Chapter W28 The Modern Linux GUI - Wayland and Weston

Objectives:

\* To give a basic overview of how a GUI desktop system on modern Linux works

\* To give the basic concepts behind the Wayland Protocol, and the Weston Compositor

\* To present the following model of developing practical GUI client applications-

Data Generation → Window Generation/Construction → Data Mapping to the Constructed Window

\* To provide simple examples of using the GTK+ and Qt toolkits to create widget graphics that comprise the Window Generation/Construction phase of the above model

\* To detail the use of a Data Generation application, gnuplot, and how gnuplot can be used to map its data to windows created by Qt Creator

To cover the Commands and Primitives-

**gnuplot,weston,weston-launch, xwininfo**

W28.1 Introduction

Everybody knows how to use a GUI on a computer, even kids in the 1st grade. So why present information about GUI’s on Linux to sophisticated college-level and beyond adults? Most importantly because if you use the model that we present here in this chapter, you can harness the GUI capabilities of your Linux system for the development of applications in the most productive and effective way.

That model, in skeletal form, is this-

Data Generation → Window Generation/Construction → Data Mapping to the Constructed Window

Constructing cool-looking GUI interfaces is great, but unless you can hook them up to something that generates useful data that you want to display for some important purpose, they just look good but do nothing. That data generation component of the model could be a database management program, a spreadsheet, or a scientific calculations application that writes volumes of data to external files, etc.. Or in our simplified case shown here, a plotting program such as Gnuplot.

Our model with a little more flesh on its bones is now this:

a) Gnuplot (or the Data Generation component of the model in our presentation) produces the data for graphing,

b) a window compositor like Weston manages all windows on-screen, and handles the window dressing and the frame you want to show the graphed data in,

c) toolkits such as GTK3+ or Qt produce the windows,

d) and then Gnuplot renders a pretty and useful picture, the graph of the data, in the appropriate window.

We start by describing the modern Linux GUI protocol and reference window composition software- Wayland and Weston. We then examine some of the most basic aspects of the toolkits GTK3+ and Qt. Finally, we describe how to work with Gnuplot to implement our model.

A note here on the Linux system distributions and their versions that we use in this chapter. At the time of the writing of this book, the mainstream Linux distros that by default use the modern GUI protocol Wayland, and its window composition software Weston, were all from the Debian-family. Even the purest of these, RebeccaBlackOS is a Ubuntu-derivative. We refer to them in this chapter as our Wayland Reference Systems- Ubuntu 17.10, Ubuntu 18.04, and Linux Mint 19. In Sections W28.1 through 4, all of these reference systems can be used. But in Sections W28.5 through W28.7, we utilize other Linux distros and their versions, such as Linux Mint 18.3 and CentOs 7.5. Those systems and versions are more compatible with the subjects shown in those sections, and the way we present those subjects. Perhaps in the future, all Linux distros and versions can be used effectively by a beginner to execute the model components as we show them here, and at this particular point in time.

W28.1.1 Wayland and Weston

Question: What are Wayland and Weston?

Answer:

GUI interactivity on Linux systems is based around events and requests, as we further detail in Section W28.2. These events and requests (as “messages”) transit between “compositor” software, which controls the GUI display (and also hooks up to the interactive hardware devices, like mouse and keyboard,) and “client” software applications that may be running on the same system as the compositor.

Wayland is the controlling software protocol, that allows a compositor to communicate via events and requests with clients. It uses an asynchronous, pseudo-OOP approach to achieve the message passing.

It is useful for someone trying to understand where Wayland and Weston fit into GUI interactivity to think of the compositor as a window manager. A compositing window manager gives each client application its own memory buffer, and then it “composites” all these buffers into an image in the display memory. The compositor can be a standalone display server running on the Linux kernel, using modesetting and evdev input devices, or a communicating with a Wayland-enabled client. The Linux kernel includes a Direct Rendering Manager (DRM), which has two parts: the kernel drm driver, and your hardware graphics card driver; all contemporary graphics cards support direct rendering. Direct rendering harnesses the Graphics Processing Unit (GPU) directly, and is much faster at processing video than the CPU.

To quote the principal developer of Wayland (underlining added by author):

“The Wayland protocol is essentially only about input handling and buffer management. The compositor receives input events and forwards them to the relevant client. The clients creates buffers and renders into them and notifies the compositor when it needs to redraw. The protocol also handles drag and drop, selections, window management and other interactions that must go through the compositor. However, the protocol does not handle rendering, which is one of the features that makes Wayland so simple. All clients are expected to handle rendering themselves...”1

1 https://wayland.freedesktop.org/faq.html

Weston is a Wayland compositor, and is in a sense the prototypical implementation of a Wayland compositor.

W28. 2 The Key Components of Interactivity: Events and Requests

When you work interactively with a Linux computer, you provide input in a variety of ways, and the computer, after doing some processing, gives you feedback in return. Limiting the form of this feedback to text and graphics, usually the computer responds by displaying information on the screen.

On a modern computer workstation, you are able to use several devices, such as keyboard, mouse buttons, digitizing tablet, touchscreen, and so on, to provide input to an application program in a style of interaction known as *interrupt-driven interaction*. The application is processing data, or in a wait state, until signaled by a particular input device. These interrupt signals are known as input *events* from one or more devices, which can be ordered in time by forming a list of them, or *event* *queue*. The applications written to interact graphically, the *client*s, can then process this queue of input events, do the work necessary to form responses to the events. They can then output the responses as *requests* for graphical output to the graphical *server* display. A schematic illustration of this is shown in Figure W28.1.

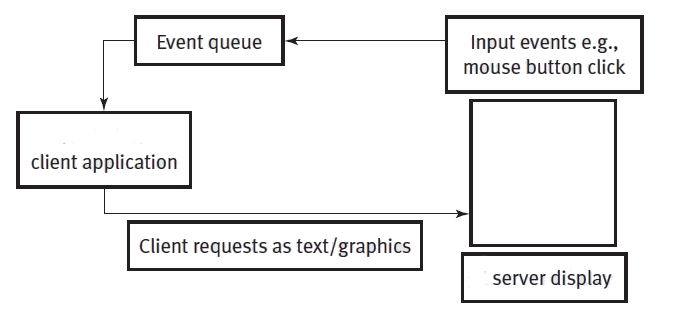


Figure W28.1 A Generic Event–Request Model.

In terms most applicable to this chapter, Wayland as a communications protocol, is represented by the arrows in Figure W28.1. Wayland treats these arrow paths, on one level known in Wayland jargon as the “wire protocol”, as Linux system processes communicating using Inter-Process Communications (IPC.) We give a system programming description of this general method of communicating in Chapter W20. On another level, it treats the arrow paths as objects using an asynchronous message passing model.

What is important to note here is that client application and server display program (the compositor) are just complex computer software programs. The client application, and server display program are running in user space, and the two-leveled Wayland protocol runs and exists in kernel space. The chief takeaway from this last sentence is that client application programming, via the use of the “toolkits” we show in Sections W28.5 and 6, are exposed to developers that can write and construct user-written programs, without having to deal directly with the underlying complexity of both the Wayland protocol and kernel-level programs.

A key concept in this model that sometimes causes confusion is the difference between *server* and *client*. One possible cause for confusion here is that traditionally, on a computer network, a server is thought of as a machine that serves files to many other machines, which is certainly a different function than a server as illustrated in Figure W28.1. A server in the context of what we are showing here is the software that actually displays, or serves output, to the user as graphics, and it is driven by, and drives the hardware devices physically attached to the computer. For example, the keyboard, mouse, and monitor screen in front of the user are parts of the server; they graphically get and *serve* information from and to the user. The client is an application program that connects to, receives input events from, and makes output requests to the server. Be aware that sometimes, and in certain contexts, the client is spoken of as a hardware device, like a workstation or computer. We avoid this, and we will always use the term *client* to refer to application program code, rather than to a piece of hardware. As stated in Section W28.1, a server and client can exist on the same workstation or computer, and use interprocess communication (IPC) mechanisms, and Linux socket mechanisms, to transfer information between them. A *local client* is an application that is running on the same machine that you are sitting in front of. A *remote client* is an application that is running on a machine connected to your server via a TCP/IP network connection. But they both look and feel exactly the same to the user when they appear on her screen.

W28.3 How Wayland Renders Images on your Screen Display

To further clarify the event-request model described in the previous section, we present Figure W28.2, which, in a simplified way, illustrates the “rendering pipeline” from client application to screen display. A provide a few important definitions, as they apply to Wayland and Weston, which we hope will help you to understand the mechanisms at play in Figure W28.2.

Compositor

The intermediate software that manages events and requests to and from client applications, and the underlying graphics hardware of the computer. Usefully thought of as a “window manager”. Weston is a good example of this.

DRM

The Linux kernel’s Direct Rendering Manager that allows user space applications to send drawing commands directly to the Graphics Processing Unit (GPU) that drives the graphics card in your Linux computer.

EGL

An Application Programming Interface(API) library of interactive rendering programs that uses standard graphics drawing modules, such as OpenGL, to produce graphics.

evdev

An input event processing component of the Linux kernel, that takes events from devices like a USB mouse, and passes them to the Weston compositor.

Gnome Clutter, GTK+, and Qt

Examples of C or other language graphics library toolkits, that you can use to write graphics-based client application programs. All of them work with EGL to address the underlying layers of Figure W28.2. What is a “toolkit”? Simply a pre-packaged, sometimes graphically front-ended, collection of software routines used to produce graphical objects. Examples of toolkits are Glade and Qt-Creator. An example of a graphical object that these two toolkits produce is a Window Gadget (widget.) A vast majority of the graphics you see on-screen when using a desktop system such as Gnome, KDE, or Cinnamon in Linux Mint, are widgets.

KMS

Kernel Mode Setting, which allows setting of display resolution of the video card in kernel space rather than in user space. Allows for very fast switching between screen resolutions.

Mesa

A closer-to-the-hardware library of interactive rendering programs that uses EGL to directly render to the screen display using the DRM.

In Figure W28.2, there are three client applications, that have used Gnome Clutter, GTK+, and QT for their graphics.

The most efficient and time effective way of writing client applications for Wayland is to use the above three “toolkit” libraries, or another similar one of your choice, depending upon what your design goals are, and how familiar you are with using the toolkit. We show some basic examples of using the GTK+ and Qt toolkits, for Wayland-based C and C++ programs, in Sections W28.5 and W28.6.

The client-server/event-request interaction between these client applications and the Wayland Compositor software, Weston, can be described as follows-

1. The evdev software module in the Linux kernel receives an event from the hardware along the USB bus, perhaps a mouse click on a USB-mouse. It sends this event to Weston using the proper Wayland protocol.
2. Weston looks through what is known as its “scenegraph” to determine which window on the screen display the event should apply to. The scenegraph is a display listing of what is on screen, and Weston can apply appropriate affine transformations to the elements in the scenegraph. Thus, Weston can pick the right window, and transform the overall screen coordinates to a specific set of window coordinates.
3. When Weston passes the event to the client, the client then performs computations, perhaps requiring an update of the screen display window that the client is rendering in. The rendering in the window is done by the client via software known as EGL, and the client just sends a request to Weston to indicate what was updated.
4. These requests, collected by Weston (as what are known in modern jargon as “damage” requests) from its clients and then re-composites the entire screen display based upon the damage requests received from the clients.

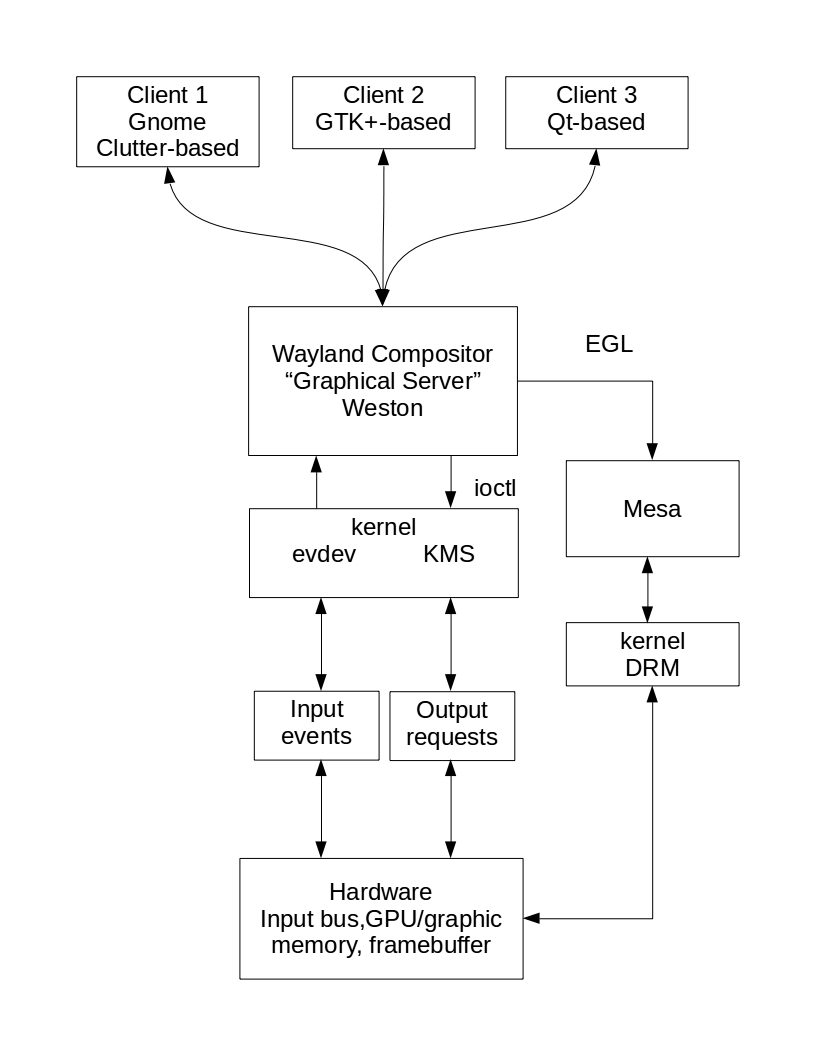


Figure W28.2 Client and Server Topologies.

In addition to the event/request pipeline via EGL shown in Figure W28.2, it is possible for client applications to directly render to the frame buffer in hardware that controls the screen display, using DRM and Mesa. This path through Weston is shown on the right-hand side of Figure W28.2, but could also connect the clients to Mesa directly, bypassing Weston.

In-Chapter Exercise W28.1

Is your Linux system running Wayland and Weston? How can you most easily find that out while you are sitting in fron of the system and using it? You can use these two commands-

**echo $DESKTOP\_SESSION**

**echo $XDG\_SESSION\_TYPE**

To fulfill the requirements of this exercise, use these two commands at the start of every one of the sessions in the sections of this chapter, particularly before you start Weston with the commands we show in Sections W28.4, and in Sections W28.5,6,and 7.

Then, make a listing of the output of the two commands on your Linux system, and from what section you obtained those results. Finally, and most critically, explain why you got those results, and what you think they mean in terms of running Wayland and Weston on your system.

W28.4 Weston

Weston is a prototype of a Wayland compositor that has been developed by the Wayland project. It is written in C, and has a very free and unrestricted license. For the time being, Weston only has official support for the Linux operating system because of its dependence on critical features of the Linux kernel. These features are kernel mode-setting, Graphics Execution Manager (GEM), and udev, and have not been implemented fully in other Unix-like operating systems, such as FreeBSD. When running on Linux, handling of the input hardware relies on evdev, as shown in Figure W28.2.

Weston has support in all three of the major distribution branches of Linux, Debian, Slackware, and RedHat/CentOS, although it has found the most support in the Debian family, Debian, Ubuntu, and Linux Mint.

W28.4.1 Ways to Launch Weston

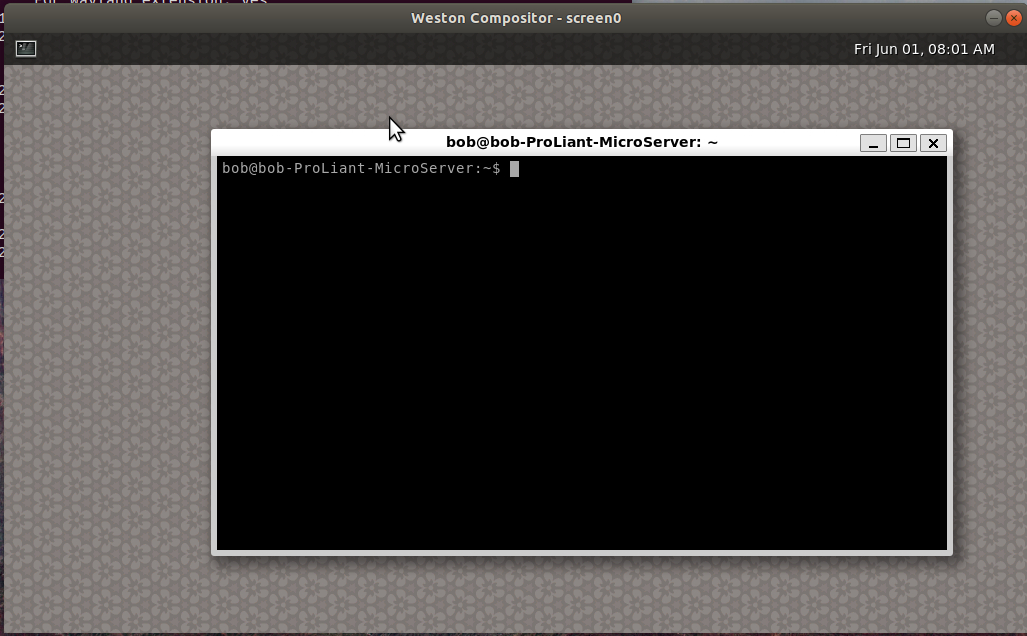
On our Wayland reference distributions of Linux systems, Ubuntu 17.10, Ubuntu 18.04, and Linux Mint 18.3, there are basically two ways to launch and use the Weston compositor.

*Weston Launch Way 1*

The first, and simplest way (and the way we use in all of our illustrations of Weston), is to first install Weston on those systems, using the following command:

$ **sudo apt-get install weston**

Then in a terminal window, type **weston**



*Figure W28.3 Weston Screen Display with Terminal*

A new window will open on screen, as seen in Figure W28.3. You can then launch a terminal inside of the Weston compositor display by clicking on the icon in the upper-left corner of the Weston display, as seen in Figure W28.3. You will be able to type Linux commands in this terminal window, to launch applications that will display their results in the Weston display.

To close or end this Weston session, just click on the Destroy Window button (an X) in the upper right-hand corder of the

*Weston Launch Way 2*

The second, slightly more complicated way, is to install Weston with the **sudo apt-get install weston** command, and then to use systemd to change the target state of your Linux computer to the non-graphical, multi-user state. This is done as is shown in Chapter 18 in the printed book with the **systemctl isolate** command, as follows:

$ **sudo systemctl isolate multi-user.target**

Then, using the keystroke combination **<Ctrl>+<Alt>+<Function Key>**, where **Function Key** can be any of the function keys F1 through F7 on your computer keyboard, you can switch to a new “virtual terminal”, log in with your username and password, and type **weston**

Your entire screen display is now taken up by the Weston compositor.

To close or end this Weston session, in a Weston terminal window, reboot the system with the following command:

$ **sudo shutdown -r now**

At the time this book was written, there was only one Linux system that was based upon only Wayland display management, and the Weston Compositor. It is called RebeccaBlackOS. We provide a screen shot of Weston on RebeccaBlackOS in Figure W28.4.

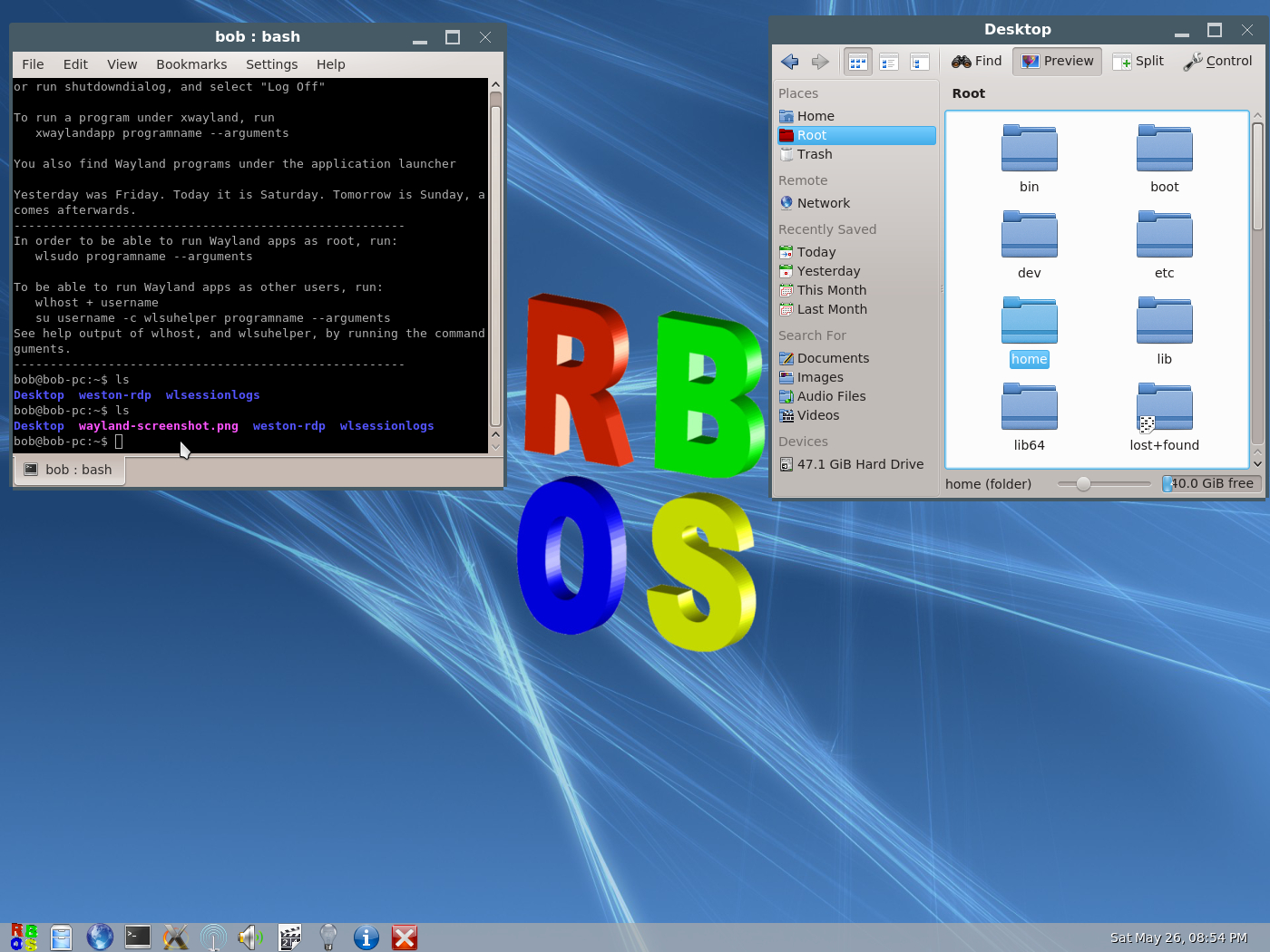


Figure W28.4 Weston on RebeccaBlackOS

W28.5 Basics of the GTK3+ Toolkit

Question: How does the GTK3+ toolkit actually work?

Answer: As stated and implied in every section of this chapter above, GUI toolkits use the event-request programming model. When there is no input by the user, the GTK3+ client application is in a wait state, waiting for input. If the user clicks a mouse button over some object on-screen, then the compositor delivers that event to the GTK3+ client application event-request loop.

When an on-screen widget receives an event, it emits a signal. The signal notifies the client application by calling functions you’ve “wired” to the signal. Such functions are known as callbacks. When callbacks are received by the client application, the client application generates a request for graphical action. An example of this event-request loop is when a button widget is clicked on, a dialog box appears in the window that the button is in. Look back at Figure W28.2 to trace the event signal, callback, and request path that evolves in this wired signaling operation. After the callback function has done its work and the compositor acts upon the damage request and has re-composited the screen, the GTK+ client application returns to the wait state.

Our purposes in this chapter are not to give you a complete and synoptic reference tutorial on GTK3+ or Qt. There are numerous documentation sources on-line and built into those frameworks that accomplish this, and many online sources and published books as well. In order for you to appreciate this fact, we pose a few In-Chapter Exercises that ask you to investigate these resources fully. But that doesn’t mean that we don’t give the beginner at least a glimpse into the inner workings of these software frameworks. We encourage further exploration and experimentation, via the built-in help systems available for each system, and the extensive on-line documentation available. This is particularly true of the Qt framework.

A good example of a line of code in a GTK3+ program (taken from example-1.c below) that conforms to the above description is:

button\_box = gtk\_button\_box\_new (GTK\_ORIENTATION\_HORIZONTAL);

or -

handler\_id = widget.connect( data)

where

button\_box ( or handler\_id) is an “instance” of a widget object that has been created,

gtk\_button\_box\_new ( or widget.connect) is the name of the callback function. It contains the code which generates the signals,

GTK\_ORIENTATION\_HORIZONTAL ( or data) the data which is passed when the signal is issued.

The objective of this section is to give a very basic and directed overview of the GTK3+ software framework. This direction is is based on the simple model we described in Section W28.1. To reiterate, the model can be summarized as follows:

Data Generation → Window Generation/Construction → Data Mapping to the Constructed Window.

The primary purpose of conforming to this model is that it portrays a very practical and useful method for using and applying a Wayland-based GUI to the beginner. As in the previous sections of this chapter, it does not require that you have any more than just a basic computer programming knowledge of the C (or in the case of Section W28.6, the C++) programming language.

It is important to realize that both GTK3+ and Qt can be used to create very basic GUI elements, such as a single window that is empty, up to entire window management systems, such as Gnome and KDE.

W28.5.1 Programming Examples Using GTK+

In the following sections, we present basic examples of using the GTK+ toolkit to construct widgets that can be deployed by a client application.

Be aware that we did these examples primarily using Linux Mint 18.3.

W28.5.2 GTK3+ Framework

GTK3+ (formerly GIMP Toolkit) is a cross-platform widget toolkit for creating graphical user interfaces.

Each user interface created by GTK+ consists of “widgets”, short for window gadgets. These are implemented in C using the GTK3+ Glib Object System (GObject), that provides an object-oriented “framework” for the C language. As we show below in the example code, the GTK3+framework allows an ordinary user to customize the GUI’s they create using their own user-written code on top of the code already provided by the GObject framework. GObject is a software library providing a portable object system, and transparent cross-language interoperability. GObject is used for both directly in programs to provide object-oriented C-based APIs, and through language library bindings to other languages, for example with PyGObject to Python 3.

It is one of the most popular toolkits for the Wayland, along with the Qt (pronounced “cute”) framework.

W28.5.3 Installing GTK3+

On our reference Wayland Linux system, Ubuntu 17.10 Desktop, GTK+ was already installed. To install this framework on other Debian-family systems, such as Linux Mint 18.3, use the following command:

$ **sudo apt-get install libgtk-3-dev**

Also, in order to compile C programs on Debian-family Linux Mint 18.3, you must first install as the root user the build-essential package (as described in Appendix A of the printed book) with the following command:

**$ sudo apt-get update && apt-get install build-essential**

To install GTK+ on CentOS 7.4 , first install the epel “extras” package if you haven’t already done this as described in Appendix A of the printed book. Then use the following command as the root user to install GTK3+:

# **yum install gtk3**

W28.5.4 GTK3+ Basic Programming Examples

Widgets are organized in a hierarchy. The window widget is the main container. The user interface is then built by adding buttons, drop-down menus, input fields, and other widgets to the window. If you are creating complex user interfaces it is recommended to use GtkBuilder and its GTK-specific markup description language, instead of assembling the interface manually. You can also use a visual user interface editor, like Glade.

GTK+ is very similar to the Wayland protocol structure, in that it is event-driven. The toolkit listens for events on the Wayland compositor, such as a mouse click, and passes the event notification to your client application.

This section contains some tutorial information to get you started with GTK+ programming. It assumes that you have GTK+, its dependencies and a C compiler installed as noted in Section W28.5.1. If you need to build GTK+ itself first, refer to the Compiling the GTK+ in Section W28.5.4.

To begin our introduction to GTK, we'll start with a simple signal-based Gtk application. This program will create an empty 250 × 250 pixel window, with “window dressing” provided by Wayland and Weston. Window dressing consists of the window frame, with buttons for destroying the window, iconifying it, or minimizing/maximizing it.

**GTK Example 0**

Create a new file with your favorite text editor, named gtk0.c , that has the following content:

#include <gtk/gtk.h>

static void

activate (GtkApplication\* app,

gpointer user\_data)

{

GtkWidget \*window;

window = gtk\_application\_window\_new (app);

gtk\_window\_set\_title (GTK\_WINDOW (window), "A Blank Window");

gtk\_window\_set\_default\_size (GTK\_WINDOW (window), 200, 200);

gtk\_widget\_show\_all (window);

}

int

main (int argc,

char \*\*argv)

{

GtkApplication \*app;

int status;

app = gtk\_application\_new ("org.gtk.example", G\_APPLICATION\_FLAGS\_NONE);

g\_signal\_connect (app, "activate", G\_CALLBACK (activate), NULL);

status = g\_application\_run (G\_APPLICATION (app), argc, argv);

g\_object\_unref (app);

return status;

}

You can compile the program above with GCC on Debian-family systems (such as Linux Mint 18.3, as we did) using the following command:

**gcc -Wall gtk0.c -o gtk0 $(pkg-config --cflags --libs gtk+-3.0)**

For more information on how to compile a GTK+ application, please refer to Section W28.5.4, Compiling GTK+ Applications. Also refer to the appropriate GTK3+ manuals and online documentation for compiling on your Linux system.

We ran this program on our Linux Mint 18.3 system, inside of the Weston-enabled window, with the following command:

$ **./gtk0**

and we got the following output-

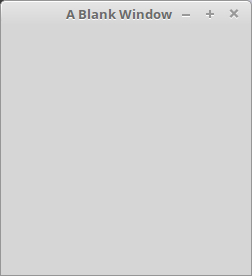


Figure W28.5 Blank Window created by GTK3+

In-Chapter Exercise W28.2

Consulting the available online and printed documentation for GTK3+, give a complete description of the functions, objects, arguments, and signal passing mechanisms that are found in program gtk0.c.

The following example is slightly more complex, adding a button to the window, and tries to showcase some additional useful capabilities of GTK+.

**GTK Example 1. C Implementation of a “GTK+ in Weston” window in GTK+**

Create a new file named gtk1.c, with your favorite text editor, that has the following content:

#include <gtk/gtk.h>

static void

print\_hello (GtkWidget \*widget,

gpointer data)

{

g\_print ("GTK+ in Weston\n");

}

static void

activate (GtkApplication \*app,

gpointer user\_data)

{

GtkWidget \*window;

GtkWidget \*button;

GtkWidget \*button\_box;

window = gtk\_application\_window\_new (app);

gtk\_window\_set\_title (GTK\_WINDOW (window), "Sophisticated Window");

gtk\_window\_set\_default\_size (GTK\_WINDOW (window), 250, 250);

button\_box = gtk\_button\_box\_new (GTK\_ORIENTATION\_HORIZONTAL);

gtk\_container\_add (GTK\_CONTAINER (window), button\_box);

button = gtk\_button\_new\_with\_label ("GTK+ in Weston");

g\_signal\_connect (button, "clicked", G\_CALLBACK (print\_hello), NULL);

g\_signal\_connect\_swapped (button, "clicked", G\_CALLBACK (gtk\_widget\_destroy), window);

gtk\_container\_add (GTK\_CONTAINER (button\_box), button);

gtk\_widget\_show\_all (window);

}

int

main (int argc,

char \*\*argv)

{

GtkApplication \*app;

int status;

app = gtk\_application\_new ("org.gtk.example", G\_APPLICATION\_FLAGS\_NONE);

g\_signal\_connect (app, "activate", G\_CALLBACK (activate), NULL);

status = g\_application\_run (G\_APPLICATION (app), argc, argv);

g\_object\_unref (app);

return status;

}

To familiarize you with another way of compiling a GTK+ C program on Debian-family systems (such as Linux Mint 18.3 as we have done), compile the program above with GCC using the following command:

**gcc `pkg-config --cflags gtk+-3.0` gtk1.c -o gtk1 `pkg-config --libs gtk+-3.0`**

We ran this program on our Linux Mint 18.3 system, inside of the Weston-enabled window, with the following command:

$ **./gtk1**

and we got the following output-



Figure W28.6 Sophisticated Window with Button

As initially stated as the answer to our preliminary question in this section, and implied by our conclusion to the commentary of Example 0.:

*GTK3+ works using the event-request model by passing signals as pointers or data between graphical widget components of your designed GUI, and the data generating modules and the underlying hardware drivers that are available on your system that display graphics on-screen.*

In-Chapter Exercise W28.3

Consulting the available online and printed documentation for GTK3+, give a complete description of the functions, objects, arguments, and signal passing mechanisms that are found in program gtk1.c.

If you have seen our presentation of Python 3 in Chapter W19 at this book’s website, you can appreciate that GTK3+ code can be “bound” to Python as well as to C. Similar to Example 1, we provide a Python 3 version that accomplishes the creation of a simple window and with the additional feature of having a built-in “destroy window” button in it.

**GTK Example 2. Python 3 Implementation of a “GTK in Weston” window in GTK+**

Create a new file named example-2.py, with your favorite text editor, that has the following content:

import gi

gi.require\_version('Gtk', '3.0')

from gi.repository import Gtk

window = Gtk.Window(title="GTK in Weston")

window.connect("destroy", lambda w: Gtk.main\_quit())

window.add(Gtk.Label("GTK in Weston"))

window.show\_all()

Gtk.main()

To run this code in Python 3, use the following command:

$ **python3 example-2.py**

We get the following output on-screen when we run example-2.py on our Linux Mint 18.3 system-



Figure W28.7 Python GTK3+ Window

W28.5.5 GTK3+ Compilation Appendix

To compile a GTK+ application, you need to tell the compiler where to find the GTK+ header files and libraries. You can do this with the pkg-config utility.

The following command line session, done on Linux Mint 18.3, illustrates how pkg-config is used (the actual output on your system may be different):

$ pkg-config --cflags gtk+-3.0

-pthread -I/usr/include/gtk-3.0 -I/usr/lib64/gtk-3.0/include -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include -I/usr/include/pixman-1 -I/usr/include/freetype2 -I/usr/include/libpng12

$ pkg-config --libs gtk+-3.0

-pthread -lgtk-3 -lgdk-3 -latk-1.0 -lgio-2.0 -lpangoft2-1.0 -lgdk\_pixbuf-2.0 -lpangocairo-1.0 -lcairo -lpango-1.0 -lfreetype -lfontconfig -lgobject-2.0 -lgmodule-2.0 -lgthread-2.0 -lrt -lglib-2.0

The simplest way to compile a program is to use "backticks", or accent grave. If you enclose a command in backticks (not single forward quotation marks!!), then its output will be substituted into the command line before execution. So to compile a GTK+ gtkx.c program, you would type the following:

$ gcc `pkg-config --cflags gtk+-3.0` gtkx.c -o gtkx `pkg-config --libs gtk+-3.0`

W28.6 Using the Qt Toolkit on Linux Mint 18.3

As seen in Figure W28.2, there are basically two ways a Wayland client application program can use its program code to work in conjunction with the Wayland protocol and the Weston compositor. This can be done directly by using low-level drawing functions, or by using a toolkit specifically designed to act as a simple-to-use intermediary that minimizes the coding complexity of dealing with Wayland protocol and those lower-level drawing functions. Qt, like GTK3+, is just such an intermediary toolkit. It is a “framework”, or programming environment, that allows an ordinary user to modify the pre-built Qt system with user-written C++ code, to create a customized Wayland client application program. Of course, and easier and more user-friendly approach to using the Qt toolkit is to use a graphical front end Integrated Development Environment (IDE) to it, known as Qt Creator. Either by programming with the toolkit in C++, or using Qt Creator, this framework abstraction frees the user from dealing with the lower level details of the Wayland protocol, programming in C or C++, or addressing the lower-level drawing functions in modules like EGL, Mesa, Cairo, Pango, etc. . That is basically why the toolkit frameworks were developed extensively by the interested parties, and made freely available to developers.

Note again that we did everything in the sections on Qt using Linux Mint 18.3, but also show some information for CentOS 7.5.

In the following sub-sections and exercises, we show the following:

\* How to install the Qt framework, and Qt Creator, on Debian-family Linux Mint 18.3 and CentOS 7.5.

\* A basic overview of how to do Qt C++ programming from the command line.

\* Describe and explain the basics of how to use Qt Creator, the graphical front end IDE for Qt.

Be aware that the prerequisite for this section is a rudimentary knowledge of C++ Object Oriented Programming (OOP) concepts and its syntax, in order for you to appreciate the concepts involved in Qt programming.

W28.6.1 Some Preliminary Programming Considerations

It is essential that you have the Gnu C Compiler (gcc), and a text editor such as nano, vi or emacs installed on your system.

Even though the Qt framework is based upon the OOP model of C++, and uses the syntax of that language, it also extends the syntax to include some Qt-specific constructs and functionality. For example, the Signals and Slots features of Qt differ from and extend the C++ class model.

W28.6.2 Installing the Qt Framework with Qt Creator, and Obtaining Help

In order to compile C++ programs on Debian-family Linux Mint 18.3, as the root user you must first install the build-essential package (as described in Appendix A of the printed book) with the following command:

**$ sudo apt-get update && apt-get install build-essential**

To begin to work with Qt, you must first download and install it. This is done most effectively using the graphical software package manager system on your Linux system. On our Debian-family Linux Mint 18.3 system, we used the Software manager to download and install the Qt-sdk package. This package included the core toolkit, and also the Qt Creator Integrated Development Environment (IDE).

To install Qt on CentOS 7.4 , as the root user, first install the epel “extras” package if you haven’t already done this as described in Appendix A of the printed book, using the following command:

**# yum install epel-release**

Then use the following command as the root user to install qt-creator:

# **yum install qt-creator**

A verbose and extensive help and documentation system, known as Qt Assistant, is also installed with the Qt-sdk package. In order to get help using Qt Assistant, you can type **assistant** on the Linux command line after you have installed the Qt-sdk package, and a GUI Qt Assistant window will open on-screen.

The main Qt Assistant window pane illustrates an Introduction to Qt Assistant. The Contents tab of the Qt Assistant window contains major help topic headings, such as the Qt Designer Manual, and other relevant major topic headings. If you click on the Search tab, and type in a topic you want to get help on, you can get a listing of the help pages available for that topic. When we typed in **signals**, a very important and relevant topic that is illustrated in the following sub-sections, Qt Assistant listed Signals and Slots as a topic area we could get help on. When we navigated to Signals and Slots help, we got a very extensive explanatory display in the main window pane, describing those two critical features of Qt.

We encourage you to take full advantage of Qt Assistant for further descriptions and more verbose and complete help on everything we present here.

W28.6.4 *Creating an Executable Qt Program from the Command Line*

Qt has its own compiling, linking, and assembling procedure, as shown. The following steps show how to use the Qt procedure to compile a Qt program, create a Qt project, and execute a client application program on our Linux Mint system.

You do not use the GNU C++ compiler directly to do any of these operations.

Step 0: Create an empty directory under your home directory, with a name like **qtprogs1**. Make that directory the present working directory. Use a text editor of your choice to enter and save the Qt code of any of the example exercises in Section W28.6.5.4 into a file with the file extension **.cpp**—for example, **exercise1.cpp**,which contains Qt client application program code.

Step 1: At the shell prompt, type **qmake -project**

Step 2: At the shell prompt, type **qmake**

Step 3: At the shell prompt, type **make**

Step 4. In the directory you will now have the following files in the directory qtprogs1:

**exercise1.cpp**, **exercise1.o,** **Makefile**, **qtprogs1.pro**, and **qtprogs1**.

Step 5: At the shell prompt, type **./qtprogs1**

Step 6: The graphics contained in the Qt program you entered into **exercise1.cpp** are now shown on the screen. Use the kill window button to close this Qt window.

W28.6.5.4 Qt C++ Programming Examples

Here is the simplest Qt client application program you can enter. Follow the instruction steps in Section W28.6.5.3 to create the program and its accompanying project modules, and then execute it.

**Qt Exercise W28.1: Simplest Qt Program**

// helloLinux/main.cpp

#include <QApplication>

#include <QLabel>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

QLabel label("Hello Linux");

label.show();

return a.exec();

}

Follow the instruction steps in Section W28.6.5.3 to create the program and its accompanying project modules, and execute it on your Linux system.

**Qt Exercise W28.2: Simple Qt Program with a GUI**

In this example, we create and show a built-in Qt text edit capability in an open window. This represents a simple Qt program that has a GUI.

Here is the code:

#include <QApplication>

#include <QTextEdit>

int main(int argv, char \*\*args)

{

QApplication app(argv, args);

QTextEdit textEdit;

textEdit.show();

return app.exec();

}

In-Chapter Exercise W28.4

Consulting the available online and printed documentation for Qt, give a complete description of the functions, objects, arguments, and signal passing mechanisms that are found in program Qt Exercise W28.2. Then follow the instruction steps in Section W28.6.5.3 to create the program and its accompanying project modules, and execute it on your Linux system.

**Qt Exercise W28.3: Adding a Quit Button**

In a real application, you would usually create more than one widget to allow a rich and varied dialog between client application data-generating code and Qt code. We will now show a simple example of a QPushButton beneath the text edit window created in Qt Exercise W28.2. The button will exit the QTextEdit application when pushed (i.e., clicked on with the mouse).

Here is the code:

#include <QtGui>

int main(int argv, char \*\*args)

{

QApplication app(argv, args);

QTextEdit \*textEdit = new QTextEdit;

QPushButton \*quitButton = new QPushButton("&Quit");

QObject::connect(quitButton, SIGNAL(clicked()), qApp, SLOT(quit()));

QVBoxLayout \*layout = new QVBoxLayout;

layout->addWidget(textEdit);

layout->addWidget(quitButton);

QWidget window;

window.setLayout(layout);

window.show();

return app.exec();

}

In-Chapter Exercise W28.5

Consulting the available online and printed documentation for Qt, give a complete description of the functions, objects, arguments, and signal passing mechanisms that are found in program Qt Exercise W28.3. Then follow the instruction steps in Section W28.6.5.3 to create the program and its accompanying project modules, and execute it on your Linux system.

**Qt Exercise W28.4: Connecting Signals and Slots**

The following Qt code places three widgets in a window, and defines interconnections between the signal elements and slot elements of those widgets.

// signalSlot2/main.cpp

#include <QApplication>

#include <QVBoxLayout>

#include <QLabel>

#include <QSpinBox>

#include <QSlider>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

QWidget window;

QVBoxLayout\* mainLayout = new QVBoxLayout(&window);

QLabel\* label = new QLabel("0");

QSpinBox\* spinBox = new QSpinBox;

QSlider\* slider = new QSlider(Qt::Horizontal);

mainLayout->addWidget(label);

mainLayout->addWidget(spinBox);

mainLayout->addWidget(slider);

QObject::connect(spinBox, SIGNAL(valueChanged(int)),

label, SLOT(setNum(int)));

QObject::connect(spinBox, SIGNAL(valueChanged(int)),

slider, SLOT(setValue(int)));

QObject::connect(slider, SIGNAL(valueChanged(int)),

label, SLOT(setNum(int)));

QObject::connect(slider, SIGNAL(valueChanged(int)),

spinBox, SLOT(setValue(int)));

window.show();

return a.exec();

}

Follow the instruction steps in Section W28.6.5.3 to create the program and its accompanying project modules, and execute it on your Linux system.

W28.6.5 Creating a Simple Widget Using Qt Creator

Now that you have seen how to create basic Qt projects and programs from the command line, we present a simpler and more intuitive technique to accomplish the same thing using a GUI.

Qt Creator is an IDE which is used, via a very developed and complete GUI interface, to access and control the Qt software application development framework. It includes a visual debugger and an integrated GUI layout and forms designer, and many other extensive toolkit additions to help expedite project work. The Qt Creator on-line Manual gives extensive tutorials and help on every aspect of Qt Creator.

Practice Session W28.1 Empty Widget Creation in Qt Creator

Objectives: To illustrate how to create a basic Qt Creator project, and place an empty (blank window) widget in that project.

Our purpose in creating an empty widget is to be able to use an additional application, gnuplot, which is described fully in Section W28.7, to fill that empty widget with plots of data without having to program the plotting in C++.

Pre-Requisites: Having installed Qt-sdk along with Qt Creator on Linux Mint, as shown in Section W28.6.2. Or on CentOS systems, installing qt-creator as shown in Section W28.6.2. Having installed gcc according to the instructions in Appendix A of the printed book. Using the **mkdir** command to make a project directory for the Qt Creator project files we are about to create. We named our project directory “qt3”, which will play into the Commentary instructions for CentOS 7.5 below.

No previous knowledge of either Qt, Qt Creator, or C++ is required for this Practice Session.

Background: This Practice Session allows you to do window creation, in conformance to our development model of Data Generation→Window Creation→ Data Mapping. It steps you through exactly how to start a new Qt Creator project, and create an empty widget. To best see the full extent of capabilities that the Qt framework provides, you can view all of the Examples made available in the Qt Creator Project Window, shown as a button choice in Figure W28.8.

Requirements: Do the following steps, in the order presented.

Step 0. If you have not already done so after the installation shown in Section W28.6.2, place the Qt Creator icon on your desktop so you can launch it easily.

Commentary- You can do this on Debian-family systems, on CentOS 7.5 make the Applications menu choice Programming> Qt Creator.

Step 1. Launch Qt Creator. The Main Qt Project Window appears on screen, as seen in Figure W28.8

Commentary-

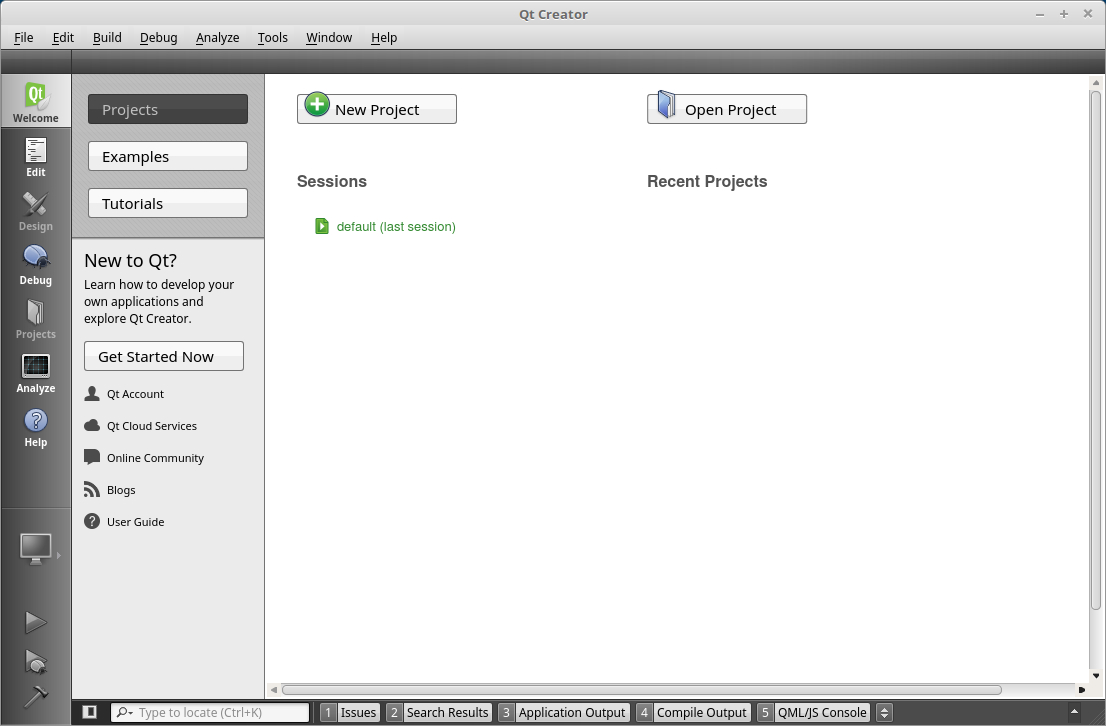


Figure W28.8 Main Qt Creator Project Screen

Step 2. Click on the widget entitled “New Project”. The New Project window opens on screen. The Application and Qt Widgets Application choices are highlighted by default. Click on the Choose... button in the lower-right hand side of the New Project Window.

Commentary- You are creating the new project with a default project type.

Step 3. The Qt Widgets Application window opens on screen. Type in the name you want to give the project, and the folder where you want to create project components in. Click on the Next> button in the lower-right hand side of this window.

Commentary- Read the screen-displayed Introduction. We named our project “t3”. In section W28.4.7, we are going to map gnuplot graphics in the empty widgets created by Qt Creator.

As stated in the prerequisites, we strongly advised you to create a project directory, under your home directory, to hold the project components before you launched Qt. Our project directory, where Qt will build everything, is named “qt3”, as specified in the prerequisites to this example.

Step 4. The Kit Selection screen opens on screen. Click on the Next> button in the lower-right hand side of this window.

Commentary- On Debian-family Linux Mint 18.3, we left “Select all kits” as the default. On CentOS 7.5, we did the following-

1. On the Qt Widgets Application screen, we chose Manage,
2. On the Options Screen, on the tab Build & Run Kits, we highlighted Manual - - > Desktop Default,
3. Make sure the compiler is GCC (x86 64-bit) in /usr/bin,
4. Click Apply, then OK,
5. Click Next>,

Step 5. The Class Information window opens on screen. Click on the Next> button in the lower-right hand side of this window.

Commentary- On Debian-family Linux Mint 18.3, we left all of the basic information about class definitions at their defaults. On CentOS 7.5, we did the same thing.

Step 6. The Project Management window opens on screen. Click on the Finish button in the lower-right hand side of this window.

Commentary- On Debian-family Linux Mint 18.3, the “Files to be added” display shows a summary of everything Qt Creator will put in your designated project directory for an empty widget, without you having to write any C++ code!

Step 7. The mainwindow.cpp editor screen opens, showing you the C++ code of mainwindow.cpp.

It should contain something very similar to this-

#include "mainwindow.h"

#include "ui\_mainwindow.h"

MainWindow::MainWindow(QWidget \*parent) :

QMainWindow(parent),

ui(new Ui::MainWindow)

{

ui->setupUi(this);

}

MainWindow::~MainWindow()

{

delete ui;

}

Click the right-facing green arrow in the lower-left corner of the window display. See the Commentary for variations on this procedure for CentOS 7.5. The project builds, and an empty window, or widget, appears on screen. If you click that green arrow more than one time, you will get as many empty widgets built and displayed as the number of times you click the green arrow! You can close each one of these empty widgets by using the destroy window button in each one.

Commentary- This step achieves the objective of your basic exposure to the Qt Creator GUI.

On CentOS 7.5, we had to do the following-

1. Clicked on the Build Hammer in the lower-left of the Qt Creator window.
2. Clicked the right-facing green arrow in the lower-left corner, signifying the Run command. At this point on CentOS 7.5, we got an error!
3. In the Custom Executable window that appears on screen, as seen in Figure w28.9, we used the Browse… buttons to designate the pathname to where the executable file and working directories that Qt created for this project were. For example, the executable file is in the folder “/home/bob/qt3/build-t3-Desktop-Debug”, and it is named “t3”. Since we named the project “t3”, that will be the name of the executable file along this pathname. The Working Directory choice will automatically be chosen for you when you browse using the lower browse button, once you designate the path to the executable first! For example, it will be the folder “/home/bob/qt3/t3”. Click OK when you have made the proper choices here.
4. When you click the right-facing green arrow once more, to signify Run again, your window will appear on screen on the CentOS 7.5 desktop!

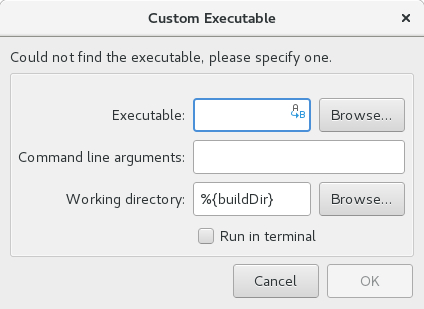


Figure W28.9 Custom Executable Window

Step 8. Quit Qt Creator by making the File> Exit menu choice in the mainwindow.cpp screen display.

Commentary- After quitting Qt Creator, if you make the current working directory the one you designated in Step 3., there will be two folders in that directory. In the folder named “build-name-Desktop-Debug”, where name is the name you supplied in Step 3., you will find an executable image “name”, where name is the name of you supplied in Step 3.. In our case it was “t3”. If you double-click on that executable image, you will create another empty widget on your desktop. You can also use a terminal and make the current working directory, in our case “/home/bob/qt3/ build-t3-Desktop-Debug’, and execute the file gnuplot with the command **./t3**

Conclusions: Qt Creator allows you to quickly and easily create a new Qt framework project, and design and place widgets as part of that project. The full extent of Qt Creator capabilities is best seen in the Qt Creator Manual.

In-Chapter Exercise W28.6

On Debian-family systems, after installing the Qt-sdk package on your system, do Practice Session W28.1. On CentOS systems, after installing qt-creator, do Practice Session W28.1.

W28.7 Gnuplot

To complete our study of GUI systems on Linux, we turn in this section to a plotting package named gnuplot. For our purposes, there is a very important reason for using gnuplot: it makes it possible to conform to a modularized idea of developing for a GUI system. You can develop a discreet module that creates data, another discreet module that creates and orchestrates the window display you want to present the data in, and then finally a discreet module that maps the data into the window display.

We encourage you to compare the particular use case we show in this section, and this modularized approach, to the specific design and programming details and methodologies that would accomplish the same end result in:

\* Wayland/Weston-based C programs,

\* “augmented” C programs for the GTK3+ framework, or

\* C++ Qt framework programs.

This model’s particular modularized use case does not replace using GTK3+, Qt, or Qt Creator as we have shown them in the previous sections of this chapter. In addition, the window(s) you create are framed and constrained to a large extent by the particular window management system you are using. For example, by default we are using Linux Mint Cinnamon, which has its own look and feel, window dressing, etc.. Everything we do here can also be done in a pure Wayland/Weston compositor environment, if you want to see the data framed in another window manager. Or for that matter, in any other legacy window manager you might want to have running on your Linux system.

Again, please be aware that we did this section using Linux Mint 18.3.

W28.7.1 Installing gnuplot

There are two ways you can install gnuplot on your Linux system, which are of course dependent upon the package management system and commands you have available.

The first way is the simplest: use the graphical software installer available on your system, and install the latest version. When we did this on Linux Mint 18.3, for example, we used the Software Manager and installed gnuplot. At the time of the writing of this book, the latest gnuplot version and patch available through the Software Manager was gnuplot5, which gave us gnuplot 5.0.3. If you want to have access to the qt terminal type, instead of installing “plain” gnuplot via the Software manager, you can install gnuplot-qt. This terminal type is useful to have when doing Section W28.7.6.

The second way allows you to download the latest version of the source code as a tar.gz file, and install it from that. Follow these steps-

1) Set your web browser to www.gnuplot.info/download.html, and you will find the latest version available as a tar.gz file at Sourceforge. Download the latest version. At the time we wrote this chapter, we were able to download gnuplot-5.0.5.tar.gz from the Sourceforge site.

2) While in the directory you downloaded or moved the tar.gz file to, on the command line, use the following commands:

$ **tar xzf gnuplot-5.0.5.tar.gz**

$ **cd gnuplot-5.0.5**

$ **./configure**

$ **make**

Output truncated...

$ **sudo make install**

Output truncated…

$

Make sure you substitute the version number in the above commands for your latest version number!

3) From anywhere on your system, type the following to launch gnuplot:

$ **gnuplot**

G N U P L O T

Version 5.0 patchlevel 5 last modified 2016-10-02

Copyright (C) 1986-1993, 1998, 2004, 2007-2016

Thomas Williams, Colin Kelley and many others

gnuplot home: http://www.gnuplot.info

faq, bugs, etc: type "help FAQ"

immediate help: type "help" (plot window: hit 'h')

Terminal type set to 'x11'

gnuplot>**exit**

**$**

In the gnuplot directory created by the tar command in step 2), there will be an INSTALL file that you can use with whatever version of gnuplot you downloaded in step 1), and which gives more specific instructions and other information about how to install the latest version of gnuplot.

W28.7.2 What gnuplot Is, and Basic Syntax for Interactive and Batch Modes

Gnuplot is a command-line data and scientific graphing package for Linux and UNIX.

Gnuplot supports many types of plots in either 2D and 3D. It can draw using lines, points, boxes, contours, vector fields, surfaces, and various associated text. It also supports various specialized plot types.

Gnuplot also supports many different types of output formats: interactive screen terminals (with mouse and keyboard input), printers, and to a variety file formats (eps, emf, fig, jpeg, LaTeX, pdf, png, postscript, etc.). Recent additions in Release 5 include wxWidgets terminals, and Qt-based graphics terminals.

The “interactive” command language is case sensitive. All command names have standardized abbreviations. Any number of interactive commands may appear on a line, separated by semicolons (;). String arguments to interactive commands may be delimited by either single or double quotes, although there are some differences in the interpretation of quoting.

Example:

gnuplot> **set title "My First Plot"; plot ’data’; print "all done!"**

Commands may extend over several input lines by ending each line but the last with a backslash (\). The backslash must be the last character on each line. In gnuplot documentation, curly braces ({}) denote optional arguments and a vertical bar (|) separates mutually exclusive choices. Gnuplot keywords or help topics are indicated by backquotes or boldface. Angle brackets (<>) are used to mark replaceable tokens. In many cases, a default value of the token will be taken for optional arguments if the token is omitted, but these cases are not always denoted with braces around the angle brackets. For built-in help on any topic, type help followed by the name of the topic or help ? to get a menu of available topics.

When run from the Linux command line in what is known as “Batch Mode”, gnuplot is launched by using the syntax:

$ gnuplot {OPTIONS} file1 file2 ...

where-

file1, file2,... are validly formatted input files for the `load` command. The `load` command executes gnuplot commands contained in file1, file2… as if each line of the specified input file(s) had been typed in interactively. Files created by the `save` command can later be `load`ed. Any text file containing valid gnuplot commands can be created and then executed by the `load` command. Files being `load`ed may themselves contain `load` or `call` commands.

On Wayland-based systems, you can use:

$ gnuplot {OPTIONS} {OPTIONS} file1 file2 …

W28.7.3 Batch Mode Examples

Gnuplot may be executed in either Batch or Interactive modes, and the two may even be mixed together in valid ways. Any Batch mode arguments are assumed to be either program options (first character is -) or names of files containing gnuplot commands. The option -e "command" may be used to force execution of a gnuplot command. Each file or command string will be executed in the order specified. The special filename "-" is indicates that commands are to be read from stdin. Gnuplot exits after the last file is processed. If no load files and no command strings are specified, gnuplot accepts interactive input from stdin.

Both the exit and quit commands terminate the current command file and load the next one, until all have been processed.

Following are some basic Gnuplot examples:

To launch an interactive session:

$ **gnuplot**

To launch a Batch session using two gnuplot command files "input1" and "input2":

$ **gnuplot input1 input2**

To launch an interactive session after an initialization file "header" and followed by another command file "trailer":

$ **gnuplot header - trailer**

To give gnuplot commands directly on the Linux command line, using the "-persist" option so that the plot remains on the screen afterwards:

$ **gnuplot -persist -e "set title ’Sine curve’; plot sin(x)"**

To set user-defined variables a and s prior to executing commands from a file:

$ **gnuplot -e "a=2; s=’file.png’" input.gpl**

W28.7.4 Batch Mode Plotting to a Terminal with Persistence of the Plot Window

gnuplot terminals open separate display windows on the screen into which plots are drawn. One of the most important plot command options is the persist option. It tells gnuplot to leave these windows open when the main program exits. It has no effect on subsequent interactive terminal output. For example if you issue the following two batch mode commands:

$ **gnuplot -persist -e ’plot [-5:5] sinh(x)’**

$ **gnuplot -persist -e ’plot [-5:5] tanh(x)’**

After the first command, gnuplot will open a display window using the default terminal type, draw the sinh plot into it, and then exit, leaving the display window containing the sinh plot on the screen. After the second command, a new display window will open, using the default terminal type, and the tanh plot will be drawn into it.

W28.7.5 Interactive Mode and Terminal Type

A basic premise of this section is that you can find out the window ID number of any window displayed by the legacy windowing system on our reference Wayland distributions Ubuntu 17.10 and 18.04, and Linux Mint 19. The procedures that allow you to do this dovetail with the presentation of creating a “blank” window with Qt Creator, as we did in Section W28.6.5. These procedures also help us to conform to our modularized model of practical GUI creation and use-

Data Generation → Window Generation/Construction → Data Mapping to the Constructed Window.

For example, we would like to know the window ID number of the window we created in Section W28.5.5 using Qt Creator. To find out the window ID number of any displayed window, use the following command:

$ **xwininfo**

xwininfo: Please select the window about which you

would like information by clicking the

mouse in that window.

xwininfo: Window id: 0x5600005 "MainWindow"

Absolute upper-left X: 69

Absolute upper-left Y: 49

Relative upper-left X: 10

Relative upper-left Y: 36

Width: 400

Height: 300

Depth: 24

Visual: 0x319

Visual Class: TrueColor

Border width: 0

Class: InputOutput

Colormap: 0x5600004 (not installed)

Bit Gravity State: NorthWestGravity

Window Gravity State: NorthWestGravity

Backing Store State: NotUseful

Save Under State: no

Map State: IsViewable

Override Redirect State: no

Corners: +69+49 -1451+49 -1451-731 +69-731

-geometry 400x300+69+49

$

The above output was obtained when a Qt Creator-created window was on-screen. We moved the crosshairs over it when prompted by xwininfo, and clicked the left mouse button. From the output, you can see that the window’s ID number is 0x5600005, and its title is "MainWindow".

If you type **gnuplot** on the Bash command line, you enter gnuplot “Interactive Mode”. Another simple example of this is as follows:

gnuplot> **plot [-5:5] tanh(x)**

Once this interactive command is given, a window opens on-screen that shows you a plot of tanh(x) with the x range varying from -5 to +5.

Without changing the default terminal type on our Linux Mint system when we installed gnuplot-qt using the Software Manager, as shown in Section W28.7.1, the terminal type was set to qt.

In a Qt terminal, there are a some menu choices seen at the top of the window, which we list here from left to right:

File Output – copy to clipboard, print, export to pdf, etc.

Replot (a green circular arrow)- replots the window

Grid- places a grid on the window

Previous Zoom (a minus signed magnifier)- goes to the previous zoom

Next Zoom (a plus sign magnifier)- goes to the next zoom

Settings- Allows changes of terminal configuration settings like background color, replot on resize, etc.

To zoom in a Qt terminal window, click the right-most mouse button once to designate a corner of the plot you want to zoom to, and then click the right-most button again to designate the opposite corner of the zoom.

In addition to persistence in both Batch and Interactive modes, setting various characteristics of the plot window are important to us in this section. The following interactive commands lets you-

a) set the terminal type to x11,

b) designate a specific display ID (0x5600005) within which to plot,

c) have that window persist after the program exits,

d) set its size to 700 x 500, and

e) plot sin(x) in that display.

Another important example of gnuplot Interactive Mode is as follows:

gnuplot> **set term x11 window “0x5600005” persist size 700,500**

Terminal type set to 'x11'

Options are 'XID 0x5600005 persist enhanced size 700,500 '

gnuplot> **plot sin(x)**

The important characteristic of using gnuplot that these two commands illustrate is that if you have previously created a “blank” window with another GUI application, which is what we have done in Section W28.4.6 with Qt Creator, and, as determined with xwininfo, it has the display ID of 0x5600005, your plot of sin(x) will be drawn in that display.

This now makes it possible to conform to our modularized idea of a development model. You can have a discreet module create the data using your favorite programming language (C, Python, C++, bash, etc.), design the window display module you want to present the data in with toolkits such as GTK3+ or Qt, and then map the data into the window with either a gnuplot batch or interactive module.

We encourage you to compare, in this particular use case, this modularized approach to the specific details and methodologies of accomplishing the same end result in either a purely C program development environment, or in an “augmented” C program for the GTK3+ framework, or in a C++ Qt framework program. Certainly this particular modularized use case does not replace using C graphics programming, GTK3+, or Qt in the situations where they excel, or are more germane to the task(s) at hand.

The strong point we are trying to make here is that using the highly-developed toolkits and gnuplot greatly facilitates the development of GUI client applications, given the modularized model we espouse in this chapter.

In-Chapter Exercise W28.7

In gnuplot, type **set terminal**

You will get a listing of the terminal types available for you to set for your installation of gnuplot.

The terminal types actually available to you are dependent upon what driver library packages you have installed previous to installing gnuplot!

Experiment with setting the terminal to some of the different types shown by the set terminal command in gnuplot.

W28.7.6 Plotting in Interactive Mode

There are four fundamental gnuplot commands that create plots: plot, splot, replot, and refresh. Plot generates 2D plots, splot generates 3D plots ( 2D projections onto a picture plane). Replot appends its arguments to the previous plot or splot and executes the modified command. Refresh re-executes the previous plot or splot command using previously stored data rather than rereading data from a file.

W28.7.6.1 Plotting Data Contained in a File

Data contained in a file can be plotted with plot or splot by specifying the name of

the properly-formatted data file (enclosed in single or double quotes) on the `plot` command line.

Syntax:

plot '<file\_name>' {binary <binary list>}

{{nonuniform} matrix}

{index <index list> | index "<name>"}

{every <every list>}

{skip <number-of-lines>}

{using <using list>}

{smooth <option>}

{volatile} {noautoscale}

A short explanation of the arguments is- `binary` allows data entry from a binary file, `index` selects which data sets in a multi-data-set file are to be plotted, `every` specifies which points within a single data set are to be plotted, `using` determines how the columns within a single line are to be interpreted, and `smooth` allows for simple interpolation and approximation. `splot` has a similar syntax, but does not support the `smooth` option.

Data files should contain at least one data point per line (`using` can select one data point from the line). Lines beginning with `#` will be treated as comments and ignored.

Each data point represents an (x,y) pair. For `plot`s with error bars or error bars with lines , each data point is (x,y,ydelta), (x,y,ylow,yhigh), (x,y,xdelta), (x,y,xlow,xhigh), or (x,y,xlow,xhigh,ylow,yhigh).

If line numbering is present at the beginning of each line of a data file, that numbering must be separated by white space (one or more blanks or tabs) from the remainder of the data on that line, unless a format specifier is provided by the `using` option. This white space divides each line into columns. However, whitespace inside a pair of double quotes is ignored when counting columns, so the following datafile line has three columns:

1.0 "second column" 3.0

A data file can contain only one column (the y value). If x is omitted, gnuplot

provides integer values for x, starting at 0.

In-Chapter Exercise W28.8

Create a data file of interest to you with your favorite text editor, containing at least 20 paired x-y data points. Then use gnuplot to read that data in from the file, and plot it in a Qt Creator-created window.

In-Chapter Exercise W28.9

Plot the following functions in a single, persistent Qt Creator-created window: 2tan(x), sin(x), cos (x), tanh(x).

W28.7.6.2 Plotting Styles

There are many 2-D and 3-D plotting styles available in gnuplot, which have evolved over the long history of gnuplot use on both UNIX and Linux systems. We give a listing of them as they pertain to a particular kind of graphical presentation method (if applicable), and then give a few descriptions of the most important ones. For example, he commands **set style data** and **set style function** change the default plotting style for subsequent the plot and splot commands.

You can also specify the plot style explicitly as part of the plot or splot command. If you want to mix plot styles within a single plot, you must specify the plot style for each component.

An example of this would be as follows:

gnuplot> plot ’statistics’ with boxes, sin(x) with lines

Each plot style has its own expected set of data entries in a data file. For example, by default the lines style expects either a single column of y values (with implicit x ordering) or a pair of columns with x in the first and y in the second.

Descriptive Statistics- Boxerrorbars, Boxes, Boxplot, Boxxyerrorbars, Candlesticks, Circles, Ellipses,

Dots, Histograms, Newhistogram, Xerrorbars, Xyerrorbars, Yerrorbars, Xerrorlines, Xyerrorlines, Yerrorlines.

General Graphics- Filledcurves, Fillsteps, Histeps, Image, Impulses, Labels, Lines, Linespoints, Parallelaxes, Points, Polar, Steps, Vectors, 3D (surface) plots, 2D projection.

W28.7.6.3 Obtaining Help on Important Commands

In Interactive mode, if you type help **topic**, where **topic** is any of the major or sub-topics listed in Table W28.1, you will get verbose help on that topic. We list all major topics in gnuplot help, and their major sub-topics for your convenience here.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bind | call | cd | clear | do |
| evaluate | exit | fit | help | history |
| if | import | load | lower | pause |
| plot |  |  |  |  |
|  | acsplines | axes | bezier | binary |
|  | cnormal | csplines | cumulative | data |
|  | datafile | errorbars | errorlines | every |
|  | example | for | frequency | functions |
|  | index | kdensity | mcsplines | parametri |
|  | ranges | sampling | sbezier | smooth |
|  | special-filnames | style | thru | title |
|  | uniques | unwrap | using | volatile |
|  | with |  |  |  |
| print | printerr | pwd | quit | raise |
| refresh | replot | reread | reset | save |
| set-show |  |  |  |  |
|  | angles | arrow | autoscale | bars |
|  | bmargin | border | boxwidth | cbdata |
|  | cbdtics | cblabel | cbmtics | cbrange |
|  | cbtics | clabel | clip | cntrlabel |
|  | cntrparam | color | colorbox | colorsequence |
|  | contour | dashtype | decimalsign | dgrid3d |
|  | dummy | encoding | fit | fontpath |
|  | format | grid | hidden3d | history |
|  | historysize | isosamples | key |  |
|  | linetype | link | lmargin | loadpath |
|  | locale | logscale | macros | mapping |
|  | margins | monochrome | mouse | multiplot |
|  | mx2tics | mxtics | my2tics | mytics |
|  | mztics | object | offsets | origin |
|  | output | palette | parametric | paxis |
|  | pm3d | pointintervalbox | pointsize | polar |
|  | print | psdir | raxis | rmargin |
|  | rrange | rtics | samples | size |
|  | style | surface | table | terminal |
|  | termoption | tics | ticscale | ticslevel |
|  | timefmt | timestamp | title | tmargin |
|  | trange | urange | view | vrange |
|  | x2data | x2dtics | x2label | x2mtics |
|  | x2range | x2tics | x2zeroaxis | xdata |
|  | xdtics | xlabel | xmtics | xrange |
|  | xtics | xyplane | xzeroaxis | y2data |
|  | y2dtics | y2label | y2mtics | y2range |
|  | y2tics | y2zeroaxis | ydata | ydtics |
|  | ylabel | ymtics | yrange | ytics |
|  | yzeroaxis | zdata | zdtics | zero |
|  | zeroaxis | zlabel | zmtics | zrange |
|  | ztics | zzeroaxis |  |  |
| shell |  |  |  |  |
| splot |  |  |  |  |
|  | binary | datafile | errorbars | errorlines |
|  | example | for | grid\_data | parametric |
|  | ranges | style | surfaces | title |
|  | with |  |  |  |
| stats | system | test | undefine | unset |
| update | while |  |  |  |

Table W28.1 gnuplot Command Help Matrix

W28.7.6.4 qt and X11 Terminals

In this sub-section, we describe the two most important terminal types that are useful in gnuplot for Linux, and also for our modularized model. They are the Qt and X11 terminals.

W28.7.6.5 qt Terminal Type

The qt terminal device generates output in a separate window with the Qt library.

Syntax:

set term qt {<n>}

{size <width>,<height>}

{position <x>,<y>}

{title "title"}

{font <font>} {{no}enhanced}

{dashlength <dl>}

{{no}persist} {{no}raise} {{no}ctrl}

{close}

{widget <id>}

W28.7.6.6 X11 Terminal Type

The mainstay terminal type useful for our modularized model is X11.

Syntax:

set terminal x11 {<n> | window "<string>"}

{title "<string>"}

{{no}enhanced} {font <fontspec>}

{linewidth LW}

{{no}persist} {{no}raise} {{no}ctrlq}

{{no}replotonresize}

{close}

{size XX,YY} {position XX,YY}

set terminal x11 {reset}

W28.7.6.7 Plotting in Multiple Windows or Multiple Graphs in One Window

You can plot graphs in multiple discreet windows by setting the current terminal to x11 window <n> , which outputs the graph to window number n.

Most importantly, as shown in that section, the gnuplot specification of an x11 terminal can connect to X windows previously created by an outside application via the window option.

The window option requires a string containing the X ID for the window in hexadecimal format as an option argument. Gnuplot uses that external X window as a container. In this way, gnuplot’s mouse features work within the contained plot window. To repeat the example of setting the terminal type interactively:

gnuplot> **set term x11 window "0x360001"**

A plot window created in this way can be closed by pressing the letter q while that window is the active window ( mouse cursor rolled into it), or by closing it using the desktop manager window border destroy window button.

Additionally, consider the following gnuplot Batch mode program:

**set multiplot layout 2,2 rowsfirst**

**# --- GRAPH sin**

**set label 1 'x' at graph 0.92,0.9 font ',8'**

**plot sin(x) with lines ls 1 dt 2**

**# --- GRAPH cos**

**set label 1 'y' at graph 0.92,0.9 font ',8'**

**plot cos(x) with lines ls 1 dt 3**

**# --- GRAPH tan**

**set label 1 'z' at graph 0.92,0.9 font ',8'**

**plot tan(x) with lines ls 1 dt 4**

**# --- GRAPH hyperbolic tan**

**set label 1 't' at graph 0.92,0.9 font ',8'**

**plot tanh(x) with lines ls 1 dt 5**

**unset multiplot**

It uses the **multiplot** option of the **set** command to “tile” the single current window into a 2 by 2 matrix, and then proceeds to plot graphs of sin, cos, tan, and tanh in each of the tiles of that single window. Also, it uses the set command to place labels on each graph, and the **with** option of the **plot** command to customize the line type and color of each graphed function.

The syntax of the multiplot option is as follows:

set multiplot

{ title <page title> {font <fontspec>} {enhanced|noenhanced} }

{ layout <rows>,<cols>

{rowsfirst|columnsfirst} {downwards|upwards}

{scale <xscale>{,<yscale>}} {offset <xoff>{,<yoff>}}

{margins <left>,<right>,<bottom>,<top>}

{spacing <xspacing>{,<yspacing>}}

In-Chapter Exercise W28.10

To get a better idea of the capability of plotting multiple graphs in a single window, in gnuplot, type **help multiplot**.

W28.8 Conclusion

In this chapter, we presented the following model, for developers, of a practical GUI client application workflow-

Data Generation → Window Generation/Construction → Data Mapping to the Constructed Window

We gave a basic overview of how graphics rendering on a GUI desktop system for modern Linux works. We elaborated on the basic concepts behind the GUI Wayland Protocol, and the reference, or prototypical, Weston Compositor. We provided several simple examples of using the GTK+ and Qt toolkit programming frameworks, to create widget graphics that comprise the Window Generation/Construction phase of the above workflow model.

Finally, we detailed the use of a Data Generation application, gnuplot, and how gnuplot can be used to map its data to windows created by the Qt Creator IDE.