Easy I/O with IO Channels

From Issue #141
January 2006
Nov 29, 2005 By Robert Love

Glib's IO Channels provide several powerful features:

- Buffered I/O: as with C's Standard Library, IO Channels provide user-side buffering, which minimizes the number of system call invocations and ensures that I/O is performed in optimally sized units.
- Portability: IO Channels are portable, working in various UNIX systems as well as Windows.
- Simple yet efficient I/O routines: helper routines to make common programming chores, such as “read exactly one line” or “read the whole file” easy.

Main loop integration: integration into the Glib main loop means multiplexed I/O and event-driven programming is easy.

Although designed for powerful and complex GNOME applications, Glib is actually a separate library from GNOME and readily usable from any C application.

Main Loops

A main loop, sometimes called an event loop, allows a single-threaded process to wait for and handle events originating from multiple sources. Most GUI programmers are familiar with main loops: they allow event-driven GUI programming to register callback functions that are invoked in response to events, such as a button press or window close. The Gtk+ main loop is built on top of Glib's main loop.

The Glib main loop is implemented using multiplexing I/O—in Linux, via the poll() system call. Events are associated with file descriptors, which are watched via poll(). In this manner, the application need not check incessantly for new events but can sleep, consuming no processor time, so long as there is no activity.

Glib's main loop associates a callback function with each event. When the event occurs, the main loop will call back into the given function. Callbacks are invoked in the order that their associated event occurs, although priority may be given to events to change this order. Because multiple events may be watched and multiple callback functions registered, even a single-threaded process can juggle numerous events.

Glib

The Glib library is GNOME's base library, providing portability wrappers and a set of helper functions to make programming in C less arduous. Although part of GNOME, Glib is very much usable on its own, and many non-GNOME projects do indeed utilize Glib without touching any other parts of GNOME. In fact, Glib is even beneficial to console applications. This article makes no assumptions about the use of other GNOME components; the interfaces covered work equally well in a complex GNOME application and a simple console program.

Compiling an application with the requisite Glib support is made easy by the pkg-config program. You can build a binary gio from the source file gio.c with the following command:

```
gcc -Wall -O2 \
        `pkg-config --cflags --libs glib-2.0` \
        -o gio \
        gio.c
```

IO Channels

An IO Channel is represented by the GIOChannel data structure. Its fields are private, and it is accessed using only the official IO Channel interfaces.

Each IO Channel is associated with a single file or other “file-like” object. On Linux, an IO Channel can be associated with any open file descriptor, including sockets and pipes. Once associated, the IO Channel is used to access the file.

Watches are created against a given IO Channel, along with a set of events on which to wait for and a callback function to invoke in response. The watches then integrate with Glib's main loop. When an event occurs—say, a socket has new data available for reading—the watch is triggered and the callback is automatically invoked.

The watch lies at the heart of the power of IO Channels: applications can create multiple watches and integrate them, along with numerous other events, into the Glib main loop, providing event-driven programming to even simple single-threaded applications.

Getting Started

Listing 1 is a complete and working console application that uses IO Channels to communicate across two pipes. It creates two IO Channels, one for the read side of the pipe and another for the write side of the pipe. It then registers watches for these two IO Channels. One watch invokes a callback, gio_in(), when the pipe is available for reading (that is, when a read from the read side of the pipe will not block). The other watch invokes a callback, gio_out(), when the pipe is available for writing (that is, when a write to the write side of the pipe will not block). The gio_out() callback writes a small message into the pipe. The gio_in() callback reads the available data from the pipe and prints it to standard out.

Listing 1. A Complete and Working Console Application That Uses IO Channels to Communicate across Two Pipes

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#include <glib.h>

static gboolean
gio_in (GIOChannel *gio, GIOCondition condition, gpointer data)
{
        GIOStatus ret;
        GError *err = NULL;
        gchar *msg;
        gsize len;
```

```
	if (condition & G_IO_HUP)
		g_error ("Read end of pipe died!\n");

	ret = g_io_channel_read_line (gio, &msg, &len, NULL, &err);
	if (ret == G_IO_STATUS_ERROR)
		g_error ("Error reading: %s\n", err->message);

	printf ("Read %u bytes: %s\n", len, msg);

	g_free (msg);

	return TRUE;
}

static gboolean
gio_out (GIOChannel *gio, GIOCondition condition, gpointer data)
{
	const gchar *msg = "The price of greatness is responsibility.\n";
	GIOStatus ret;
	GError *err = NULL;
	gsize len;

	if (condition & G_IO_HUP)
		g_error ("Write end of pipe died!\n");

	ret = g_io_channel_write_chars (gio, msg, -1, &len, &err);
	if (ret == G_IO_STATUS_ERROR)
		g_error ("Error writing: %s\n", err->message);
```

```
	printf ("Wrote %u bytes.\n", len);

	return TRUE;
}

void
init_channels (void)
{
	GIOChannel *gio_read, *gio_write;
	int fd[2], ret;

	ret = pipe (fd);
	if (ret == -1)
		g_error ("Creating pipe failed: %s\n", strerror (errno));

	gio_read = g_io_channel_unix_new (fd[0]);
	gio_write = g_io_channel_unix_new (fd[1]);
	if (!gio_read || !gio_write)
		g_error ("Cannot create new GIOChannel!\n");

	if (!g_io_add_watch (gio_read, G_IO_IN | G_IO_HUP, gio_in, NULL))
		g_error ("Cannot add watch on GIOChannel!\n");
	if (!g_io_add_watch (gio_write, G_IO_OUT | G_IO_HUP, gio_out, NULL))
		g_error ("Cannot add watch on GIOChannel!\n");
}

int
main (void)
{
	GMainLoop *loop = g_main_loop_new (NULL, FALSE);
```

```
        init_channels ();
        g_main_loop_run (loop); /* Wheee! */
        return 0;
}
```

To be sure, this is an example rooted solely in explanation. It is silly to operate a pipe like this in a single application. Further, the program will continually read from and write to the pipe (you can kill the process with Ctrl-C). Nonetheless, this example serves a good purpose: it demonstrates event-driven programming and the utility of a main loop multiplexing I/O. The natural extension of this program would be to separate it into two processes, a consumer and a producer, and actually communicate interprocess over the pipe. Add a handful of other IO Channels, some GUI events, a few timers, and so on, to the main loop, and you will have a real program!

Creating an IO Channel

There are two ways to create a new IO Channel. The easiest method creates the IO Channel from an existing open file descriptor. The file descriptor can map to any object, including sockets and pipes:

```
GIOChannel *gio;

gio = g_io_channel_unix_new (fd);
if (!gio)
        g_error ("Error creating new GIOChannel!\n");
```

As its name suggests, this function is UNIX-specific. Another method is available for creating an IO Channel in a platform-independent manner:

```
GIOChannel *gio;
GError *err = NULL;

gio = g_io_channel_new_file ("/etc/passwd", "r", &err);
if (!gio)
        g_error ("Error creating new GIOChannel: %s\n", err->msg);
```

The second parameter specifies the mode with which to open the file: one of r, w, r+, w+, a or a+. These values have the same meaning as with fopen(). For example, in this

code snippet, we are asking to create a read-only IO Channel.

In our example program in Listing 1, we create two IO Channels using g_io_channel_unix_new(), one for each end of the pipe.

Creating a Watch

Given an IO Channel, creating a watch is easy:

```
guint ret;

ret = g_io_add_watch (gio, G_IO_IN, callback, NULL);
if (!ret)
        g_error ("Error creating watch!\n");
```

The first parameter, gio, is the IO Channel we want to watch. The second parameter is a mask of one or more conditions for which to watch. The condition G_IO_IN is true when there is data to be read without blocking. Other conditions are G_IO_OUT (data can be written without blocking), G_IO_PRI (urgent data is available to read), G_IO_ERR (an error occurred) and G_IO_HUP (the connection was hung up). The third parameter is the callback function that the Glib main loop will invoke when the event occurs.

Watch callbacks take the following form:

```
gboolean callback (GIOChannel *gio,
                   GIOCondition condition,
                   gpointer data);
```

where gio is the applicable IO Channel, condition is a bitmask of the triggered events and data is the last argument given to g_io_add_watch().

If the return value of the callback is FALSE, the watch is automatically removed.

In our example program in Listing 1, we create two watches, one for each of our IO Channels.

Reading from an IO Channel

The Glib library provides three basic interfaces for reading from an IO Channel.

The first, g_io_channel_read_chars(), is used to read a specific number of characters from an IO Channel into a pre-allocated buffer:

```
GIOStatus g_io_read_chars (GIOChannel *channel,
                           gchar *buf,
                           gsize count,
                           gsize *bytes_read,
                           GError **error);
```

This function reads up to count bytes from the IO Channel channel into the buffer buf. Upon successful return, bytes_read will point to the number of bytes actually read. On failure, error will point to a GError structure.

The return value is an integer with one of four values: G_IO_STATUS_ERROR (an error occurred), G_IO_STATUS_NORMAL (success), G_IO_STATUS_EOF (end-of-file was reached) or G_IO_STATUS_AGAIN (resource temporarily unavailable, try again).

The second interface, g_io_channel_read_line(), is used to read an entire line from a given IO Channel. It will not return until a newline-delimited line is read:

```
GIOStatus
g_io_channel_read_line (GIOChannel *channel,
                        gchar **str_return,
                        gsize *length,
                        gsize *terminator_pos,
                        GError **error);
```

Upon successful return, str_return will contain a pointer to a newly allocated block of memory of length bytes. terminator_pos is the offset into str_return of the terminating character. The data returned by this function must be freed via a call to g_free().

The final interface, g_io_channel_read_to_end(), reads all remaining data from the file into the given buffer:

```
GIOStatus g_io_channel_read_to_end (GIOChannel *channel,
                                    gchar **str_return,
                                    gsize *length,
                                    GError **error);
```

Upon successful return, str_return will contain a pointer to a newly allocated block of memory of length bytes, which must be freed via g_free().

This function should not be used on IO Channels that map to file descriptors that do not necessarily return end-of-file when exhausted of data. For example, we could not use this function in our example program, because pipes do not return end-of-file until the other side has closed its end of the connection. Thus, our example would block indefinitely if we use g_io_channel_read_to_end().

Instead, in our example, we use g_io_channel_read_line() to read an entire line from the pipe.

Writing to an IO Channel

Glib provides a single interface for writing to an IO Channel:

```
GIOStatus
g_io_channel_write_chars (GIOChannel *channel,
                          const gchar *buf,
                          gssize count,
                          gsize *bytes_written,
                          GError **error);
```

A successful call to g_io_channel_write_chars() will write up to count bytes from the buffer pointed at by buf into the file represented by the IO Channel channel. If count is negative one, buf will be treated as a NULL-delimited string. On return, bytes_written contains the actual number of bytes written.

Like C's Standard I/O Library, IO Channels perform buffered I/O to optimize performance. Thus, a write request may not actually be submitted to the kernel after each call to g_io_channel_write_chars(). Instead, glib may wait until a sufficiently large buffer is full and then submit the write request, in one large swoop.

The g_io_channel_flush() function is used to force a submission of any pending write requests to the kernel:

```
GIOStatus
g_io_channel_flush (GIOChannel *channel,
                    GError **error);
```

Buffering can be turned off altogether, if so desired:

```
g_io_channel_set_encoding (gio, NULL, NULL);
g_io_channel_set_buffered (gio, FALSE);
```

The second function disables buffering. The first sets the IO Channel's encoding to NULL, which is required for disabling buffering.

In our example program, we use g_io_channel_write_chars() to write a small string to the pipe. We do not mess with the buffering of the IO Channel.

Seeking Around an IO Channel

As a file is written to or read from, Glib automatically updates the application's position within the file. When a file is first opened, the position is set to the start. Read four bytes, and the position is subsequently set to the fifth byte. This is the file I/O behavior nearly all developers are familiar with.

Sometimes, however, an application wants to seek around the file on its own. For that, Glib provides a single function:

```
GIOStatus
g_io_channel_seek_position (GIOChannel *channel,
                            gint64 offset,
                            GSeekType type,
                            GError **error);
```

A successful call will change the current position in the file as described by offset and type.

The type parameter is one of G_SEEK_CUR, G_SEEK_SET or G_SEEK_END. G_SEEK_CUR asks that the file's position be updated to offset bytes from the current position. G_SEEK_SET asks that the file's position be set to offset. Thus, the following code sets the file position to the start of the file:

```
GIOStatus ret;
```

```
GError err;

ret = g_io_channel_seek_position (gio,
                                  0,
                                  G_SEEK_SET,
                                  &err);
if (ret)
        g_error ("Error seeking: %s\n",
                 err->message);
```

Finally, G_SEEK_END asks that the file's position be set to offset bytes from the end of the file.

We do not use g_io_channel_seek_position() in our sample application, because even if we had a reason to, pipes are not seekable.

Closing an IO Channel

When done with an IO Channel, it is destroyed and the file is closed via a call to g_io_channel_shutdown:

```
GIOStatus
g_io_channel_shutdown (GIOChannel *channel,
                       gboolean flush,
                       GError **err);
```

If flush is TRUE, any pending I/O is first flushed.

Deprecated Functions

Several older IO Channel functions are provided by Glib: g_io_channel_read(), g_io_channel_write() and g_io_channel_seek(). The functions discussed in this article have replaced these older, deprecated functions and should be used instead. Particularly, never mix these old functions with the other functions on the same IO Channel.

Putting It All Together

We have now covered all of the IO Channel interfaces used in Listing 1. Only two details go unexplained. First, in our example program, we create a pipe in the usual manner:

```
int fd[2], ret;

ret = pipe (fd);
if (ret)
        g_error ("Creating pipe failed: %s\n",
                 strerror (errno));
```

Second, our main() function initializes the Glib main loop, calls our function to initialize the IO Channels and then runs the main loop:

```
int
main (void)
{
        GMainLoop *loop;

        loop = g_main_loop_new (NULL, FALSE);
        init_channels ();
        g_main_loop_run (loop);       /* Wheee! */
        return 0;
}
```

The g_main_loop_new() function creates a new main loop and returns a pointer to a new GMainLoop structure. Once ready to start the main loop running, we call g_main_loop_run() and let Glib handle the rest.

Conclusion

Applications that use IO Channels combine a portable multiplexed I/O solution with smart buffering and Glib main loop integration. The result is a solution that allows applications to juggle I/O among hundreds of file descriptors. A graphical network client can manage all of its open sockets, handle new connections seamlessly, juggle a dozen open files and respond to numerous GUI events from a single place and with a single thread.

Glib's IO Channels make I/O easy, efficient, and—gasp—even fun!

Resources for this article: /article/8632.

Robert Love is a senior kernel hacker in Novell's Ximian Desktop group and the author of Linux Kernel Development (SAMS 2005), now in its second edition. He holds degrees in CS and Mathematics from the University of Florida. Robert lives in Cambridge, Massachusetts.