5. About Box

You will probably want to have an "About" entry in the "Help" menu of your application, and it should display the standard about box. There is a dialog in gnome for just this sort of puprose, the

GnomeAbout object. It is created with the gnome_about_new which has the following prototype

```
GtkWidget* gnome_about_new(const gchar *title, /* Name of the applica-
tion. */
                           const gchar *version, /* Version. */
                           const gchar *copyright, /* Copyright notice
                                                      (one line.) */
                           const gchar **authors, /* NULL termi-
nated list of
                                                     authors. */
                           const gchar *comments, /* Other comments. */
                           const gchar *logo /* A logo pixmap file. */
                           );
```

After you create the dialog box, you should set it's parent to be your application window with *gnome_dialog_set_parent*. And then you can just show the dialog. The following implements an about box dialog, we assume that VERSION has been #define'd to be the string with the version number. We also pass NULL as the logo, meaning we have no logo picture to display.

```
GtkWidget* dlg;
char *authors[] = {
        "George Lebl",
        NULL;
};

dlg = gnome_about_new("Some application", /* Name of the application. */
                      VERSION, /* Version. */
                      "(c) 1999 George Lebl, /* Copyright notice
                                                (one line.) */
                      authors, /* NULL terminated list of
                                  authors. */
                      "Just some application, "
                      "blah blah blah", /* Other comments. */
                      NULL /* A logo pixmap file. */
                      );

gnome_dialog_set_parent(GNOME_DIALOG(dlg), GTK_WINDOW(app));

gtk_widget_show(dlg);
```

3.5.3. Entries

Sometimes, especially in properties dialogs, you want fields for entering text, files, pixmaps, icons or double precision numbers. This is what the *gnome-*entry* widgets do.

3.5.3.1. GnomeEntry

This is an entry for regular text, but it includes history of previously entered values. Note that this widget is not derived from *GtkEntry*, but owns such a widget. This means that you can't use *GtkEntry* methods on this object directly, but you need to get a pointer to the *GtkEntry* object inside *GnomeEntry*. When you call *gnome_entry_new*, you pass a *history_id* string to it. This is a unique identifier to identify this entry, or this type of entries in your application. All the entries that share this *history_id* will have common history of values. After you create a *GnomeEntry* you use the *gnome_entry_gtk_entry* function to get a pointer to the *GtkEntry* object inside and bind any signals or manipulate text with that instead. Here is an example:

```
GtkWidget *gnomeentry;
GtkWidget *gtkentry;
...
gnomeentry = gnome_entry_new("text1");

/* get the GtkEntry to bind a "changed" signal to figure out when the
   user changed the entry */
gtkentry = gnome_entry_gtk_entry(GNOME_ENTRY(gnomeentry));
gtk_signal_connect(GTK_OBJECT(gtkentry),"changed",
                   GTK_SIGNAL_FUNC(entry_changed), NULL);
```

3.5.3.2. GnomeFileEntry

GnomeEntry is a basis for *GnomeFileEntry*. Again it is not derived, but *GnomeEntry* is owned by *GnomeFileEntry*. This type of hierarchy is throughout all the gnome entry widgets. *GnomeFileEntry* adds a browse button on the right side of the entry, and also accepts file drops from the file manager for example. It's use is extremely similar to *GnomeEntry*. You create the entry with *gnome_file_entry_new*. The first argument is the *history_id* of the *GnomeEntry*, and the second argument is the title of the browse dialog box. To get the *GtkEntry*, you again use the gtk_entry method, named *gnome_file_entry_gtk_entry*. To finally get the filename, you can get the exact text from the *GtkEntry*, or you might use a convenience method, *gnome_file_entry_get_full_path*, which takes a flag *file_must_exist* as it's second argument. If this flag is set, the function returns NULL if the file doesn't exists. If the flag is not set or the file does exist, the function returns the full path to the file.

3.5.3.3. GnomePixmapEntry

This is an entry for entering pixmaps (Images) of any size. It again includes (not derives from) *GnomeFileEntry*, so it can do everything the file entry can (including accepting drops). However this entry adds a preview box for the pixmap above the entry. Also it's file selection dialog includes a preview box to the right side of the file list. It's use is again very similar to the entries above. You call *gnome_pixmap_entry_new* with the same arguments as *GnomeFileEntry*, with an added flag, *do_preview*. This flag specifies if the preview box is visible or not. But be careful, it doesn't save memory not to show the preview, it just saves space. Again you use a *gnome_pixmap_entry_gtk_entry* to get the *GtkEntry* widget. To get a filename of the the pixmap, if it could be loaded as an image for the preview (using imlib), you can use *gnome_pixmap_entry_get_filename*, which returns NULL if the pixmap files doesn't exist or could not be loaded, and the full filename otherwise.

3.5.3.4. GnomeIconEntry

The icon entry is very similar to the *GnomePixmapEntry*, but it is meant for images in the standard 48x48 icon size. Also instead of the preview box, there is a button with the image scaled to 48x48. If you press the button you get a listing of images from the same directory as the current icon. To create an icon entry use *gnome_icon_entry_new* with *history_id* and *browse_dialog_title* string arguments. Once you need an existing icon that is a real image, you use *gnome_icon_entry_get_filename* which works just like *gnome_pixmap_entry_get_filename*. You can also get the *GtkEntry* by using *gnome_icon_entry_gtk_entry*. Example:

```
GtkWidget *iconentry;
GtkWidget *gtkentry;
char *somefilename;
...

iconentry = gnome_icon_entry_new("icon","Browse...");

/* we want to set 'somefilename' as default icon */
gnome_icon_entry_set_icon(GNOME_ICON_ENTRY(iconentry), somefilename);

/* we get the GtkEntry to figure out when we changed */
gtkentry = gnome_icon_entry_gtk_entry(GNOME_ICON_ENTRY(iconentry));
gtk_signal_connect(GTK_OBJECT(gtkentry),"changed",
                   GTK_SIGNAL_FUNC(entry_changed), NULL);

...
/* here we want to get the selected icon */
char *icon;
icon = gnome_icon_entry_get_filename(GNOME_ICON_ENTRY(iconentry);
...
/* make sure to free icon after use */
g_free(icon);
```

3.5.3.5. GnomeNumberEntry

GnomeNumberEntry is an entry widget for entering double precision numbers with a calculator. Most of the time for number entries you want to use the *GtkSpinButton* widget, however for applications such as mortgage calculators, or finance programs, where calculations are necessary, you will want to use this entry type. Basically it's a *GnomeEntry* widget with a button on the right side of it which calls up a dialog with a calculator. The user can use the calculator and press OK and the number entry is updated to what it was on the calculator. To create a number entry widget, just use *gnome_number_entry_new*, passing it the *history_id* as the first argument and the title of the calculator dialog as the second argument. To get the *GtkEntry* widget just use *gnome_number_entry_gtk_entry*. To get the number as a *double* value, use *gnome_number_entry_get_number* method.

3.5.4. Using Images

When you need to use images in your apps, most likely you'll want the *GnomePixmap* widget. It's advantage is that it makes using images much easier without having to learn imlib, which is the image library used by this widget.

There are numerous *new* functions for *GnomePixmap*, depending on the source of the pixmap. The most used will probably be *gnome_pixmap_new_from_file* which takes a filename which is an image loadable by imlib and creates a pixmap widget for you. There is also *gnome_pixmap_new_from_file_at_size* to which you pass also the size to which the image should be scaled. If you have already loaded the image with imlib (in case you wanted to do other things to the pixmap first), you can use *gnome_pixmap_new_from_imlib* and *gnome_pixmap_new_from_imlib_at_size*. Which take a *GdkImlibImage* as the first argument. If you already have a pixmap widget and want to change the image inside it, you can use the *gnome_pixmap_load_** which have almost the same syntax as the new functions, except that you pass the *GnomePixmap* as the first argument, and then the rest of the arguments as above, and of course replace the _new_from_ for _load_.

Here's an example of it's use:

```
GtkWidget *pix;
...
/*load somefile.png and scale it to 48x48*/
```

```
pix = gnome_pixmap_new_from_file_at_size("somefile.png",48,48);
/*now you can pack pix somewhere just like any other widget*/
...
/*now we want to change the files to otherfile.png and do no
  scaling*/
gnome_pixmap_load_file(GNOME_PIXMAP(pix),"otherfile.png");
```

3.5.5. Session Management

Your app should be able to save it's settings and restore them when the user restarts your application, it should also be able to do this for several different sessions. For instance the user might have a normal session, but sometimes log into a special session where he has different settings in applications. *gnome-libs* actually hides the ugly details of this. For the most part you do not need to worry about the real details of session management, unless you wish to do something very clever or if your app does some complicated state saving. To do simple session saving all you need is the following code (mostly taken from gnome-hello-4-SM example program):

```
/*the save_yourself handler, you can safely ignore most of the
  parameters, and just save your session and return TRUE*/
static int
save_yourself(GnomeClient *client, int phase,
              GnomeSaveStyle save_style, int shutdown,
              GnomeInteractStyle interact_style, int fast,
              gpointer client_data)
{
        /*get the prefix for our config*/
        char *prefix= gnome_client_get_config_prefix (client);

        /*this is a "discard" command for discarding data from
          a saved session, usually this will work*/
        char *argv[]= { "rm", "-r", NULL };

        /* Save the state using gnome-config stuff. */
        gnome_config_push_prefix (prefix);

        gnome_config_set_int("Section/Key",some_value);
        ...
        gnome_config_pop_prefix ();
        gnome_config_sync();

        /* Here is the real SM code. We set the argv to the
```

```
                   parameters needed to restart/discard the session that
                   we’ve just saved and call the
                   gnome_session_set_*_command to tell the session
                   manager it. */
                argv[2]= gnome_config_get_real_path (prefix);
                gnome_client_set_discard_command (client, 3, argv);

                /* Set commands to clone and restart this application.
                   Note that we use the same values for both - the
                   session management code will automatically add
                   whatever magic option is required to set the session
                   id on startup. The client_data was set to the
                   command used to start this application when
                   save_yourself handler was connected. */
                argv[0]= (gchar*) client_data;
                gnome_client_set_clone_command (client, 1, argv);
                gnome_client_set_restart_command (client, 1, argv);

                return TRUE;
        }

        static void
        die (GnomeClient *client, gpointer client_data)
        {
                /* Just exit in a friendly way.  We don’t need to
                   save any state here, because the session manager
                   should have sent us a save_yourself-message
                   before.  */
                gtk_exit (0);
        }

        ...
        GnomeClient *client;
        ...
        /*this is somewhere in your main function presumably.
          make sure this is done AFTER the gnome_init call!*/

        /* Get the master client, that was hopefully connected to the
           session manager int the ’gnome_init’ call.  All communication
           to the session manager will be done with this master client. */
        client = gnome_master_client ();

        /* Arrange to be told when something interesting happens.  */
        gtk_signal_connect (GTK_OBJECT (client), "save_yourself",
                            GTK_SIGNAL_FUNC (save_yourself),
                            (gpointer) argv[0]);
```

```
gtk_signal_connect (GTK_OBJECT (client), "die",
                    GTK_SIGNAL_FUNC (die), NULL);

/*check if we are connected to a session manager*/
if (GNOME_CLIENT_CONNECTED (client)) {
        /*we are connected, we will get the prefix under which
          we saved our session last time and load up our data*/
        gnome_config_push_prefix
                (gnome_client_get_config_prefix (client));

        some_value = gnome_config_get_int("Section/Key=0");

        gnome_config_pop_prefix ();
} else {
        /*we are not connected to any session manager, here you
          will just initialize your session like you normally
          do without a session manager*/
        ...
}
```

This is a very simple session management which will be enough for most programs, for more information on session management, you should consult the gnome developer documentation which should be available by now.

3.5.6. Multiple Document Interface

3.5.6.1. The Main MDI Window

If your app handles documents, most likely you will want it to handle multiple documents at one time. Gnome provides an MDI model that is customizable by the user and simple to use. They can use three models of the document display. Either a notebook style which is the most useful one, where documents can be docked in notebooks, and can be dragged out into separate windows if desired. Or a toplevel style where each document is a separate toplevel window. Or finally a modal style where there is only one window and the documents must be switched though a menu. (Note that the examples here are taken from the *gnome-hello-7-mdi* example app in *gnome-libs*, slightly modified. Also note that this example is no longer in gnome-libs, but I reproduce here the really important parts of that example)

To use the MDI features. You basically replace the the *gnome_app_new* call with *gnome_mdi_new* with the same arguments as *gnome_app_new*. To add menus and tool-bar, you use *gnome_mdi_set_menubar_template* and *gnome_mdi_set_toolbar_template* with the GnomeUIInfo as the argument. For MDI, these aren't the actual menus, as it will add it's own items to the menus of each child. After this you set where the menu additions take place. You call *gnome_mdi_set_child_menu_path* to the toplevel menu name after which the child's own menus are inserted. This is the "File" menu in most cases. Then you want to specify the path (menu name) to the menu into which you want to insert a list of the children, you do this by calling *gnome_mdi_set_child_list_path* with the name of the menu and add a '/' on the end of it to specify that you want to insert those items into the menu, not after the menu. Example:

```
GtkWidget *mdi;
...
mdi = gnome_mdi_new("gnome-hello-7-mdi", "GNOME MDI Hello");
...
/*main_menu and toolbar_info are the menu and tool-bar
  descriptions*/
gnome_mdi_set_menubar_template(mdi, main_menu);
gnome_mdi_set_toolbar_template(mdi, toolbar_info);

/* and document menu and document list paths (see
   gnome-app-helper menu insertion routines for details)  */
gnome_mdi_set_child_menu_path(GNOME_MDI(mdi), "File");
gnome_mdi_set_child_list_path(GNOME_MDI(mdi), "Children/");
```

In our GnomeUIInfo structures we have defined a menu named "File" and a menu named "Children". The children menu was not given any items, it's just an empty menu.

Then you should open the main toplevel window with *gnome_mdi_open_toplevel*. This will open a toplevel window without any children. If you wish to use MDI's session management functionality, you can define a function that creates a child given it's name. This is done with the *gnome_mdi_restore_state* method, which takes the config path as the second argument and a function pointer to a function which takes a string and returns a new *GnomeMDIChild* widget (a widget sub-classed from *GnomeMDIChild* actually). Say for example you are using the session management shown above, so you could use:

```
gnome_config_push_prefix (gnome_client_get_config_prefix (client));
restart_ok = gnome_mdi_restore_state(GNOME_MDI(mdi), "MDI Session",
                                     my_child_new_from_config);
gnome_config_pop_prefix ();
```

The restart_ok is a boolean value telling you if the loading actually loaded all the data correctly.

You should also bind the *destroy* signal of the mdi object to do *gtk_main_quit* when the mdi is destroyed.

3.5.6.2. The MDI Children

For complicated apps, all children should be derived from the virtual *GnomeMDIChild* object. For simple apps, you don't need to derive a new object, you can just use the *GnomeMDIGenericChild*, and use the fact that you can store arbitrary data on arbitrary *GtkObject*s to store your own data on the object.

To use the generic child object, you create it with *gnome_mdi_generic_child_new* to which you pass the name of the child. When you get the object, you will need to set it up for your use. First you add a function for creating new views of the same data. A view is just a different window displaying the same file or data. This is done with a call to *gnome_mdi_generic_child_set_view_creator* to which you pass a pointer to a creator function which takes the child widget and a data pointer as arguments and returns a data widget, which is not the actual child widget, but actually the child of the *GnomeMDIGenericChild* widget. After this you set the template for the child's menus with *gnome_mdi_child_set_menu_template*, to which you pass the *GnomeUIInfo* array pointer of the child menu definitions. Then you should call *gnome_mdi_generic_child_set_config_func* to set a function which returns a newly allocated string to save in the config file. This string will be used to load up the child next time you start and do the *gnome_mdi_restore_state* call. It should probably be a filename of the document, or some string from which you can completely recreate that window/document. Then you need to call *gnome_mdi_generic_child_set_label_func* with a pointer to a function that takes the *GnomeMDIGenericChild* as the first argument, the old label widget pointer as the second argument, which would be null if no label widget was yet set, and a data argument. This function can either create a new label and destroy the old one, or just set the label if the label exists. The label can be any widget, for example the *gnome-hello-7-mdi* example code uses a horizontal box widget into which it adds a pixmap and a gtk label. After this if you need to add the child to the mdi yourself, if you are loading a new file for example, you use *gnome_mdi_add_child* and *gnome_mdi_add_view*, to add a new child and a new view to the mdi. If you are creating a new child from the *gnome_mdi_restore_state* function, you should just return the child, the mdi will take care of adding it and adding the appropriate views. You also probably want to set some data on the child widget at this time to store your data with the object.

Here's a short example of creating a new child.

```
GnomeMDI *mdi;
...
GnomeMDIGenericChild *child;
...
/*create a new child named 'name'*/
if((child = gnome_mdi_generic_child_new(name)) != NULL) {
        /*creator of a view*/
        gnome_mdi_generic_child_set_view_creator
                (child, my_child_create_view, NULL);
        /*set a menu template for child menu*/
        gnome_mdi_child_set_menu_template
                (GNOME_MDI_CHILD(child), main_child_menu);
        /*set function to get config string*/
        gnome_mdi_generic_child_set_config_func
                (child, my_child_get_config_string, NULL);
        /*set function that sets or creates a label*/
        gnome_mdi_generic_child_set_label_func
                (child, my_child_set_label, NULL);

        /* add the child to MDI */
        gnome_mdi_add_child(mdi, GNOME_MDI_CHILD(child));

        /* and add a new view of the child */
        gnome_mdi_add_view(mdi, GNOME_MDI_CHILD(child));
}
```

3.5.7. Miscellaneous Widgets

3.5.7.1. Web Links With GnomeHRef

Sometimes you might want to put a button in your app that starts a browser for the user or points an already running browser to some location. All you need to do is to call gnome_href_new with an URL as the first argument and the label as the second. For example:

```
GtkWidget *widget;
...
widget = gnome_href_new("http://www.gnome.org","The GNOME Website");
```

3.5.7.2. Selecting Icons With GnomeIconSelection

Normally you will probably want to select icons using the GnomeIconEntry widget, but sometimes you maybe want to put the icon listing window directly into your application, without having to mess with the icon listing yourself. This widget is only useful if you have some directory from which to pick icons. You create the widget with the *gnome_icon_selection_new*. Then when you want to add a directory of icons, use *gnome_icon_selection_add_directory* method with the argument being the directory to add. You can add multiple directories. Then when you have added all that you want, you have to call *gnome_icon_selection_show_icons* and that will load and show the icons. To select a specific icon, use the *gnome_icon_selection_select_icon* with the filename of the icon to select. The filename should be just the base name of the icon not the entire path. Once you want to get the icon that was selected, you use the *gnome_icon_selection_get_icon* method which takes a boolean argument ’full_path’ and returns a pointer to a string with the file or NULL if nothing was selected. If that ’full_path’ argument is TRUE, the returned value will be the full path of the icon. Note that the returned value points to internal memory so you should not free it. Example:

```
GtkWidget *widget;
char *some_icon_directory;
char *some_other_icon_directory;
...
/* here we create the widget */
widget = gnome_icon_selection_new();
gnome_icon_selection_add_directory(GNOME_ICON_SELECTION(widget),
                                   some_icon_directory);
gnome_icon_selection_add_directory(GNOME_ICON_SELECTION(widget),
                                   some_other_icon_directory);
gnome_icon_selection_show_icons(GNOME_ICON_SELECTION(widget));
...

/* here we want to get the selection (as full path) */
char *filename;
...
filename = gnome_icon_selection_get_icon(GNOME_ICON_SELECTION(widget), TRUE);
```

3.6. GnomeCanvas Widget

While *GnomeCanvas* widget is inside the libgnomeui library, it definitely deserves a separate chapter. The canvas is a very high level high performance graphics drawing widget and on top of that it’s easy to use. It includes support for both Xlib drawn graphics, which is faster especially over the network, and anti-aliased drawing for better looking results.

3.6.1. Creating a Canvas Widget

To create a gnome canvas widget, you call the *gnome_canvas_new*. You need to make sure that the canvas is created with a proper visual and colormap. For example if you wish to draw imlib images inside it, you should do this:

```
GtkWidget *canvas;
...
gtk_widget_push_visual(gdk_imlib_get_visual());
gtk_widget_push_colormap(gdk_imlib_get_colormap());
canvas = gnome_canvas_new();
gtk_widget_pop_visual();
gtk_widget_pop_colormap();
```

After this you also want to call *gnome_canvas_set_pixels_per_unit* to set the scale of the canvas. You can then do *gtk_widget_set_usize* to set the size of the widget, and *gnome_canvas_set_scroll_region* to set the region in which you can scroll around in, this is given in (x1, y1, x2, y2). Basically it's the outer limits of your drawing. So once the canvas was created, you could do:

```
GnomeCanvas *canvas;
...
/*already created a canvas, now set it up*/
gnome_canvas_set_pixels_per_unit(canvas,10);
gnome_canvas_set_scroll_region(canvas,0.0,0.0,50.0,50.0);
```

3.6.2. Groups and Items

In the canvas there are items, the actual objects that are on the canvas, and groups, which are just groupings of items. A group is actually derived from a base *GnomeCanvasItem* object, this is useful to applying functions to all the items inside the group. Such as moving or hiding the entire group. There is also one default group, the root group. You can get this group by calling *gnome_canvas_root*.

3.6.3. Creating Items

Creating items is slightly different usual. It's using the standard GTK+ object model argument mechanism. Basically you call *gnome_canvas_item*, with the parent canvas group as the first argument, the type of object as the second argument, and then arguments given in pairs (argument, value), terminated with a NULL. This is best illustrated by an example:

```
GnomeCanvas *canvas;
GnomeCanvasItem *item;
```

```
...
item = gnome_canvas_item_new(gnome_canvas_root(canvas),
                             GNOME_TYPE_CANVAS_RECT,
                             "x1", 1.0,
                             "y1", 1.0,
                             "x2", 23.0,
                             "y2", 20.0,
                             "fill_color", "black",
                             NULL);
```

Note that it's extremely important that the value be the exact type, since the compiler won't do the cast for you. If you're doing any calculations and aren't sure that you get the right type, just cast it. I believe most if not all numbers for canvas items are doubles.

To find out the arguments that each item takes, consult the gnome documentation or look into the *libgnomeui/gnome-canvas*.h* header files. They contain a table at the top of the file just like the one that follows (which was taken from *libgnomeui/gnome-canvas-rect-ellipse.h*).

For example here are arguments for rectangle (GNOME_TYPE_CANVAS_RECT) and ellipse (GNOME_TYPE_CANVAS_ELLIPSE):

Table 3-14. Arguments for Rectangle and Ellipse Canvas Items

Name	Type	Read/Write	Description
x1	double	RW	Leftmost coordinate of rectangle or ellipse
y1	double	RW	Topmost coordinate of rectangle or ellipse
x2	double	RW	Rightmost coordinate of rectangle or ellipse
y2	double	RW	Bottommost coordinate of rectangle or ellipse
fill_color	string	W	X color specification for fill color, or NULL pointer for no color (transparent)
fill_color_gdk	GdkColor*	RW	Allocated GdkColor for fill
outline_color	string	W	X color specification for outline color, or NULL pointer for no color (transparent)

Name	**Type**	**Read/Write**	**Description**
outline_color_gdk	GdkColor*	RW	Allocated GdkColor for outline
fill_stipple	GdkBitmap*	RW	Stipple pattern for fill
outline_stipple	GdkBitmap*	RW	Stipple pattern for outline
width_pixels	uint	RW	Width of the outline in pixels. The outline will not be scaled when the canvas zoom factor is changed.
width_units	double	RW	Width of the outline in canvas units. The outline will be scaled when the canvas zoom factor is changed.

Now suppose we want to change some of these properties. This is done with a call to *gnome_canvas_item_set*. The first argument to this function is the canvas item object pointer. The next arguments are the same argument pairs as above when creating a new canvas object. For example if we want to set the color to red on the rectangle we created above, we can do this:

```
GnomeCanvas *canvas;
GnomeCanvasItem *item;
...
gnome_canvas_item_set(item
                      "fill_color", "red",
                      NULL);
```

Then there are item methods for other operations on items. For example the *gnome_canvas_item_move* method will take the x and y as second and third argument, and will move the item relative to it’s current position by x and y. Or the *gnome_canvas_item_hide* and *gnome_canvas_item_show*, which hide and show the item, respectively. To control the z order of the items, you can use the methods *gnome_canvas_item_raise_to_top* and *gnome_canvas_item_lower_to_bottom* to raise or lower the item to the top or bottom of it’s parent group’s z order. To have finer control over z order you can use the *gnome_canvas_item_raise* and *gnome_canvas_item_lower* methods which take an extra integer argument which is 1 or larger, and specifies the number of levels the item should move in the z order.

3.6.4. Anti-aliasing Canvas

To create a canvas which uses anti aliasing for rendering of it's items, instead of *gnome_canvas_new* function, you should use the *gnome_canvas_new_aa*. You should also use the *GdkRgb* visual and colormap. So you would do this to create a new anti-aliased canvas:

```
GtkWidget *canvas;
...
gtk_widget_push_visual (gdk_rgb_get_visual ());
gtk_widget_push_colormap (gdk_rgb_get_cmap ());
canvas = gnome_canvas_new_aa ();
gtk_widget_pop_colormap ();
gtk_widget_pop_visual ();
```

After this you can use the canvas in exactly the same manner as the normal canvas.

Anti-aliased canvas items can generally do more then normal canvas items. This is because of limitations of Xlib as a graphics library. It can for example do any kind of affine transformation on it's objects, where on and Xlib canvas you can only do affine transformations on some objects.

3.7. Drag and Drop

While drag and drop belongs into GTK+ itself, I thought it would be better to cover it after some parts of GNOME were discussed.

3.7.1. Accepting Drops

You have already seen one drop handler back when we were discussing the mime types. Basically, to accept drops, you have to decide which mime type of data you want to be able to receive. You have already seen one for "text/uri-list". Basically your handler will only receive data of those mime types that you specify, so you only need to know how to decode those.

To specify the mime types you want to receive, you create an array of *GtkTargetEntry* structures, where the first element is a string of mime type, the second is an integer flag and the third is an integer info. You can leave the flags at 0. The info field can be used if you have several entries you are accepting, as the info integer will be passed to your drop handler, so you can create a switch statement to handle the different types of data. If you have only one type, just leave this at 0.

After this you need to set up the widget for dragging. You do this by calling the *gtk_drag_dest_set* function. The first argument is the widget you want to set up, the second is a flags argument for setting up which types of default drag behavior to use, you can leave this at

GTK_DEST_DEFAULT_ALL. The next argument is the array of *GtkTargetEntry* structures, the next argument is the number of items in that array. The last argument is the type of action that you accept. The types can be any of the following ORed together: *GDK_ACTION_DEFAULT*, *GDK_ACTION_COPY*, *GDK_ACTION_MOVE*, *GDK_ACTION_LINK*, *GDK_ACTION_PRIVATE* and *GDK_ACTION_ASK*. The most useful are *GDK_ACTION_COPY* and *GDK_ACTION_MOVE*. If you are for example passing around strings or other data, you will most likely use *GDK_ACTION_COPY* only.

Then you need to set up and bind the drop handler. The drop handler should have the following prototype:

```
void
target_drag_data_received  (GtkWidget          *widget,
                            GdkDragContext     *context,
                            gint                x,
                            gint                y,
                            GtkSelectionData   *data,
                            guint               info,
                            guint               time);
```

The data you have is in the structure *GtkSelectionData*, in the *data* field. That’s all you need to do for normal DND. Here’s and example:

```
static void
target_drag_data_received  (GtkWidget          *widget,
                            GdkDragContext     *context,
                            gint                x,
                            gint                y,
                            GtkSelectionData   *data,
                            guint               info,
                            guint               time)
{
        g_print("Got: %s\n",data->data);
}
...
static GtkTargetEntry target_table[] = {
        { "text/plain", 0, 0 }
}
...
gtk_drag_dest_set (widget,
                   GTK_DEST_DEFAULT_ALL,
                   target_table, 1,
                   GDK_ACTION_COPY);
gtk_signal_connect (GTK_OBJECT (widget), "drag_data_received",
                    GTK_SIGNAL_FUNC (target_drag_data_received),
                    NULL);
```

For more information about drag and drop, you should see *GTK+* documentation at www.gtk.org (http://www.gtk.org/).

3.7.2. Allowing Drags

Now let's look at the source side of DND. You set up the *GtkTargetEntry* array, in the same manner as above. Then instead of the *flags* argument you substitute a mask for the start mouse button of the drag. This could be *GDK_BUTTON1_MASK | GDK_BUTTON3_MASK* for 1st and 3rd mouse buttons. Then you need to bind the *drag_data_get* signal that will send the data for the drag on it's way, and *drag_data_delete* if the action is *GDK_ACTION_MOVE*, to delete the data since the move was successful. Here's a simple example that will work with the above code snippet for drop:

```
static void
source_drag_data_get  (GtkWidget          *widget,
                       GdkDragContext     *context,
                       GtkSelectionData   *selection_data,
                       guint               info,
                       guint               time,
                       gpointer            data)
{
        char string[] = "Some String!";
        gtk_selection_data_set (selection_data,
                                selection_data->target,
                                8, string, sizeof(string));
}
...
static GtkTargetEntry target_table[] = {
        { "text/plain", 0, 0 }
};
...
gtk_drag_source_set (widget,
                     GDK_BUTTON1_MASK|GDK_BUTTON3_MASK,
                     target_table, 1,
                     GDK_ACTION_COPY);
gtk_signal_connect (GTK_OBJECT (widget), "drag_data_get",
                    GTK_SIGNAL_FUNC (source_drag_data_get),
                    NULL);
```

The *gtk_selection_data_set* function copies the data into the selection data, which is used for the transfer.

Chapter 4. Building GNOME Apps

4.1. Using a Simple Makefile

Using a simple makefile is the fastest way to compile a small GNOME application. If you require a more sophisticated build environment, you should use an autoconf/automake setup, which I will briefly talk about later.

4.1.1. The gnome-config Script

The command line to the C compiler for building a GNOME application can be quite long and would be hard to figure out by hand. So gnome-libs installs a script to simplify this. It is called *gnome-config* and it takes two options, *–cflags* and *–libs*. The –cflags option will give you the compiler flags, needed for the compilation step, and –libs will give you the libraries you need to pass to the linker. You also need to pass another set of arguments to gnome-config. It needs to know what libraries you wish to use. For our purposes, this is *gnome* and *gnomeui*. So for example to get the compiler flags for some program using the standard gnome and gnomeui libraries, you would call "gnome-config –cflags gnome gnomeui".

4.1.2. A Simple Example Makefile

Now to build a simple makefile, you can use variables *CFLAGS* and *LDFLAGS* and the implicit rules that at least GNU make supports (others probably do as well, but I’m not familiar with other makes). So for example let’s say you have an application that has a main.c, main.h, extra.c and extra.h and the executable is called gnome-foo.Now let’s build a small Makefile for this app.

```
CFLAGS=-g -Wall ‘gnome-config -cflags gnome gnomeui‘
LDFLAGS=‘gnome-config -libs gnome gnomeui‘

all: gnome-foo

gnome-foo: main.o extra.o
main.o: main.c main.h extra.h
extra.o: extra.c extra.h

clean:
        rm -f core *.o gnome-foo
```

This is an extremely simple makefile, but it should get you started.

4.2. Using automake/autoconf

Using automake and autoconf is really beyond the scope of this document, but you should go read manuals online at http://www.gnu.org/manual/manual.html, or read the info pages if you have them installed with gnome-help-browser.

There is now an example application which can help you get started with autoconf/automake, the internationalization setup, and other build issues, as well as serve as a good hello world example. You can get it at any gnome ftp site mirror (go to http://www.gnome.org/ftpmirrors.shtml for a list of mirrors) in the *sources/GnomeHello/* directory.

Chapter 5. Conclusion

5.1. Getting Further Help

One of the best ways to get help with programming in gnome is probably to first read the available documentation at www.gnome.org (http://www.gnome.org/), or the developer web site at developer.gnome.org (http://developer.gnome.org/). You should also subscribe to the gnome-devel-list@gnome.org (mailto:gnome-devel-list@gnome.org), to subscribe, send a message with *subscribe* in the subject line to gnome-devel-list-request@gnome.org (mailto:gnome-devel-list-request@gnome.org). To reduce the traffic on the list you should first consult the documentation before asking a question. Also look at www.gnome.org/mailing-lists/ (http://www.gnome.org/mailing-lists/) for a list of all GNOME relevant mailing lists, including the *GTK+* list.

However I still consider the header files for the libraries most helpful. This is mostly as there isn't yet as much documentation out there as there should be, but also the header files will always contain all of the definitions and they will be up to date with the current version, which a manual might not be. Most GTK+ and GNOME function names are very descriptive and it's easy to figure out what they do. I use the header files only. It's much easier to just look at the function prototype and figure out what it does, then to hunt around in a reference manual. Then again you usually have to know what header file to look at, which is not all that hard, given that the header files are named by the objects or modules they represent. For example the header file for *gnome-config* is *libgnome/gnome-config.h*. The header file for *GnomeCanvas* is *libgnomeui/gnome-canvas.h*.

5.2. Future GNOME Library Developments

This tutorial covers programming with version 1.0 of the gnome libraries. But of course there is life after 1.0. There are many things still planned for the libraries. But don't worry, we will try to keep as much compatibility with the 1.0 version as humanely possible.

Here's a short list of things that will be or currently is worked on.

- More common dialogs, such as a native gnome file picker dialog.
- More corba integration of the entire desktop. This is including much more corba support from the core libraries.

Better canvas, including better alpha channel support and printing directly from the canvas.

Rewrite of the configuration setup.

Much much more! ... Stuff we haven't even thought of yet!