Astronomical Data Analysis Software and Systems X
ASP Conference Series, Vol. 238, 2001
F. R. Harnden Jr., F. A. Primini, and H. E. Payne, eds.

Interfacing Real-time Linux and LabVIEW

P. N. Daly

National Optical Astronomy Observatories, 950 N. Cherry Avenue, P. O. Box 26732, Tucson, AZ 85726–6732, USA

Abstract. Real-time Linux is a set of extensions to the kernel that provides hard real-time functionality with low, bounded latencies and deterministic response. The main methods for communicating between kernel space and user space are fifos and shared memory. LabVIEW is the well-known commercial product for developing control systems and engineering applications. This paper, presents the fifos and shared memory virtual interface (VIs) that allow LabVIEW to communicate and share (bulk) data with the real-time core.

1. Introduction

The cardinal programming rule of real-time Linux is to do as much as possible in user space with only critical sections of code in the real-time core. This demarcation requires methods of exchanging simple, structured and/or bulk data between the real-time side and the application side. Two such methods exist: fifos, for simple or moderately structured data, and shared memory, for bulk data. For the work described in this paper, the interested reader will require the common mbuff 0.7.1[1] and fifos 0.5[2] packages built under either the RTLinux 2.2[3] or RTAI 1.3[4] kernel patches to Linux 2.2.14[5].

2. Code Samples for Command Line Fifos and Shared Memory

To understand what is intended, a quick tour of using fifos and shared memory from command line applications follows.

[1] `ftp://crds.chemie.unibas.ch/PCI-MIO-E/mbuff-0.7.1.tar.gz`

[2] `http://www.realtimelinux.org/CRAN/software/rtai_rtl_fifos-05.tar.gz`

[3] `ftp://rtlinux.com/rtlinux/v2/rtlinux-2.2.tar.gz`

[4] `ftp://www.aero.polimi.it/RTAI/rtai-1.3.tgz`

[5] `ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.14.tar.gz`

2.1. Fifos

Fifos are simple, character-based devices which must be created in the real-time core before being used by an application. To enable /dev/rtf0, for example, with a buffer 1024-bytes long, the code in the init_module entry point would be:

```
int err = 0;
if ( (err=rtf_create(0,1024)) < 0 ) return -EINVAL;
```

A periodic real-time task might then write data in the handler to the fifo using a code segment such as:

```
unsigned int sent = 0;
while ( 1 ) {
 #ifdef RTL  // RTLinux 2.2 re-scheduling point
  (void) pthread_wait_np();
 #else       // RTAI 1.3 re-scheduling point
  (void) rt_task_wait_period();
 #endif
 (void) rtf_put(0,&sent,sizeof(sent));
 sent++;
}
```

The application then reads the fifo:

```
unsigned int recv = 0;
if ( (fd=open("/dev/rtf0",O_RDONLY)) < 0 ) return (fd);
while ( read(fd,&recv,sizeof(recv)) ) {
 (void) printf("%s: received %d\n",__FILE__,recv);
}
(void) close(fd);
```

It is easy to reverse this process to put data into the core and/or make the operation pass structured data. This fifo is destroyed in the cleanup_module using:

```
(void) rtf_destroy(0);
```

2.2. Shared Memory

Shared memory *must* be declared in the init_module and cast to the desired data type or structure. The user space application uses symmetric calls. For example, to obtain a 1 Mb array of signed integers, one could use:

```
static int *mptr = (int *) NULL;
mptr = (int *) mbuff_alloc("myints",1024*1024);
#ifdef __KERNEL__    // code was called by init_module
 if ( mptr == (int *) NULL ) return -ENOMEM;
#else                // code was called by user application
 if ( mptr == (int *) NULL ) return (EXIT_FAILURE);
#endif
```

The memory can then be accessed using straightforward de-referencing and pointer arithmetic to traverse the array. The memory is released using:

```
mbuff_free("myints",(void *)mptr);
```

Note that the name of the memory section must be common between the real-time core and application. Several applications may also references the same memory segment. The only restriction on the size of a memory section is the amount of system memory available. In the WIYN Top-Tilt Module project (Daly 2000), for example, 128 Mb are mapped using a single call to *mbuff_alloc*.

3. The *lvrtl* Package

The functionality described in the previous section is implemented in LabVIEW using the *lvrtl* package developed by the author. Two versions exist, one for LabVIEW 5.1[6] and one for LabVIEW 6*i*[7]. Only the LabVIEW 6*i* code will be developed further as needs arise. Documentation on the design and implementation of *lvrtl* is available in *lvrtl.pdf*[8] as included in either tarball.

This package provides a shared library (/usr/lib/liblvrtl.so) and 78 VIs that furnish access to fundamental data types. It also includes extensive test code for either hard real-time Linux variant. The following VIs are implemented in this release where *dtype* is one of the set {int8, int16, int32, uint8, uint16, uint32, float32, float64, string}:

1. rtf_open, rtf_close, mbuff_open and mbuff_close.
2. rtf_get_*dtype* and mbuff_get_*dtype*.
3. rtf_put_*dtype* and mbuff_put_*dtype*.
4. rtf_read_*dtype* and mbuff_read_*dtype*.
5. rtf_write_*dtype* and mbuff_write_*dtype*.

Note that *rtf_put_string* and *mbuff_put_string* are the only two routines that add a NULL byte to the data before transfer. The real-time core must be set up to accept this extra byte. If the number of elements to put is 0, the code will size the data array and transfer the whole structure (rather than return an error).

The 'get' and 'put' VIs both accept a number of elements of the given data type to read/write. On error, −1 is returned. On success, the number of *bytes* read or written is returned and *not* the requested number of elements. Since the *mbuff_get_*dtype and *mbuff_put_*dtype VIs also require an offset from the start of the memory array, poor G-code programming could result in a request that exceeds the bounds of the memory segment. Such an error is not trapped by this version of the code and the result will likely be an illegal pointer operation in the kernel resulting in a system crash. The programmer is urged to make sure that all inputs to these VIs are valid.

The fifo may be accessed using a blocking read, non-blocking read, read-write or write-only operation.

As an example, Figures 1 and 2 show the front panel and G-code diagram for a *rtf_read_float32* operation. This example reads five floating point numbers

---

[6] `ftp://orion.tuc.noao.edu/pub/pnd/lvrtl.1.1.51.tgz`

[7] `ftp://orion.tuc.noao.edu/pub/pnd/lvrtl.1.1.60.tgz`

[8] `http://www.realtimelinux.org/documentation/lvrtl.pdf`

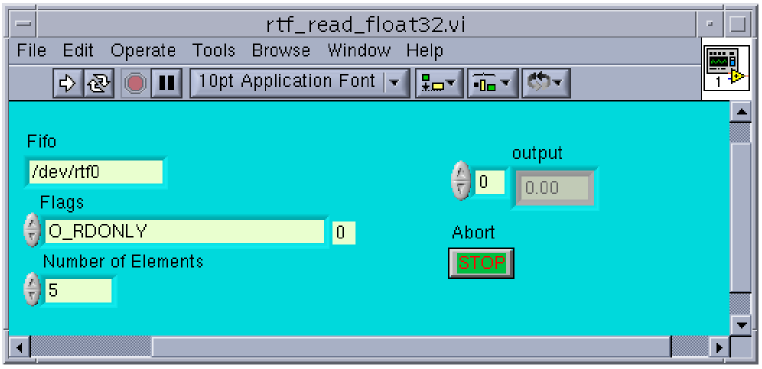

Figure 1. rtf_read_float32

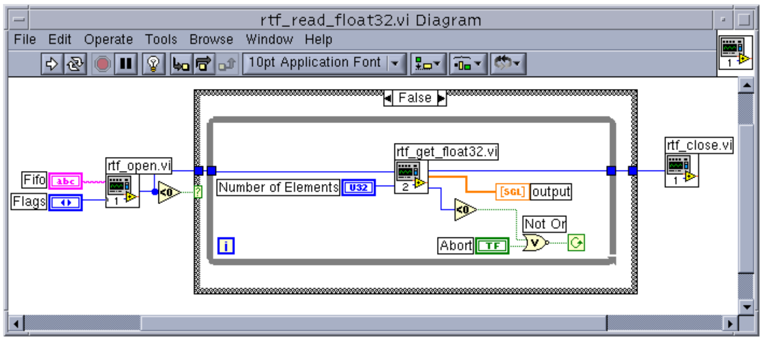

Figure 2. rtf_read_float32

from /dev/rtf0. Note how any output error from the embedded *rtf_get_float32.vi* is trapped within the G-code environment and terminates the loop. The data from each read appears in the 'output' array.

4. Structured Data

Clearly, *lvrtl* can handle basic data types but what of structured data encapsulating heterogeneous data types? There are two approaches. First, one could write one's own suite of VIs and add code to the shared library. There should be enough detail in the documentation and the examples to permit coders to do this. Alternatively, one could wire together VIs handling the data types of the individual elements of the structure. In this latter case, more than one read is required to completely obtain the structured data.

References

Daly, P. N. 2000, in ASP Conf. Ser., Vol. 216, Astronomical Data Analysis Software and Systems IX, ed. N. Manset, C. Veillet, & D. Crabtree (San Francisco: ASP), 388