# Real-Time display in RT Linux using IO channels

**M. Shivakumar and M. Hemavathy**
Central Research Laboratory, Bharat Electronics Limited
Jalahalli-post, Bangalore-560013, INDIA
{nithavis,hemavathy_m}@yahoo.com

**Abstract**

Real Time applications involve time-critical tasks, which are to be run in a high precision scheduling environment always meeting the timing constraints. Applications which run on RT Linux uses a hard real-time kernel for tasks which needs real-time processing and the Linux OS for other non real-time tasks. These include tasks like networking and display. It uses the display system present in desktop Linux operating system for part of the application involving display. At high throughput rate this leads to synchronization problems between real-time acquisition and display modules as in the case of real-time tracking applications. The Linux scheduler will not handle the data sent at the rate of a real-time task causing buffer overflow and system crash. In this paper a method to synchronize, the display task to the output data rate of the real-time task, using IO watches for the buffered Input/Output channel is discussed.

## 1 Introduction

RTLinux is a version of the Linux Operating System. It's a hard real time operating system that runs Linux as its lowest priority execution thread. The Linux thread is made completely preemptable so that real-time threads and interrupt handlers are never delayed by non real-time operations. RTLinux supports real-time interrupt handlers and real-time periodic tasks with interrupt latencies and scheduling jitter close to hardware limits.

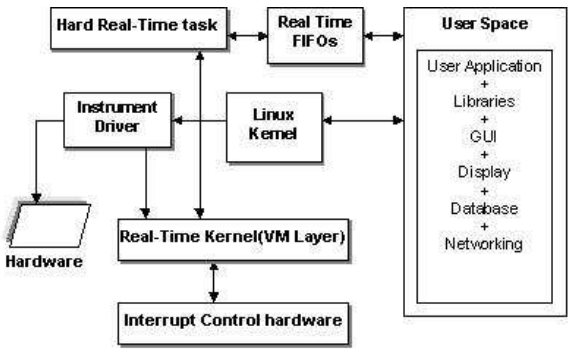


**FIGURE 1:** *Real-Time application architecture*

Real-time tasks in RTLinux can communicate with Linux processes via FIFOs and shared memory. Thus, real-time applications can make use of all the powerful, non real-time services of Linux, including:

- Networking
- Display and Graphics
- Windowing systems
- Data analysis packages
- Linux device drivers, and
- Standard POSIX API

## 2 Display Process

RT Linux does not have a real-time display mechanism within itself. It uses the standard Linux graphics facilities to do that. The real-time tasks inserts data into the FIFO and the Linux process access this data for further processing like creating display graphics based on this data. The FIFOs and RT-FIFO buffers are allocated in the kernel address space. The FIFO is filled by real-time data generated by RT Linux.

Linux user processes, on the other hand, see RT-FIFOs as ordinary character devices and read the data whenever it gets the scheduler. The display process can use any of the standard Linux Graphical Widgets like KDE, GNOME and using the standard APIs like GTK, GDK, and GNOME etc. We

can program a thread, which looks out for the RT-FIFO at a specified interval for any available data, and format that data to display it on the screen. For example, in GTK we can use the command gtk_timeout_add.

*gtk_timeout_add(ms, function, widget);*

This creates a thread, which runs in every interval specified by the timeout function. The function mentioned in the second parameter is called each time and the third parameter has the widget name.

The problem here is, this thread can be scheduled to access the FIFO in every interval specified in milliseconds. This works well for the specified interval, which is more than the Linux scheduling interval of 10ms. If the specified interval is less than the Linux scheduling interval then it will not function accordingly and it is treated as equal to the Linux scheduling jitter only. So the applications, which needs display tasks at a rate, equal to the real-time task is not possible with this set-up for lower timeout rates.

This style of accessing the FIFO to retrieve the data sent by the real-time is not synchronous and induces unwanted delays in displaying the information on the screen. Here the real-time data acquisition and the display, works independently with its own timing parameters. Even though the real-time task is fast enough to generate data, the output is displayed at a slower rate. This is due to the limitation of the display task running on the Linux scheduler. So the whole application is treated as an inefficient one to handle the real-time data at faster rates. This problem is discussed in detail with an example of circular tracking application such as RADAR or SONAR.

# 3 Tracking application

Consider a circular tracking application, which sends data at each degree interval to the real-time application in polar form. The real-time task processes the data in polar form [R,T] and converts into the Cartesian form [X,Y]. This data is passed onto the display task through the RT-FIFO. The targets identified are to be displayed along with the sweep line indicating the position of the tracking beam.

In this real-time application with the present display system using *gtk_timeout_add()* call we can schedule to access the FIFO with the interval of 10ms only. If we update the display at every degree displaying the sweep and targets at that position, it will take 10 ms x 360 = 3600 ms or 3.6 sec for one complete sweep.

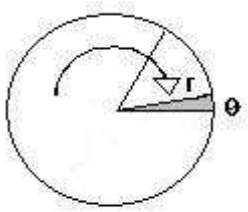


**FIGURE 2:** *A tracking application*

So we can achieve only 16-RPM at the maximum. For the higher RPM this will induce a delay in updating the display and we will not achieve per degree update. For example if we are aiming for 60 RPM the timeout signal should work with an interval of 2.778 ms per degree, which is not possible if we use the timeout system. So in order to decrease the display interval and synchronize with the real-time task the IO channel watch method is used.

# 4 GIO Channels

The GIOChannels data type provides a method for using file descriptors, pipes, and sockets, and integrating them into the main event loop. To create a new GIOChannel on UNIX systems *g_io_channel_unix_new()* is used. This works for plain file descriptors, pipes and sockets. Alternatively, a channel can be created for a file in a system independent manner using *g_io_channel_new_file()*. To add a GIOChannel to the main event loop *g_io_add_watch()* or *g_io_add_watch_full()* is used. You have to specify which events you are interested in the GIOChannel, and should provide a function to be called whenever these events occur. Inside the function *g_io_channel_read_chars()* is used to read the data from the RT-FIFO. This data is further processed and displayed.

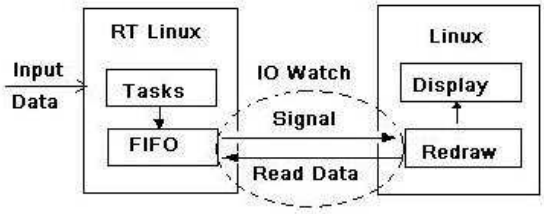


**FIGURE 3:** *The working of IO watches*

The data thus obtained is in synchronisation with the real-time data acquired through the real-time task.

# 5 Adding IO Watches

The real-time task in RTLinux, takes the input data from the external source and data is processed. Then the processed data is stored in the RT-FIFOs assigned with that task. On the Linux side, the RT-FIFO, which we want to add IO watch, is opened in the read mode with the specified options. This is typically done with Linux open() system call as shown below.

```
fd=open("/dev/rtf1", O_RDONLY, O_NONBLOCK);
```

This call opens the /dev/rtf1 with the read only option and in non-blocking mode. The file descriptor is return in fd.

A IO Channel is created by declaring a variable of the type GIOChannel and passing the file descriptor of the FIFO.

```
GIOChannel *MyChannel;
MyChannel=g_io_channel_unix_new(fd);
```

Assign the watch parameters to the GIOChannel and the function name to call whenever a new data is found. This will integrate the GIOChannel to the main event loop of the GTK. This has to be called with the following parameters.

*channel*: a GIOChannel
*condition*: the condition to watch for. Here you have to specify the condition so has to look for the arrival of new data in the FIFO.
*function*: the function to call when the condition is satisfied.
*user data* : user data if any to pass to the function.
*returns*: the event source id.

```
g_io_add_watch(MyChannel,G_IO_IN | G_IO_NVAL,
my_function, NULL);
```

When the add_watch satisfies the condition and return success, my_function is called. GIO channel read function is invoked inside my_function, which reads the channel to access the data. This has to be called with the following parameters.

*channel*: a GIOChannel
*buffer*: a buffer to read data into
*count* : the size of the buffer
*bytes read* : The number of bytes read
*error* : A location to return an error of type
*returns*: the status of the operation.

```
g_io_channel_read_chars(myChannel, &buffer,
sizeof(buffer), error, bytes_read);
```

The read data is available in the buffer variable for further display processing. The processed data is finally displayed on the screen. The timing diagram in Figure 4, shows the interaction between the real-time task and the linux display task.

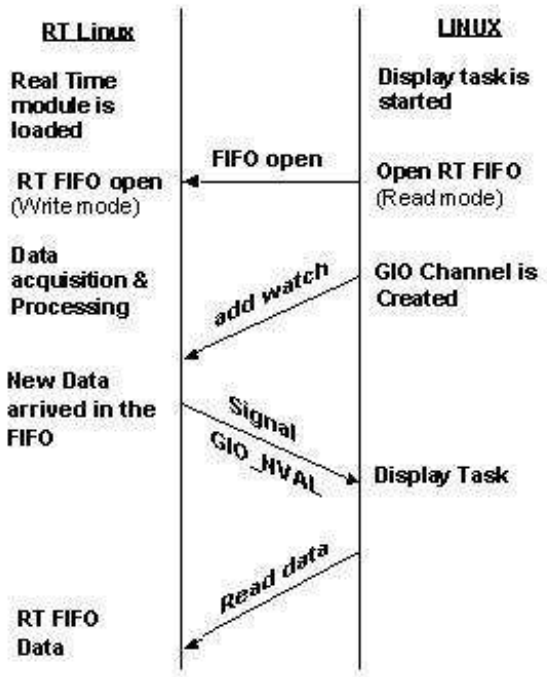


**FIGURE 4:** *Timing diagram of IO watch operation*

# 6 Tracking application using IO watch

Now consider the tracking application again with the implementation of IO watches. We can schedule the real-time process to generate the data necessary for the per degree update. For 60 RPM we can set the timer in real-time task with an interval of 2.778 ms delay. So every 2.778 ms the data needed for the one-degree updation is pushed into the RT-FIFO. At the Linux end, the main-event loop of the display process watches this new arrival of data on the FIFO and a signal is generated, which calls GIOChannel read function. Then the data in the FIFO is read and passed it on to the display function, where final processing is done and the data is displayed on the screen. In this approach one complete revolution takes 2.778 ms x 360 = 1000 ms or 1 sec, thus producing a 60-RPM. Still the speed can be increased to match the refreshing cycle of the monitor screen by reducing the timer interval in the real-time task.

# 7 Conclusion

The IO channel watch method in real-time displays helps us to achieve the display rate well equal to the

real-time data generation rate. So we can improve the performance of the display graphics as equal to the refreshing capacity of the monitor screen and the display hardware. In addition to that the data displayed on the screen is synchronized to the real-time data obtained without any delay.

# References


[1] Cort Dougan & Matt Sherer, *RTLinux POSIX API for IO on Real-time FIFOs and Shared Memory*, Finite State Machine Labs.

[2] Matt Sherer , *Writing applications in RT Linux*, Finite State Machine Labs.

[3] FSM Labs, Inc. December 2002 *Getting Started with RTLinux*.

[4] FSM Labs, Inc. 2001-2002 *Real-Time programming in RTLinux*.

[5] Alex Ivchenko , June 2001, *Application code and RT Linux*, Embedded systems programming.

[6] Kevin Dankwardt , 2000, *Fundamentals of Real Time Linux software design*, www.linuxdevices.com

[7] *The Embedded Linux GUI/Windowing Quick Reference Guide*, www.linuxdevices.com

[8] *The Real Time Linux API*,www.rtlinux.org-documentation-man_pages

[9] *The GLib reference manual*, http://developer.gnome.org/API/2.0/glib/glib-IO-Channels.html#GIOChannel

[10] *Introduction to Linux for Real-Time Control*, National Institute of Standards & Technology, Gaithersburg, MD.

[11] *GTK+ and GNOME Reference manual*, www.developer.gnome.org

[12] *Beginning Linux Programming*,www.wrox.com Wrox publications.

[13] Eric Harlow, *Developing Linux applications*, www.informit.com.