GNOME Developers Website Documentation

Table of Contents

Architecture and Design

GNOME provides an excellent framework for building applications by providing a set of core libraries. These include libraries to create graphical user interfaces, high-level components for creating applications with a uniform look and feel, a fast and thin CORBA ORB, and miscellaneous functions for handling configuration files. GNOME also provides libraries for handling XML data and HTTP connections. More importantly, GNOME provides functionality that free software systems have lacked for a long time, like a component architecture and a printing and font framework.

The following sections are devoted to overviews of the GNOME architecture, with examples and notes from the authors themselves. For the most part, these sections are technical in topic, and are aimed at the developer. However, they are presented in a non-technical manner, and people just interested in seeing what the GNOME development environment has to offer should feel at home.

GTK+
GTK+ is the base widget toolkit GNOME uses. It is an offshoot of the GIMP project, and is licensed under the LGPL. It is written primarily in C, although a large number of language bindings are available. Much of the code in GNOME is inspired by the GTK+ project.

GNOME Widgets
GNOME provides a number of higher level widgets that are not in GTK+. These widgets provide a level of consistency between applications, and ease the development of GNOME applications. They also provide a lot of the policy that goes into designing a consistent desktop.

GNOME Miscellaneous
GNOME also provides a number of other services & utility modules for application developers. These are a somewhat eclectic in nature and are lumped together here for convenience.

GNOME Desktop
The *desktop* component of GNOME is closer to what is traditionally thought of as GNOME. It includes the panel, control-center, and the desktop.

Internationalization Issues
GNOME has some support for Internationalization (I18N) and Localization (L10N), and more is on the way. This provides a transparent way for translators to customize applications in GNOME without the application author doing much work.

File System
This section discusses how GNOME interacts with the rest of the File System. It will cover VFS, Mime types, Metadata, and other file related operations.

Session Management
Session Management is the way GNOME handles logging in and logging out. It saves and restores the states of programs, and provides a mechanism for the user to take "snapshots" of their desktop.

Imaging
As befitting its roots in the GIMP project, GNOME has a highly sophisticated imaging subsystem.

Component Model
: GNOME’s distributed object framework is CORBA based, but adds advanced capabilities such as components, document objects, and scriptability.

Development
: GNOME has a fairly complex development environment.

Language Bindings
: Information in writing GNOME programs in various languages can be found here.

External Interfaces
: GNOME has some interfaces to other hardware and software. They are implemented in a way that allows other GNOME applications to take advantage of them.

GTK+

The GTK+ and GLib libraries provide the foundation for the user interface of GNOME. The GTK+ user interface toolkit was originally developed as part of the GIMP (GNU Image Manipulation Program) project and has become widely used because of its attractive appearance, flexible and convenient programming interface and unrestrictive licensing under the GNU LGPL.

GLib
: The GLib library provides functionality which makes C more pleasant and convenient to use. It is used throughout the libraries of GTK+ and GNOME as well as in GNOME programs.

GDK
: Instead of directly building on top of the X Window System, GTK+ introduces an intermediate layer, GDK, which isolates GTK+ from the details of the windowing system. This simplifies things for the programmer and increases portability.

GTK+ Object System
: Although GTK+ is written in C, a language without explicit support for object-oriented programs, the design of GTK+ is heavily object-oriented. The basis of this is a layer known as the *GTK+ Object System*. In addition to such traditional features as inheritance, polymorphism and reference counting, the GTK+ Object System also adds a number of features particularly adapted for a widget toolkit, including a *signal* system for notification and an object attribute system.

Language Bindings
: The straightforward object-oriented design of GTK+ and the fact that it is written in plain C make it highly suitable for access from other languages. Bindings exist for a large number of languages including Python, Perl, C++, Objective C and guile.

Drag and Drop
: Moving information via Drag and Drop (DND) is a capability supported in most modern user interfaces. GTK+ provides a set of interfaces for supporting drag and drop that are both easy to use and highly customizable. By writing to these interfaces, the appliciation can interoperate with programs supporting either the Motif or Xdnd Drag and Drop protocols.

Themes

GTK+ provides support for user-interface customization via *themes*. Without recompiling either GTK+ or the application, a user can choose a new look for their applications by installing a new theme. A theme can either be simply a set of colors and pixmaps used by the existing drawing code or it can be a complete replacement of the functions used to draw widgets.

Documentation

- The GTK+ Reference Documentation Project's goal is to provide comprehensive API reference documenation on GLib and GTK+.
- The GTK+ Tutorial provides a gentle introduction to using GTK+.

GLib

The GLib library provides functionality which makes C more pleasant and convenient to use. It is used throughout the libraries of GTK+ and GNOME as well as in GNOME programs. The functionality provided by GNOME can basically be divided into four categories, portability, convenience functions, generic data structures, and the GLib main loop.

For portability, GLib provides portable equivalents for a number of functions that are available in some, but not all C libraries. For instance, GLib provides functions `g_strcasecmp()` and `g_memmove()` as portable equivalents for `strcasecmp()` and `memmove()`. On platforms where the standard functionality exists, the GLib functions will just wrap these functions, on other platforms, a portable implementation will be used.

GLib also provides a number of unique functions to make using C more convenient. For instance, it provides functions to break strings into words, to do computations with dates, and to log warning messages and error messages in a flexible fashion.

The generic data structures that GLib provides, such as linked lists, hash tables, balanced trees, and variable-length arrays, allow programmers to take advantage of sophisticated data structures and improve the efficiency of their programs without having to reimplement the data structures from scratch. For example, the GHashTable type allows a programmer to create a hash table for arbitrary objects by simply providing two functions, a function to compute hash values for the objects in the table and a function to compare two values.

The last major category of functionality that GLib provides is the GLib main loop. This is a generic and extensible implementation of an event loop. Standard event sources that GLib provides include timers, IO callbacks, and idle functions, but it is also possible to add completely new types of event sources into the GLib main loop. GDK uses this functionality to add an event source for X events. By not tying the main loop directly into the toolkit as is frequently done, GLib allows both graphical and non-graphical event-driven programs (an example of the latter would be a CORBA server) to use the same event loop.

Reference

- GLib from the GTK+/GLib Reference Documentation Project.

Tutorial

- GLib from the GTK+ Tutorial

GDK

The GDK library provides a layer of abstraction that sits between GTK+ widgets and applications and the underlying windowing system. Instead of making calls directly to the X window system, applications call GDK when they need to draw to the screen or handle events.

This extra layer of abstraction provides several advantages. First, it increase portability. Porting GTK+ (and hence, to a large part, GNOME) to another windowing system reduces to porting the GDK layer. A port to Microsoft Windows has already been done. Also, it allows GTK+ programs to transparently use a number of X extensions that may or may not be present. If they are not present, GDK provides substitute functionality in terms of standard X calls. Finally, in many cases, the GDK calls are simpler than the corresponding X calls. Some rarely used parameters are omitted and the correct values for other parameters are determined automatically.

Reference

- GDK from the GTK+/GLib Reference Documentation Project.

GTK+ Object System

Although GTK+ is written in C, a language without explicit support for object-oriented program, the design of GTK+ is heavily object oriented. The basis of this is a layer known as the *GTK+ Object System*. In addition to such traditional features as inheritance, polymorphism and reference counting, the GTK+ Object System also adds a number of features particularly adapted for a widget toolkit, including a *signal* system for notification and an object attribute system.

Inheritance in the GTK+ Object System is achieved by nesting structures within each other. For instance, the GtkButton class inherits from GtkWidget, so the first part of a GtkButton structure is a GtkWidget structure. This means that a pointer to a GtkButton can be cast into a pointer to a GtkWidget. Each class also has an associated *class structure*, which is essentially a table of pointers to the class's implementation of functions that it overrides from the parent class. (For example, the GtkWidgetClass structure includes a pointer to a `draw()` function; GtkButtonClass provides an implementation that draws buttons.)

Each GTK+ object class can have an associated set of signals. Each signal represents a certain type of event or other occurence that an application would want to attach a callback to. For instance, the GtkButton class provides the `"clicked"` signal that is *emitted* when the user clicks the button widget. Any number of callbacks can be connected to a signal and when the signal is emitted all callback will be called in order. Signals can also be used to change the behavior of widgets - for instance, by connecting to the `"insert_text"` signal for a GtkEntry widget, an application can filter the text that the user enters into the Entry to allow only numbers.

Each class also has an associated set of *arguments*; each argument represents some characteristic of the widget that can be queried or set. For instance, the GtkLabel widget provides a `"label"` argument to allow setting the text of the label and a `"justify"` argument to allow setting the justification of the label. The powerful feature of the argument system is that the set of arguments for a class can be dynamically queried at run time. This allows a graphical builder application to provide interfaces for setting the arguments of widgets it didn't know about in advance.

Reference (GTK+ Reference Documentation Project)

- GtkObject
- Signals
- gtk-object-properties.html

Language Bindings

The straightforward object-oriented design of GTK+ and the fact that it is written in plain C make it highly suitable for access from other languages. Bindings exist for a large number of languages including Python, Perl, C++, Objective C and guile.

Links

- C++
  - Gtk--

Drag and Drop

Moving information via Drag and Drop (DND) is a capability supported in most modern user interfaces. The user clicks on a source with the mouse, then drags to a destination. An icon is displayed to give feedback to the user. GTK+ provides a set of interfaces for supporting drag and drop that are both easy to use and highly customizable. By writing to these interfaces, the appliciation can interoperate with programs supporting either the Motif or Xdnd Drag and Drop protocols.

The GTK+ interfaces are separated into two parts, on both the source and destination sides. There is a low-level interface that allows for detailed customization of the drag-and-drop behavior, and then a high-level, simple interface that allows the most common types of drag and drop to be accomplished with very little code.

Internally, GTK+ supports both the new Xdnd drag and drop protocol, which is rapidly gaining support, and the Motif drag and drop protocol. This is mostly transparent to the application programmer - so without special effort, they gain interoperability with both toolkits and thereby with applications now supporting Xdnd, such as Qt and Star Office, and with the large installed base of Motif applications.

The Xdnd protocol specifies that the data types for transferred data are negotiated as MIME types. This convention has been adopted throughout GNOME.

Articles

- Drag-and-Drop in GTK+ and GNOME. A whitepaper which gives an overview of drag and drop in GTK+ and GNOME.

Reference (GTK+ Reference Documentation Project)

- Drag and drop in GTK+
- Information abou the low-level interfaces in GDK. Not generally useful for application programmers.

Themes

GTK+ provides support for user-interface customization via *themes*. Without recompiling either GTK+ or the application, a user can choose a new look for their applications by installing a new theme. A theme can either be simply a set of colors and pixmaps used by the existing drawing code or it can be a complete replacement of the functions used to draw widgets.

There are several concepts important in understanding how themes work in GTK+. A *style* is a set of information about how to draw a particular widget. It includes information about the colors, background pixmaps and fonts for the widget. A style also includes a pointer to a *theme engine* is a shared library with code for drawing the basic components of widgets. (Shadowed boxes, frames, arrows, check-button indicators, etc.).

The colors, fonts, and theme engine to use for the widgets of an application are configured in a *resource file*. The resource file may also contain theme-engine specific data. Finally, a *theme* is the combination of a resource file with any other files it needs, such as image files.

GNOME defines a standard file format for distributing unified themes; a theme file is a tarball containing a single subdirectory named for the theme. Inside this directory there is a `README.html` file and an icon in PNG format called `ICON.png`, and then subdirectories for each type of theme information. GTK+ theme information should be in a subdirectory called gtk, which should contain a resource file called `gtkrc`.

Reference (GTK+ Reference Documentation Project)

- Resource Files.
- Styles.
- Theme Engines.

Links

- gtk.themes.org. A web site devoted to GTK+ themes, including a repository of themes and some information about developing themes.

GNOME Widgets

GNOME provides a rich set of widgets for applications to use in addition to those provided by GTK+. These widgets provides a higher level of user interaction then the GTK+ ones, and enforce a bit more policy then

GnomeApp
This is the primary GNOME widget. This is the top-level widget that most GNOME apps use. It handles creating the title bar, menus, toolbars and statusbars.

GnomeMDI
The MDI architecture allows a user to manage multiple documents in a easy fashion. It is different from the classic MDI style that embeds a window in a window, but rather allows the user to "rip out" notebook tabs.

Dialog Boxes
: GNOME provides functions to generate all types of dialog boxes.

Property Dialogs
: The Property Dialog is used in applications to set its settings. It handles the signals and presentation in a convenient manner.

Druids
: Druids provide a consistent way to walk users through several stages of a configuration process. It fulfills a role "Wizards" play in the windows world.

zvt term
: A complete terminal emulator in a widget.

GnomeApp and GnomeApp-helper

The gnome-app module of libgnomeui implements an object that eases one of the most common tasks in application writing: the main window. The `GnomeApp` widget sets up a main window, and allows easily adding the menu bar, tool bar, and content area to this window. By using `GnomeDock` to hold the tool and menu bars, `GnomeApp` is able to allow the user to rearrange the application layout to suit their needs.

The creation of menu bars and tool bars is further eased by the gnome-app-helper module. Instead of writing code to manually create widgets for each menu item, the developer simply fills in a structure that provides information on the menu/toolbar items to be created, and then calls a function to have the menus and toolbars created. Issues such as stock item icons, submenus, and dynamic help menus are all handled by gnome-app-helper.

Reference

- GnomeApp - API Documentation
- Supplemental Helper functions for GnomeApp - API Documentation

gnome-mdi

The GnomeMDI object offers a method of managing a number of documents, and displaying their views in a consistent, configurable fashion. The developer can create multiple documents, and multiple views for those documents, without worrying about handling the user interface for those multiple documents and views. GnomeMDI can automatically save and restore its state and the state of its children and layout, which is particularly useful to simplify implementation of session managment.

Via the GNOME control center, the user can change which way they want to see multiple views displayed. Currently available options are the notebook, separate window, and single view.

Reference

- GnomeMDI - API Documentation
- GnomeMDIChild - API Documentation
- GnomeMDIGenericChild - API Documentation

GNOME Dialogs

GNOME includes functionality for easily generating application dialog boxes. There are three major types of dialog boxes, the *message box*, the *about box*, and the *generic dialog box*.

The `GnomeDialog` is the basic dialog box. It consists of a row of stock buttons at the bottom of the dialog, and an empty GtkVBox to pack your widgets in (see picture here). Most GNOME dialogs inherit from this widget. GNOME provides an interface to allow developers to easily run and get results from the dialog.

The `GnomeMessageBox` is the simplest of the GNOME Dialogs. It is a dialog with an icon and a message (see picture here). When created, you can specify the type of dialog, and the message. The types available are `INFO`, `WARNING`, `ERROR`, `QUESTION` and `GENERIC`.

The third type of dialog is a lot more specialized. It is the `GnomeAbout` found under the "Help" menu option in most applications (see picture here). There isn't a lot of customization available for this dialog -- you simply pass in the title, the version, the copyright, the author(s), a comment, and an (optional) logo.

Reference

- GnomeDialog - API Documentation
- GnomeMessageBox - API Documentation
- Supplemental "pre-cooked dialogs" - API Documentation

Property Boxes

This section hasn't been written yet.

GNOME Druids

GNOME Druid is a set of widgets that provide a consistent way to walk users through several stages of an initial configuration process. It fulfills the role "Wizards" play in the windows world.

The main widget in the Druid is the `GnomeDruid` widget. It is a container widget that holds the information pages inside. It handles all of the control flow during the setup. It will go through all the pages in it in a linear fashion by default, and keeps track of which page is currently shown. Every page in it is inherits from the `GnomeDruidPage` virtual widget.

There are actually three different `GnomeDruidPage` widgets currently available. They are the `GnomeDruidPageStart`, the `GnomeDruidPageStandard`, and the `GnomeDruidPageFinish`. These widgets are, quite obviously the beginning, middle and end pages of a druid. The first and last are quite similar in look, and let you set a large number of properties, such as color, title, text, and watermark and logo images. The Standard page is a little less defined, has a `GtkVBox` that you can pack your own widgets into. While these widgets are sufficient for most tasks, if you do need a custom widget, you can inherit from `GnomeDruidPage`.

Every time the user changes pages, the current page will send out a "next" or "back" signal to let the application know. It can select which page to show next based on how you handle the signal. In addition, the page can change the sensitivity of the "Next" and "Back" buttons to restrict the user from progressing until enough information has been entered.

Tutorial

- Panel Applet tutoral
- Panel Applet tutoral - Printable version
- Panel Applet documentation (from the source code)

ZVT Term Widget

The ZvtTerm widget is part of the gnome-libs package. It provides a complete **xterm** compatible cursor addressable, colour terminal emulator. It is used by gnome-terminal as the heart of its display engine.

Features

All new, clean, and fast design

The widget consists of all new code designed from the ground up to be easy to maintain and run *fast*. It also has a low memory footprint, and the source is approximatly 1/3 the size of xterm, while still implementing most used terminal features.

(despite claims otherwise, "konsole" is not the only X11 terminal emulator to be written in the last decade!)

Unicode support

Recently added support for UTF-8 display, select and paste. Currently supports fixed-width, iso10646-encoded and wide fonts.

Background pixmaps

A very fast implementation of background pixmaps and pseudo-transparency means all users can have beautiful desktops without heavily impacting their work.

Secure and portable

Only a small external program is required to interact directly with the operating system and pseudo tty interface. This is easily ported to different Unix systems and auditable for security.

Easy to use and feature rich

Through a simple api adding a complete terminal execution environment to any application (including secure tty setup and utmp/wtmp logging) is next to trivial (3-4 function calls). It can also be used as a direct text display engine with colour/attributes and cursor addressible display without the need for a separate sub-process.

Features include configurable colours, pixmaps/transparency, beeps, blinking cursor, selecting by word characters, and more. Plus all the usual stuff like selection/pasting, and scrollback buffer.

xterm compatible
: It aims towards being a terminal-compatible dropin for the xterm program. This is to aid interoperability with foreign systems. The rarely used Tektronix graphics terminal component has been dropped however.

Dingus Click
: Allows auto highlighting of a set of text matching a regular expression. Used by the gnome-terminal to launch a web-browser when the user shift-clicks on a URL.

Actively developed
: Steadily improving feature set and stability.

Reference

There is a comprehensive api reference and programmers guide available.

There is currently no documentation available on the supported escape sequences, although any reference on xterm or rxvt should suffice.

GNOME Miscellaneous Features

This section covers some GNOME programming features which did not fit into any other categories.

gnome-score
: Functions to create and manipulate score files for games.

gnome-config
: Creation and manipulation of config data for GNOME programs.

popt
: Command-line parsing library used by GNOME applications.

gdm
: GNOME replacement for xdm login utility.

XML
: The Extensible Markup Language.

Sound
: GNOME Sound support.

GNOME Game Score System

The GNOME may enable distributed enterprise buzzword-compliant applications, but that doesn’t mean you can’t have a little fun with the GNOME games. To keep track of the results of these games, games track the outcome of a game level, and then provide the GNOME game scoring subsystem with that information so it can maintain a "top ten" list of scores and their owners. This list can be game-wide or per-level. In addition, a dialog box to automatically display the all-time high scores is available to games. This is all done in a secure fashion, so that users cannot record high scores without actually having achieved them.

gnome-config

The core GNOME libraries provide a simple mechanism for storing program configuration information. Applications can save numeric, boolean, or string values using a key/value mechanism. Keys can be grouped into logical sections for organization.

Work is underway to allow for a more sophisticated configuration system that will allow network server-based configuration, notification for applications when a global option changes, and better interaction with session management.

By using the GNOME configuration system, applications get a standard place and mechanism to store their configuration information. This eliminates the need for each application to define its own configuration file format.

The popt argument parser

One of the tasks that almost every application must take care of is acting on the arguments that are passed to it via the command line. To make this task easier, GNOME utilizes the popt library.

The popt library exists essentially for parsing command-line options. It is found superior in many ways when compared to parsing the argv array by hand or using the getopt functions getopt() and getopt_long() [see getopt(3)]. Some specific advantages of popt are: it does not utilize global variables, thus enabling multiple passes in parsing argv ; it can parse an arbitrary array of argv-style elements, allowing parsing of command-line-strings from any source; it provides a standard method of option aliasing (to be discussed at length below.); it can exec external option filters; and, finally, it can automatically generate help and usage messages for the application.

Like getopt_long(), the popt library supports short and long style options. Recall that a short option consists of a - character followed by a single alphanumeric character. A long option, common in GNU utilities, consists of two - characters followed by a string made up of letters, numbers and hyphens. Long options are optionally allowed to begin with a single -, primarily to allow command-line compatibility between popt applications and X toolkit applications. Either type of option may be followed by an argument. A space separates a short option from its arguments; either a space or an = separates a long option from an argument.

GNOME applications can set up custom command line options, and retrieve command line arguments, by using the gnome_init_with_popt_table() function in place of gnome_init(). More information on the use of ’options’, ’flags’, and ’return_ctx’ arguments to this function can be found in the popt programming guide.

GNOME Display Manager

GDM is the GNOME Display Manager. It is an entirely new implementation of XDMCP (the X Display Manager Control Protocol) and associated functionality. gdm consists of three separate parts: A small daemon, a graphical login program and a host chooser. gdm 1.0 implements all significant features required for managing local and remote displays.

gdm daemon:

- X Authentication
- Default and per-display initialization scripts
- Pre and post session scripts
- Pluggable Authentication Modules
- XDMCP
- TCP Wrappers for access control

gdmgreeter:

- Logo image (in any GdkImlib supported format)
- A face browser like the one on NeXT/SGI
- Tab-completion (no, doesn't work on passwords)
- Halt, reboot and laptop suspend
- Iconified login window (i.e. for xfishtank)
- Session selection support (Package manager friendly)
- Language selection support

gdmchooser:

- Visual host browser
- Customizable icons

Most features can be turned on and off in the configuration file by the sysadmin.

XML

XML is a standard to build tag-based structured documents. As the XML FAQ notes,

> "XML is not a single, predefined markup language: it's a metalanguage -- a language for describing other languages -- which lets you design your own markup. (A predefined markup language like HTML defines a way to describe information in one specific class of documents: XML lets you define your own customized markup languages for different classes of document.) It can do this because it's written in SGML, the international standard metalanguage for markup."

gnome-xml, the XML implementation used by GNOME, was created by Daniel Veillard as part of his work for the W3 Consortium. Coming from the originators of the XML specification, it handles XML files in a fully spec-compliant manner, allowing GNOME programs to take the fullest advantage of this technology.

Programs currently using gnome-xml include: gill, dia, gdome, gnome-core/applets/newslashapp, gnome-db, gnome-dom, gnome-print, gnorpm, gnumeric, granite, guppi, guppi2, gwp, libglade, and think. The use of gnome-xml is expected to become even more widespread as new GNOME applications appear.

GNOME Sound Support

GNOME currently (1999-07-10) uses the Esound library to provide constant digital audio access to all GNOME applications. This allows multiple applications to play back and record digital audio streams simultaneously, as well as playing back precached audio samples upon demand. The audiofile library is also used by GNOME to provide easy manipulation of a wide range of digital audio file formats.

In addition, GNOME applications can easily register events for which they want user-configurable sounds to be played, and then trigger those events through the gnome triggers system. The sound events are automatically listed in the control center's sound configuration screen, allowing the user to easily assign different sound files to specific events.

GNOME Desktop Components

The following sections describe the different architectural components which compose the GNOME user interface.

Applets
: Small graphical applications which dock inside the GNOME panel

Capplets
: Control applets - run inside the GNOME Control Center to allow configuration of desktop (for example, desktop background color).

Desktop
: The handling objects on the background of the desktop (eg. icons).

Window Manager Interaction
: How GNOME interacts with window managers.

Panel
: The GNOME panel

Applets

The GNOME panel allows the embedding of small programs, called panel "applets". These applets provide quick access to frequently used information, and permit the user to get common tasks done faster.

Panel applets distributed with the GNOME panel include a laptop battery monitor, an audio CD player, system load monitors, an assortment of clocks, a volume control, a desktop switcher (pager), and a fish called Wanda.

Developers who wish to turn their standalone gtk+ programs into panel applets will find that the process is extremely simple. The panel comes with a library intended to make the task of writing applets easy - all that is necessary is to create an applet widget, and add the contents as with any other container.

The applets use The GNOME CORBA Framework for comunication with the panel. The panel however provides a simple library, which is in fact a **GTK+** widget, which completely hides all the corba API, and makes writing applets, as easy as writing a normal GTK+ application.

It is sometimes desirable to have applets where one process serves several applet windows, such as a clock applet where the user has two instances of the clock on say two different panels. This can be done from one process, thus saving memory.

To make the applet's life easier, the panel takes care of all the session management, position saving, applet launching, and sends the applet signals to dynamically react to changes in the *enviroment* (e.g. Changed orientationof the panel, size changes, etc...)

Capplets

The control-center is used by GNOME to customize the user's desktop environment. It consists of a window with a list of Control Applets, or "capplets" available to the user. These capplets configure parts of the GNOME environment. As an example, configuring the screen saver, background color, mime-type handling and mouse properties are all handled there. GNOME comes with a number of default capplets, but has an easily extendable system.

Generally, only things that are modified by the user for his environment are appropriate for inclusion in the control-center. As an example, configuring a Joystick would be good capplet, while configuring the system's NFS export table is not. In addition, applications should restrict their configuration options to a GNOME property box, instead of allowing external configuration. Finally, if you are setting something up for the first time, a Druid might be better used.

Writing a capplet is quite straightforward. You can substitute a CappletWidget for GtkWindow as your top-level widget, and it will take care of embedding itself. All interaction with the control-center is handled for you, as are any necessary invocations. The control-center itself will prevent more then one capplet running simultaneously. The only tricky part of writing a capplet is handling the try/revert code correctly.

Desktop Icons and File Manager

A very important part of a complete desktop environment is the interface to the user's files. This interface is offered by the GNOME Midnight Commander (gmc), which provides the desktop icons and filesystem windows for GNOME. Using the right mouse button, common operations such as Open/Delete/Copy/Move can be performed, the properties of the desktop icons and files (such as permissions, icons, labels, and such) can be changed, and file-specific tasks can be started. The filesystem windows offer a variety of views (tree, icon, basic list, detailed list, and custom list) as well as file searching and selection. Drag and drop is fully supported, providing the user with an intuitive method of file manipulation.

On a programming level, the developer is available to talk to the file manager via a CORBA interface to ask it to create new windows for specific directories, and close these windows.

Window Manager

People frequently ask about the relationship between the GNOME Project and window managers. GNOME does not specify a particular window manager. It is intended that any window manager can be used. The reason for this decision is that many people are attached to their particular window manager; forcing them to switch just to use GNOME would be counterproductive.

However, to work well with Gnome, a window manager must provide certain features which currently are not implemented in all window managers.

A GNOME-compliant window manager should implement the MWM extended window manager hints. Some GNOME applications will use these hints to increase usability. Here is a proposal showing how to implement these hints.

There is also another proposal, from several people on the gnome-list and gnome-devel-list mailing groups, to create a new set of extended window manager hints. These hints are intended to supplement, and not replace, the MWM hints.

Last, a GNOME window manager should also be a client of the session manager, following the X Session Manager Protocol. This is a requirement for a window manager to be considered even minimally Gnome-compliant.

It’s possible that these hints are insufficient or incorrect in some way. If you are a window manager author and would like to join the discussion of the extensions and additions, please join the wm-spec-list@gnome.org.

Panel

The **panel** is a generic term describing one particular control interface between the user and the desktop environment. A standard scenario is to have a bar at the bottow of the screen, with a menu from which users can launch applications (the **launch menu**) , a button bar with buttons representing more launch targets, as well as running apps. Panel applets can be also be written which dock on the panel, like a clock or an email notification program. There is a region where regular GNOME apps can dock miniviews of themselves, like a small volume slider, which when clicked on brings up the full blown mixer app.

Multiple panels are possible, so that the user can have a full-size panel across the bottom of the screen, and a smaller panel running down the right side of the screen. Panels can have **drawers** which are icons in which you can drop other icons. When you click on the drawer icon it slides out like a drawer, exposing all the icons it contains. Panels can also be set to auto-hide, or can be manually slid in and out of view with a button on each end.

The panel communicates with the Session Manager to indicate when the user is logging out and shutting down GNOME. The Session Manager will then notify all session managed applications to shut down.

The panel needs to react dynamically to changes. For example, if the user has changed the launch menu with a menu editor, the panel needs to be notified and properly re-construct its launch menu on-the-fly. You should not have to completely restart the panel to get the changes in the launch menu, like you do with almost all window managers.

The **launch menu** will be represented as a hierarchical series of directories on the filesystem. Each component of the menu is either another directory (leading to a subfolder), or a text file (which should end with a ’.desktop’ extension) of the structure:

```
[Desktop Entry]
Exec=gnomine
Icon=gnome-gnomine.xpm
Info=Gnome Mines program
Terminal=0
Type=Application
```

In this example, which is for the Gnomines program, the file defines the executable name, the name of the icon, a descriptive name of the program, whether or not to use a terminal to run the program (for something like ’top’), and the type of the object. By default, these files are stored in $(prefix)/share/gnome/apps/, but this can be configured at compile-time. The user can also create a similar tree under $(HOME)/.desktop, to add or override entries in the system-wide configuration. The **gmenu** menu editor is available to edit the menu hierarchy graphically.

Internationalization

Internationalization (often abbreviated to i18n, with the 18 standing for the for the number of letters between ’i’ and ’n’) is the process of making software suitable for use in different countries with different languages.

Localization
: GNOME presents a user interface in the user’s native language. To do this, GNOME utilizes the `gettext()` interface.

Input Methods
: For some languages, complicated processing and feedback are needed as the user enters input. GTK+ uses the X Input Method Extension to communicate with external *input methods* to do this processing.

Unicode and Complex Text Processing
: In the future, GTK+ and GNOME will be adding further enhancements to internationalization. Among these will be consistently using Unicode as an internal encoding and supporting languages written in a right-to-left direction.

Documentation

- A whitepaper on internationalization in GTK+ and GNOME.

Localization

To be useful, a program must present its messages in a language that the user can understand. The process of modifying a program so that it can display its messages in an appropriately translated form is known as *localization*.

For localization, GNOME uses the `gettext()` interface. `gettext()` works by using the strings in the original language (usually English) as the keys by which the translations are looked up. All the strings marked as needing translation are extracted from the source code with a helper program. Human translators then translate the strings into each target language.

Unicode and Complex Text Processing

Although the current versions of GTK+ and GNOME have the ability to handle internationalization for most of the languages of Europe and East Asia, there are a number of other languages that the current framework is not suitable for handling. These include the languages of the Middle East, which are written in a right-to-left direction, and the languages of South Asia, where the display process involves complicated reordering and shaping of the displayed glyphs. Also, the current internationalization schemes use a different encoding depending on the language, which puts additional burdens on the application developer.

Currently, work is underway to address these problems. For one thing, GTK+ will be converted to use Unicode consistently as an internal encoding. This addresses the question of having multiple encodings, and also helps provide a consistent basis for handling right-to-left and complex-text languages.

The actual information about how to render a particular language will reside in separate dynamically loaded modules. Moving to a modular system will allow independent work on the various languages by people expert in those languages, and will avoid increasing the size of the core to the extent that would happen if all languages were supported in the core libraries.

Links

- GScript. Web page for the development of an API for bidirectional and complex-text handling to be used for GTK+.

Input Methods

For some languages, complicated processing and feedback are needed as the user enters input. For instance, for the Asian languages, the user typically enters the pronunciation of the characters, then in a separate step, chooses the appropriate ideographic characters out of various possibilities for that pronunciation.

The input process is handled by a system called a *input method*. A simple input method within the X server is used to handle accent composition for European languages, but generally, input methods are separate processes that display feedback to the user, then send the finished input to the user.

The interaction with input methods is done using the X Input Method Extension, which comes standard with X11R6. There are input methods available for a number of languages, including Japanese, Chinese, and Korean.

GNOME Filesystem Architecture

GNOME provides several modules that add extra functionality to the filesystem provided by the OS.

Metadata
: Metadata allows storing arbitrary attributes for files.

MIME Types
: MIME types uniquely identify the type and format of data stored in a file.

VFS

The VFS (Virtual File System) allows access to different types of file systems in a consistent fashion.

Metadata

The normal Unix file system does not provide a way to store metadata for files, that is, auxiliary information that is stored in *resource forks* in other operating systems. This is information about the file itself, like the type of data it contains, the icon that should be used to represent the file in a file manager, and other miscellaneous information.

GNOME provides a simple way to store metadata for files in a consistent fashion. Each file can have an arbitrary number of key/value pairs associated to it. For example, a key of `icon-filename` may point to a data value that specifies the filename of the icon image to be used to represent that file in a file manager. A key of `icon-position` may point to a data value that specifies the coordinates in which an icon for that file resides in the desktop.

The GNOME metadata functions provide counterparts to common file system operations like copying, moving, renaming, and deleting files. These functions are used to notify the metadata database about changes in the file system structure.

Virtual File System

The virtual file system, or VFS, is an abstraction that allows applications to access different types of file systems in a consistent fashion. This allows uniform access to files in the local file system, ftp sites, RPM packages, and compressed archives. It is currently undergoing a rewrite.

The GNU Midnight Commander file manager uses the VFS to provide the user with a consistent way to access files regardless of their location.

In the future, applications will be able to use a VFS library to be able to access files in any location in the same way that the file manager does. They will also have access to high-level operations like *copy file* or *copy directory recursively* at the VFS level.

MIME types

MIME type is an industry standardized way of specifying the format and nature of a piece of information. It does so by providing an extendable list of types that can be associated with the information. GNOME has support for determining the MIME type of a file, in both a fast manner, and a more accurate, but slower fashion. The fast manner uses a regular expression on the filename to quickly try to determine the type. The slow fashion will read through the contents of the file and try to determine the type by magic.

GNOME also lets the user bind information to each mime-type. Currently, these include open-action, view-action, edit-action, drop-action, and icon-filename. For example, "gimp %f" might be bound to the edit-action of "image/x-png". This information can be set by the system in ${PREFIX}/share/mime-info, and can be overridden by the user on a per-user basis by the mime-type capplet. Currently GMC is the only major application that uses gnome-mime in gnome.

Session Management

GNOME uses the X Session Management Protocol to provide session management to GNOME applications. Session management allows GNOME applications to "save state" when the user exits the system.

- The X Session Management Client Side API can be downloaded from: ftp://ftp.x.org/pub/R6.4/xc/doc/hardc\opy/SM/SMlib.PS.gz
- A document describing details of the protocol can be downloaded from: ftp://ftp.x.org/pub/R6.4/xc/doc/hardcopy/SM/xsmp.PS.gz

GNOME also has some extensions to XSMP.

Session Management Extensions

The Gnome Session Manager complies with the X Session Management Protocol document. However, it also implements a few extensions in order to provide nicer behavior. A client is not required to use these extensions, but it can in order to better integrate with the Gnome desktop.

The current extensions are both new properties which a client can set in order to change how it is treated by gnome-session:

_GSM_Priority

> This property is an `SmCARD8` value between 0 and 99 giving the client's priority level. The session manager starts clients in priority order, starting with priority 0. If a client does not specify a priority, the default priority of 50 is used.

_XC_RestartService

> This property is of type `SmLISTofARRAY8`, and the format is *protocol*/*data*. This property can be used to inform the session manager that an alternate method is required in order to restart this client.
>
> The only currently defined protocol is `rstart-rsh`. In this case, the *data* should be the hostname on which to start the client; the session manager will use the `rstart` program to run the client. If the host running the session manager is the same as *host*, then the client will be started locally as usual.

GdkRGB

> Renders RGB images in any of the possible X visual formats.

Imlib

> Image loading, rendering, and dithering for a wide variety of image data formats.

libart

> High-performance rendering library based on the PostScript imaging model, with support for antialiasing and alpha compositing.

Canvas

A powerful graphics engine that allows high performance rendering and complex applications.

Printing

The GNOME architecture for hard-copy output.

GdkRGB

GdkRGB is a small module of GDK that can render RGB images to any of the possible X visual formats. For example, it will automatically do color reduction and dithering when rendering to an 8-bit pseudocolor visual, or it will use the original data when rendering to a 24-bit truecolor visual.

GdkRGB achieves very high-quality output by using a color cube for color reduction, and a large error diffusion matrix. It can also do dithering for 16-bit truecolor modes, which makes them look almost as good as full 24-bit truecolor visuals. Also, GdkRGB can perform operations on several pixels in parallel, thus increasing performance.

The GNOME Canvas uses GdkRGB to render the final RGB buffers to the screen, thus achieving high quality and high performance.

Imlib

Imlib is a library with two main purposes. First, it can load image files in a large number of formats. Second, it can render those images to any X visual format that the application may need by doing color substitution and dithering.

Imlib has several native image loaders for common image formats like PNG, JPEG, and XPM. If it needs to load an image whose format it does not know about, it can use fallbacks by calling external programs such as ImageMagick or NetPBM.

The ability of rendering 24-bit RGB image data into any X visual representation is very important for GNOME. This allows it to run efficiently on any kind of video hardware while still having colorful icons and graphics, and at the same time allowing legacy X programs to allocate the colors they need.

In addition, Imlib provides miscellaneous image manipulation functions such as scaling, rotation, and color histogram manipulation.

Libart

Libart is a high-performance rendering library that provides a rich imaging model. Libart’s imaging model is a superset of PostScript, and it adds support for antialiasing and alpha compositing (transparency).

Libart is used as the core rendering engine for both the GNOME canvas and the GNOME printing system. It uses sophisticated techniques such as microtile arrays and sorted vector paths to maximize performance.

Libart provides a wealth of vector path-manipulation operations, affine transformations, antialiased and alpha-composited vector path rendering, and functions for manipulating Bézier paths.

The GNOME Canvas

The GNOME canvas is an engine for structured graphics that offers a rich imaging model, high performance rendering, and a powerful, high-level API. It offers a choice of two rendering back-ends, one based on Xlib for extremely fast display, and another based on Libart, a sophisticated, antialiased, alpha-compositing engine. Applications have a choice between the Xlib imaging model or a superset of the PostScript imaging model, depending on the level of graphic sophistication required.

A canvas is a window with a collection of graphical items inside it. The canvas is designed to work as a general-purpose display engine for applications. It provides simple primitive item types such as rectangles, ellipses, and text. These can be used by applications with modest graphics needs. Applications can also define their custom canvas item types to provide sophisticated displays.

The canvas takes care of all drawing operations on its items so that it will never flicker. Canvas items are normal GTK+ objects, and they emit signals based on events from the mouse or keyboard. This allows the programmer to implement interactive behavior for the items very easily.

Documentation

GNOME Canvas White Paper.

Draft proposal for Gnome printing architecture

24 Sep 1998: Sample code is now available. Download the latest release at:

http://www.levien.com/gnome/gnome-print-0.0.1.tar.gz

There will be some minor revisions to the spec, but I think it’s in usable form already. If you’d like your Gnome app to be a testbed for the printing architecture, let me know.

Gnome is in need of a unified printing architecture. This document outlines a proposal for such an architecture, geared towards heavily graphics-intensive applications.

The goals of this architecture include:

- Absolutely uncompromised output quality
- Speed, memory efficiency, and other related performance goals
- Ability to work smoothly with PostScript printers, fonts, and other resources
- A screen display derived from the Canvas
- An extension path for a wide variety of Unicode scripts
- An extension path for a richer set of graphics operators than PostScript supports, especially transparency

- ○ To make life as easy as possible for application developers

Overview

Towards these goals, we propose an architecture comprising several different components. The main component that an application program sees is the *printing API*. This API is implemented as a library (as part of Gnome). Upon initialization, the application recieves a *printing context*, which is conceptually a canvas for the application to paint on using a sequence of paint method invocations. Finally, the application invokes the `showpage` method, which causes the page to be imaged.

The printing context has a virtualized interface, and may represent a simple translation into PostScript, rasterization for a non-PostScript printer, rasterization for the screen, or translation into a display list file format.

Another major feature of the printing API is access to fonts. In our conceptual model, fonts are not associated with a particular printer, but are rather generally available resources, and are sent to the printer when necessary.

Along with the printing API, Gnome will include a *text formatting API*, which will handle the basics of text formatting, including hyphenation, justification, kerning, and ligatures. In the extension path, this API also combines several PostScript fonts into a single *virtual font*, and also handles bidirectional text formatting.

First cut at the printing API

This draft of the printing API contains the functions needed to get basic PostScript printing working. It is expected that many functions will be added later. However, it shouldn’t be too hard to support backward compatibility with these functions.

Initialization

```
GnomePrintContext *
gnome_print_context_new (GnomePrinter *printer);
```

The main function to create a new printing context. For doing a print preview, there may be a similar function that returns both a print context and a print preview widget.

```
GnomePrinter *
gnome_print_default_printer (void);
```

This just returns the default printer. There will be a similar call for popping up a "Select printer" dialog box.

```
void
gnome_print_context_close (GnomePrintContext *gpc);

void
gnome_print_context_free (GnomePrintContext *gpc);
```

The close call sends the rendered pages to the printer (eg by invoking lpr on the temporary file). The free call destroys all data structures and frees up any other resources. If free is called before close, it’s considered an abort.

API calls for rendering vector graphics

A shorthand notation will be used here - the C prototype:

```
int
gnome_print_moveto (GnomePrintContext *gpc, double x, double y);
```

will be represented as:

```
printer->moveto (double x, double y)
```

This notation is intended to be reminiscent of object oriented notations. The return code is zero on success, or an error code on failure.

To make the API as consistent with PostScript as possible, a PrintContext contains quite a bit of implicit state, including a current color, a current path, a current clipping path, a current font, and a host of other settings for the specific graphics operators. It’s up to the specific implementation whether to actually represent this state or to simply pass it along to the next stage in the printing pipeline (i.e. to generate a PostScript file).

One anticipated future extension is the ability of the printing context to *reflect* the graphics state. This will need to be enabled in advance of any painting. When enabled, a number of methods are enabled which return pieces of the implicit state in appropriate data structures. For example, getcurrentpath () returns the current path in a Bezier path data structure. Since many of the operations can do nontrivial manipulations on the state (for example, strokepath ()), this implies that the library actually maintains the state and is capable of serious imaging functions.

Thus, one way to access some of these imaging functions would be to support a *null* printing context, the only function of which is to support these reflection calls. The implementation of the printing API may also choose to dispatch method invocations to both a simple pass-through implementation and the null context implementation to implement enabling reflection.

Like PostScript, the methods are invoked "bottom to top," i.e. each painting method paints over what’s already present. Thus, fairly sophisticated layering techniques should be possible by carefully ordering the method invocations.

```
printer->newpath ()
printer->moveto (double x, double y)
printer->lineto (double x, double y)
printer->curveto (double x1, double y1, double x2, double y2, double
x3, double y3)
printer->closepath ()
```

These calls simply append segments to the "current path" object in the printer context.

We may also want to support the rmoveto, rlineto, and rcurveto operators, which are identical except for representing coordinates relative to the current point.

Also, the arc, arcn, and arcto operators would probably be handy, even though they can be fairly easily simulated using curveto.

```
printer->setrgbcolor (double r, double g, double b)
```

Set the color, with (0, 0, 0) as black and (1, 1, 1) as white. It is expected that the universe of color setting options will expand widely as ICC profile support and prepress graphics are added. But this will do nicely for screen display and basic printing.

```
printer->fill ()
printer->eofill ()
```

Fill the current path, using either the nonzero or even-odd winding rules.

```
printer->setlinewidth (double width)
printer->setmiterlimit (double limit)
printer->setlinejoin (int jointype)
printer->setlinecap (int captype)
printer->setdash (int n_values, double *values, double offset)

printer->strokepath ()
printer->stroke ()
```

These are basically straightforward implementations of the PostScript operators.

Font support

The font methods in the print API are *low-level*. Most applications will probably want to use the higher level interface in the text formatting API.

```
GnomePrintFont *
findfont (char *fontname, double size);

printer->setfont (GnomePrintFont *font);
```

The `findfont` function doesn't work on the basis of implicit state in the print context. Rather, it goes off and finds the font, and returns some kind of handle to it. If the font cannot be found, it returns `NULL`.

```
printer->show (char *text);
```

This works the same way as the `show` operator in PostScript - it displays the text at the current point (i.e. the point set by `moveto`) in the current font, and advances the point. The text is represented as an 8-bit null-terminated string in the font's own encoding. Thus, this function is not very good if kerning, ligatures, or non-Roman scripts are desired. For most applications, the text formatting API will be superior.

Thus, this function is a fairly thin layer over PostScript's `show` operator. One function it will provide, however, is to automatically download the font to the printer if it exists in `.pfb` format on the machine but is not resident in the printer.

Matrix operations

PostScript uses the concept of a *current transformation matrix* (CTM) to represent scaling, rotation, and generalize affine transforms. The matrix is represented as a six-element array. The transformation from user space to device space is as follows:

```
x_device = x_user * CTM[0] + y_user * CTM[2] + CTM[4];
y_device = x_user * CTM[1] + y_user * CTM[3] + CTM[5];
```

The initial CTM represents the bottom left corner of the page as (0, 0) in user space, the point one inch above the corner as (0, 72), and the point one inch to the right of the corner as (72, 0). Note that this coordinate system is "upside down" relative to the usual screen coordinate system.

```
printer->concat (double matrix[6])
printer->setmatrix (double matrix[6])
```

The concat method executes CTM = matrix X CTM, using matrix multiplication as defined in section 4.3 of the PostScript Language Reference Manual, 2nd ed.

The setmatrix method blows away the current CTM and replaces it with the one given. As such, it’s fairly dangerous to use.

The translate, rotate, and scale methods can be implemented as simple wrappers over concat.

The state stack

PostScript’s state has a number of elements that are easy to set, but fairly difficult to unset, specifically modifications to the CTM, and the clipping path. Thus, printing a tree-structured page in which individual nodes modify the state is best done by wrapping the traversal of nodes in a `gsave/grestore` pair. These operators push the entire graphics state on a stack and pop it.

```
printer->gsave ()
printer->grestore ()
```

Clipping

```
printer->clip ()
printer->eoclip ()
```

These methods compute the intersection of the current path and the current clip path, and assign the result to the current clip path. There is no way to expand the clip path except for wrapping the operation in a `gsave/grestore`.

The clip method uses the nonzero winding rule, while eoclip uses the even-odd winding rule.

Images

Image support is a large can of worms - there are many, many options that could be supported. Let's keep it simple for now, though.

```
printer->grayimage (char *data, int width, int height, double matrix[6])
printer->rgbimage (char *data, int width, int height, double matrix[6])
```

The grayimage method is effectively similar to the `image` operator in PostScript, and rgbimage is effectively similar to `rgbimage` with ncomp fixed at 3. Bit depth is fixed at 8bpp.

Lots of extensions are possible here. PostScript supports bitdepths larger than 8bpp, and CMYK color spaces. Other extensions we may want to support include larger color spaces (eg 6-color hifi color), and RGBA images. But those are best left for another day.

Another important extension path is support for ICC color profiles. We expect ICC profiles to be the native color management model in the Gnome printing architecture, with PostScript CRD's basically ignored.

Just print the damned page

```
printer->showpage ()
```

This method closes out the page. If the document is being printed to a temporary file, it may just add an end-of-page code to the file. If the document is being printed directly to the printer, it may start the actual paper in motion. In the case of the null printing context, it just clears out all of the graphics state.

I am confident that this set is sufficient for most basic printing needs.

Text formatting API

It is expected that most printing in Gnome will be done through the text formatting API rather than the low level font methods of the main printing API. Here is a brief list of the additional features:

- Typographic sophistication including kerning and ligatures
- Direct support for reflection
- Easy access to hyphenation and justification
- String encoding is simple Unicode
- We will be able to render text on the screen quicker and better

The basic datatype passed from the application to the text formatting API is a *attributed text*. Conceptually, this is a sequence of 32-bit Unicode/ISO 10646 character codes, each with an associated attribute. In practice, this data will be encoded to save on space. Characters will be UTF-8 encoded. Attributes will be represented as run lengths (i.e. for characters i through j, use this attribute).

The first primary function of the text formatting API will be to convert this attributed text into a list of lines, each of which consists of a list of attributed glyphs. This process is generally known as "hyphenation and justification". In our library, it also includes the steps of kerning and ligatures. This is also the step where "virtual fonts" get resolved into real fonts.

The resulting data structure is opaque to the application (perhaps), but can be queried extensively for geometry info. Example queries are metrics (width of each line, bounding box for each line), and also enough info to resolve an (x, y) coordinate back into a character number.

A special character will be used to represent a "box" for graphics or other in-line element. The dimensions of the box are given as an attribute.

Finally, the list of lines of attributed glyphs is rendered to the printer, using an additional method of the print context.

Here’s what I envision for attributes:

- Font family (the name of the font, eg "Helvetica")
- Size
- Expansion/compression (either through scaling (ugh!) or through multiple master)
- Matrix slanting (ugh!)
- Weight (normal or bold for most fonts, numeric for multiple master)
- Other miscellaneous multiple master axes
- Italics off/on
- Kerning off/on
- Ligatures on/normal/maximal (eg ct ligature in some fonts)
- Tracking (ie letterspace)
- Small caps
- Alternate glyphs, font specific (eg a swash variant)
- Underline (ugh!)
- Strikethrough
- Vertical displacement (ie for subscripts and superscripts)
- Color

It would probably be best to implement attributes using some kind of extensible tag mechanism. I’m tempted to just use XML for the whole thing, but that might make it harder to handle queries back.

This section needs to get filled in with more detail.

A digression on virtual fonts

I see three applications where virtual fonts really help.

First, I think it’s far better to represent small capitals as an attribute rather than a font change. Standard PostScript practice is to include small caps only in a separate "expert" font. Switching back and forth between the regular and expert fonts is a pain at best, and a serious problem when switching to a regular font that does not have a separate small caps font - in the latter case, using a font attribute just causes "false" small caps to be used (i.e. regular capitals set smaller and relatively a little wider).

Second, the standard Adobe encoding only encodes the fi and fl ligatures. Others, including ff, ffi, and ffl, are included in the separate expert font. It shouldn’t be any harder to use these.

Third, when mixing Roman and non-Roman scripts, it would be very handy to have a single unified virtual font that covered both scripts, even though it would be implemented as more than one low-level font.

Virtual fonts also open an expansion path, for example for fonts with more than one color, or fonts rendered using images. In these cases, the font may not be tied directly to a traditional PostScript font at all.

Finally, virtual fonts abstract away from a specific font format. They may, for example, provide a consistent interface for using TrueType fonts as well.

Incremental vs. static rendering, or, the Caanvas

The print architecture as described above is geared towards rendering static pages. It is not good for maintaining a highly incremental display.

The primary challenge of incremental display is to compute the minimal region on the display that needs to be repainted, then traverse only that part of the data structure representing the page. To do this effectively requires detailed knowledge of the geometry of the page elements.

I currently believe that the best course of action is to go ahead with the Caanvas according more or less to the existing plan, but to make it interoperate easily with the printing subsystem.

At a minimum, there will be a printing context for dumping into a Caanvas. In addition, all Caanvas objects will contain methods for painting themselves into a printing context. And, of course, the actual capabilities of Caanvas objects should match the printing methods quite closely.

The preferred method of obtaining an on-screen print preview will be to dump into a Caanvas printing context, then displaying the resulting Caanvas as a Gtk widget. This will handily support scrolling and zooming without any additional intervention of the application.

The direct Caanvas API’s will continue to be based more on self-contained data structures rather than implicit state, so as to make the manipulation and editing of the Caanvas object tree easier, and also to support fast computation of "deltas" as objects change. Nonetheless, the conversion between Caanvas data structures and printing methods will remain simple.

Extension paths

A number of extension paths come to mind immediately. It is important to get the basic printing functionality to work well first, but it is worth planning for a number of future extensions.

Transparency

The Caanvas currently supports partially-transparent colors, as well as full RGBA images. However, PostScript does not. What's needed is a way to render pages containing transparency efficiently in PostScript. Such a task is difficult but not impossible

A sorted display list file format

The best possible printing performance can be obtained using a sorted display list file format. This is also the file format that's best to spool.

Conceptually, a sorted display list consists of a list of elements, each with an associated layer code and bounding box. The bounding box is guaranteed to enclose all successive painting methods.

In an efficient sorted display list, the bounding box starts out enclosing the page, then shrinks as rapidly as possible towards the bottom of the page. Thus, data can be sent to the printer as the file is parsed. By contrast, with PostScript, it is not possible to send the first byte of data to the printer until the last element is handled, because it's always possible that the last element paints the first pixel.

This is not to be undertaken lightly, however.

ICC transforms

For color matching, the Gnome printing model should use ICC transforms. It should be possible to associate an ICC profile with the printer, and, in addition, it should be possible to specify an ICC profile for each image. For many images, no ICC profile is available, and the image can be assumed to be in sRGB.

Prepress

To adequately support prepress, the printing model must at a bare minimum directly support a CMYK color space, as well as the appropriate ICC transforms.

However, doing prepress correctly also requires control over screening, support for trapping, and a host of other tricky things. It would also make sense to support more than four colors at this point, as well.

Internationalization

Rendering non-Roman scripts correctly is difficult. Issues include bidirectional text, placement of diacriticals, infinitely more complex ligature rules, and the need to handle very large fonts well (a typical CJK font is 3-5 megabytes).

Document Structuring Convention

One area of fairly high priority is to get Document Structuring Convention properly supported in the resulting PostScript files. To do this efficiently requires a few additional methods, especially for specifying at the beginning the number of pages and the list of fonts used.

Encapsulated PostScript

On the other side is the ability to import Encapsulated PostScript files. For outputting to PostScript, these can be included inline. For screen display, it’s probably best to invoke GhostScript to render the document.

Performance issues

In this section, I muse on how the preceding specification will be best implemented. I treat each printing context separately.

PostScript output

This section covers generation of PostScript to a file.

For the most part, printing to PostScript is just a question of writing the appropriate `printf` statements for each of the methods. The one tricky area is to download the fonts to the printer before they’re needed.

As we add extensions (mostly for transparency), the PostScript output method will become less and less trivial.

Direct rendering to a buffer

One of the other modes that will be supported is direct rendering to a buffer. This is done using the graphics primitives currently being developed for gfonted and the Caanvas.

This is not, however, the best approach for printing to a color inkjet printer. Keep in mind than an RGB buffer covering an 8x10 print area at 720 dpi is about 124 MB.

To solve this problem, display list techniques will be used instead.

Rendering to the Caanvas

One of the more efficient modes of operation will be to render into the Caanvas, effectively creating a display list, then use the Caanvas architecture to render to the screen or actual printer.

Essentially, each fill, stroke, image, or show method invocation becomes a Caanvas object. Each concat or clip method invocation causes a new node in the grouping tree, while gsave and grestore control the structure of the tree.

When the page is stored as a display list, it is much easier to efficiently implement repaint of exposed areas, as well as to support scrolling. In addition, zooming of a display list can be implemented without any work from the application side.

The Caanvas architecture is also one of the better ways of managing output to color inket printers. The simple "band" technique of rendering long narrow strips, then sending those to the printer, should work best.

The display list approach has one serious drawback: the entire page must be stored in memory, more likely than not duplicating structures stored by the application.

Rendering to non-PostScript laser printers

Here, the main issue is rendering plain text efficiently. In general, for a display list consisting of mostly plain text, the most efficient course of action is to render each glyph actually used in the document, send those to the printer, then simply invoke the glyphs as the display list is traversed.

For images and more complex graphics, this optimization is irrelevant, and it’s best to just render the entire page as a bitmap and send it to the printer.

A concept: image callbacks

One way to deal with the memory usage issues of a large display list is to represent images as callbacks to the application rather than big buffers of pixels. Then, whenever image data is needed, the rendering engine (be it PostScript or whatever) calls the callback, passing in a bounding box and a scaling factor (possibly an entire transformation matrix). This technique also supports low-resolution images for screen display as well as high-resolution images for PostScript output.

There should probably be a "free" call of the callback to indicate that the printing subsystem no longer requires the image data and will not call the callback again.

This technique addresses the main drawback of the display list architecture. The extra complexity is almost certainly worth it.

Printing to color inkjets

We’ve already covered most of the performance issues, but I’ll go over printing to a color inkjet in slightly more detail.

The application starts by getting a "color inkjet" print context. This context dumps the page elements into a display list, as described above.

Upon the showpage method invocation, the display list is complete. The print driver begins to render the page, one band at a time.

A typical band size would be the page width (about 5000 pixels for 7 inches at 720 dpi) by about 64 pixels tall, and 3 or 4 planes deep (depending on whether the page model is RGB or CMYK). This buffer is about a megabyte.

For each band, the print driver traverses the display list. Hopefully, most objects can be culled out using bounding box computations. After traversing the display list, the driver has in hand about a megabyte worth of raw image pixels. It then halftones these (using error diffusion) and writes the halftoned pixels (using the printer’s native escape codes) to a temporary file.

When the print context is closed, the print driver invokes lpr on the resulting temporary file.

For future work, the temporary file should be bypassed and there should be some way to start the paper moving as soon as the display list is in hand.

levien.com Gnome home

CORBA and components enable GNOME programs to reuse code, add scriptability, and enhance interoperability.

Bonobo
: Bonobo is a set of standard interfaces that are used as part of the GNOME project to provide standard component programming in Unix.

DOM
: Document Object Model is a standard interface for access to document elements.

Gnorba
: The GNOME CORBA framework.

ORBit
: The CORBA implementation used by GNOME.

Bonobo

Bonobo is a set of standard interfaces that are used as part of the GNOME project to provide standard component programming in Unix. The Bonobo interfaces are all based on the GNOME::obj interface that provides the foundation for discovering the functionality provided by components.

Bonobo contains a number of interfaces and libraries that provide a way to create reusable graphical components for Unix systems: instead of creating huge monolithic applications that contain every possible feature desired by the user, the user or the application would use existing components to get their job done.

Bonobo provides mechanisms for creating reusable components, reusing these components and creating compound documents from a collection of components.

Document Object Model

The Document Object Model (DOM) is a model that allows documents to contain objects that can be used, and manipulated. This allows the document to be easily changed by adding, deleting or editing elements or object attributes. This also allows for scripts and programs to update a document dynamically. The World Wide Web Consortium has a specification for DOM on its DOM page.

gnome-dom will be the GNOME implementation of the DOM specification. *gnome-dom* will contain libraries, and a server (via CORBA/ORBit) built on top of *libxml*.

gnome-dom currently exists in CVS as a skeleton implementation - volunteers are welcome to assist in completing it.

The GNOME CORBA Framework

The GNOME CORBA framework allows applications to easily use ORBit, the CORBA implementation used by GNOME. One part of this framework provides integration of the ORBit main loop with the Gtk+ main loop, security for incoming CORBA requests, and a standard method of accessing the GNOME name service.

To allow applications to request access to a specific CORBA object, GNOME CORBA servers place information in GOAD, the GNOME Object Activation Directory. This directory stores information on the CORBA objects that a program can provide to other programs. Each directory entry contains a unique implementation identifier (the "GOAD ID"), a list of interfaces that the object supports, a human-readable description of the object implementation, and information on how to create a new instance of the object implementation.

If an application provides the implementation for a CORBA object, it is simple to integrate that object into the GOAD system. An application would install a ’.goad’ data file into the correct directory as part of its installation process. Then, a few simple function calls must be made when the object is created and destroyed, and the application must handle an extra command line option. Once an object implementation is registered with GOAD, client applications can activate that implementation with a single function call.

Also provided as part of the GNOME CORBA framework are a number of interface definitions for commonly used interfaces, such as factories and reference-counted objects.

ORBit

CORBA (the Common Object Request Broker Architecture) is a standard for "distributed objects". This basically means that applications may make invoke operations on objects that are not located in the same address space. Frequently object client and server are in different processes or even on different computer systems. If you are familiar with how RPC (Remote Procedure Call) works, then thinking of CORBA as an object-oriented RPC specification may be helpful.

To make use of CORBA technology, applications must go through an ORB (Object Request Broker) library that implements the CORBA API. The ORB hides all the low level communications that are necessary for sending requests to objects, receiving replies from them, and making object implementations accessable. To the application, invoking an operation on a distributed object acts the same as a local function call.

When GNOME first started to make use of CORBA, it made use of the MICO ORB. Mico did not fit GNOME’s needs very well, though, so Elliot Lee and Dick Porter decided to write a new ORB from scratch. Thus ORBit was born.

Today, ORBit is the CORBA implementation used by many of the GNOME components. It is fast and lean, allowing the use of CORBA in areas that would not normally seem practical. It supports much of the CORBA 2.2 standard, and has hooks that allow easy integration with GNOME programs. For more information on ORBit, visit the ORBit web page.

Development

Glade Builder
: The graphical interface builder for GTK+ and GNOME programs. It is used to rapidly prototype and build applications with

Build Environment
: The utilities used to make maintaining GNOME programs easier.

Glade

Glade is a user-interface building program. It is used to rapidly prototype GNOME and GTK applications. It provides a framework that allows an application author to dynamically add, remove, and modify widgets. It is done in a very simple, yet powerful manner that lets even novice developers design a user interface in a short time.

Glade has two methods of saving its file. One involves generating the code necessary for a project. There is a C and a C++ based backend currently, and there is no reason why more couldn’t be added. This means that someone does not need to do the occasionally pedantic work behind GUI creation.

The other method saves the generated GUI to an XML file. This can be loaded up again by Glade, unlike saving to code. Additionally, applications can use it through `libglade`. This library can load the XML files and display the results at runtime.

Also, the application can get pointers to arbitrary widgets by just referencing a label. The advantage of this is that it splits the GUI away from the code, allowing a separation between the applications design and its behavior.

Reference

- Glade Home page

Build Tools

GNOME software packages make use of the following build tools for portability and Makefile maintainance:

autoconf
: Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of Autoconf when they are run, so their users do not need to have Autoconf. (autoconf info page)p

libtool
: libtool is a portable interface to the process of creating shared libraries.

automake
: Automake allows the creation of commonly used Makefiles via a macro language, and integrates with autoconf & libtool.

For more information on the build tools you can check out the Tools section of this site.

Language Support

One of the primary goals of GNOME has been to facilitate using GNOME from any programming language. By allowing the development of programs in the language that best meets the task, productivity and program quality increase.

Language bindings for Gtk+ are available for a wide variety of languages (C, C++, Objective C, Ada, Perl, Python, Guile, TOM, Eiffel, Dylan, JavaScript, Pike, Pascal, Haskell), and many of these languages also have support for GNOME widgets available. Work is underway to lengthen the list of supported languages even further.

External Components

Pilot Stuff
: An architecture for interfacing to the Pilot PDA.

GnomeDB
: The GNOME database access API.

Gnome Mailer

Allowing access to electronic mail from a diverse set of applications

Palm Pilot

The Palm Pilot support for GNOME is very immature right now. The conduit system is going through a rewrite. More information will be posted here as it becomes available.

If you would like to become involved with the GNOME/Pilot development you can subscribe to the *gnome-pilot-list@gnome.org* mailing list. For information on subscribing to this list and others go to the GNOME Mailing Lists page.

GNOME DB

GNOME DB is an attempt to implement a sort of unified data access for GNOME. Currently it supports connections to MySQL and Postgres, as well as any database with an ODBC API. Future implementations will allow different libraries offering access to flat files and DBs without ODBC drivers.

CORBA access to the database structure is allowed through the GDA ("GNOME Database Access") set of interfaces. These interfaces allow manipulation of database in a more structured, object-oriented fashion than SQL.

GNOME DB offers to the developer a higher level interface to the database and the operations performed on a database then ODBC does. Beside the "bare bone" IDL interface a client library is implemented which uses an object oriented approach to result sets returned from database operation. Using GTK’s object a recordset is derived from GTK’s Adjustable class. So using scrollbars for recordsets is just a matter of connecting the correct signals and handler functions. As an additional benefit the implementatio if MVC (Modell, View, Controller) applications is very easy.

Current work concentrates to provide a graphical user interface library, which elements can be used as BONOBO components, so that it’s easy to embed a grid for browsing a recordset in any BONOBO application. The other area much work is currently done is the management and configuration of databases and to ease the initial setup of a working database environment.

GNOME Mailer

The GNOME Mailer project is designed to create a backend to a GNOME mailing system that will handle various mail protocols.

Camel is the generic messaging library. It will eventually support the standard messaging system for receiving, sending and storing messages. You can read the preliminary Camel organization document.

GNOME Mailer is on the GNOME CVS server in the /gnome-mailer module.

Camel draws heavily from JavaMail and the IMAP4rev1 RFC. You can read the JavaMail API specification.

Gnome Mailer developpement is coordinated in the gnome-mailer mailing list. You can subscribe to this list by sending a mail to gnome-mailer-list-request@gnome.org with the word "subscribe" in the subject. You can also browse the archives.