

## 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

[pnd@noao.edu](mailto:pnd@noao.edu)

**Abstract.** This document describes the real-time fifos and shared memory interface to LabVIEW 6i and LabVIEW 5.1.

### 1. License

This document and software suite are free. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation either version 2 of the License, or (at your option) any later version. These items are distributed in the hope that they will be useful, but *without any warranty*. Without even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this document. If not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge MA 02139, USA.

### 2. Copyright(s)

©2000, P. N. Daly, this document and code. All rights reserved.

### 3. Typographic Conventions

The conventions used in this document are described in Table 1 on page 1. For reasons of clarity,

Table 1.: Typographical Conventions for this Document

Markup	Usage	Effect
<code>\rtlin{blue type-face}</code>	user input	blue type-face
<code>\rtlout{magenta sans-serif}</code>	machine output	magenta sans-serif
<code>\rtlnormal{black times-roman}</code>	normal text (reset)	black times-roman
<code>\rtlmargin{teal italic}</code>	margin notes	teal italic

the `\rtlmargin` is not shown as a margin note within the table (Daly et al. 2000).

## 4. Introduction

This document describes a suite of VIs and the associated shared library source code for interfacing real-time Linux to LabVIEW 6i/5.1. Code is available under both RTLinux 2.2 and RTAI 1.3. Since the code is freely available you may modify it as you wish but please report bugs to the principal author.

Since the way LabVIEW loads ‘code resource files’ differs under 6i and 5.1, there are two tarballs available via anonymous *ftp* from the web-site `ftp://orion.tuc.noao.edu/pub/pnd` called, not surprisingly, `lvrtl.1.1.51.tgz` and `lvrtl.1.1.60.tgz`. Note that *only* the LabVIEW 6i code will be developed further.

With this software you can read or write to a fifo or shared memory segment using fundamental data types and specified array sizes. The limit on the size of data passed is either set by the fifo size or the amount of available system memory.

### 4.1. Installing the Software

Once you have received a copy of the tarball unpack it into some suitable directory. Please edit the *makefiles* to suit your site and build the shared library and example code in the usual way:

```
% make clean all install
```

The shared library is called `lvrtl.so.1.1.51` or `lvrtl.so.1.1.60`, both in the `/usr/lib` directory, and there are links such as `liblvrtl.so` pointing to the appropriate library. In this way you can easily change systems.

Note that the real-time modules will be made for the system you have installed (RTLinux or RTAI). We assume that you know how to load the real-time Linux modules prior to executing any code described here and that *mbuff* has been made for your particular real-time Linux kernel.

## 5. The Command Line Interface

Both real-time fifos and shared memory can be tested from the command line to verify your real-time Linux installation. The modules (*test\_rfifo* and *test\_rmem*) and applications (*test\_ufifo* and *test\_umem*) can handle 9 distinct data types in either read or write mode. Note that to maintain compatibility with LabVIEW, we have used the LabVIEW data type codes as shown in Table 2 on page 3. Data types not handled are *floatExt* since Linux has no ‘16-byte double double’ representation and the *cpmlx* data types since these are just a pair of floats or doubles anyway. Both sets of modules have a frequency of 1 Hz (which you can change if you wish).

### 5.1. Real-time Fifos

The module *test\_rfifo* accepts five command line parameters and the user application *test\_ufifo* four parameters as shown in Table 3 on page 3. Note that for the user application, one can specify the mode as *-mread*, *-mwrite* or *-mnoblock*. The latter is a non-blocking read.

Table 2.: LabVIEW Data Type Codes

Data Type	<datatype>	Code
signed 8-bit integer	int8	1 (0x01)
signed 16-bit integer	int16	2 (0x02)
signed 32-bit integer	int32	3 (0x03)
unsigned 8-bit integer	uint8	5 (0x05)
unsigned 16-bit integer	uint16	6 (0x06)
unsigned 32-bit integer	uint32	7 (0x07)
single precision float	float32	9 (0x09)
double precision float	float64	10 (0x0A)
generic string	string	48 (0x30)

Table 3.: Command Line Parameters for real-time Fifo Test Module

Real-time Parameter	User Parameter	Interpretation
fifo	-f	Fifo number
size		Fifo size in bytes
dtype	-d	Data type in LabVIEW code
nelm	-n	Number of elements on fifo
mode	-m	Fifo access mode

For example, to put a single signed 8-bit integer onto fifo 0 from the real-time kernel and read that value from the user application, use:

```
% rmmmod test_rfifo
% insmod test_rfifo fifo=0 size=1024 dtype=1 nelm=1 mode="write"
% ./test_ufifo -f0 -d1 -n1 -mread
test_ufifo.c: fifo=0, dtype=1, nelm=1, mode=read
test_ufifo.c: opening /dev/rtf0 read-only
test_ufifo.c: opened fifo /dev/rtf0 OK
test_user: received int8 (0x01) msg=1 value=1
test_user: received int8 (0x01) msg=2 value=1
test_user: received int8 (0x01) msg=3 value=1
```

OK?

And one can verify the data using the *dmesg* utility:

```
% dmesg
test_rfifo.c: fifo=0, size=1024, nelm=1, dtype=1, mode=write
test_rfifo.c: known data type 0x01, mode=w
test_rfifo.c: created fifo, status=0
test_rfifo.c: created thread, status=0
test_rfifo.c: made thread periodic, status=0
test_rtl: sent int8 (0x01) msg=1 value=1
test_rtl: sent int8 (0x01) msg=2 value=1
test_rtl: sent int8 (0x01) msg=3 value=1
```

OK?

A more complicated example would reverse the operation and write, say, 5 floating point number to the real-time core:

```
% rmmmod test_rfifo
% insmod test_rfifo fifo=0 size=1024 dtype=9 nelm=5 mode="read"
% ./test_ufifo -f0 -d9 -n5 -mwrite
```

OK?

The data generated in this case is created in the *test\_data\_init* function and is related to data type. It is left as an exercise for the interested reader to confirm that the data is passed correctly.

## 5.2. Shared Memory

The module *test\_rmem* and application *test\_umem* accept four command line parameters as shown in in Table 4 on page 5.

For example, to put a single signed 8-bit integer into shared memory from the real-time kernel and read that value from the user application, use:

```
% rmmmod test_rmem
% insmod test_rmem sname="myint8" dtype=1 nelm=1 mode="write"
% ./test_umem -smyint8 -d1 -n1 -mread
test_umem.c: sname=myint8, dtype=1, nelm=1, mode=read
test_umem.c: created sname, pointer=0x40014000
mbuff_umem: received int8 (0x01) msg=1 value=1
```

Table 4.: Command Line Parameters for Shared Memory Test Module

Real-time Parameter	User Parameter	Interpretation
sname	-s	Name of memory section
dtype	-d	Data type in LabVIEW code
nelm	-n	Number of elements in memory
mode	-m	Memory access mode

```
mbuff_umem: received int8 (0x01) msg=2 value=1
mbuff_umem: received int8 (0x01) msg=3 value=1
```

OK?

And one can verify the data using the *dmesg* utility:

```
% dmesg
test_rmem.c: sname=myint8, nelm=1, dtype=1, mode=write
test_rmem.c: created sname, pointer=d0846000
test_rmem.c: created thread, err=0
test_rmem.c: made thread periodic, err=0
mbuff_rtl: sent int8 (0x01) msg=1 value=1
mbuff_rtl: sent int8 (0x01) msg=2 value=1
mbuff_rtl: sent int8 (0x01) msg=3 value=1
```

OK?

A more complicated example would reverse the operation and write, say, 5 floating point number to the real-time core:

```
% rmmod test_rmem
% insmod test_rmem sname="myfloat" dtype=9 nelm=5 mode="read"
% ./test_umem -smyfloat -d9 -n5 -mwrite
```

OK?

The data generated in this case is created in the *test\_data\_init* function and is related to data type. It is left as an exercise for the interested reader to confirm that the data is passed correctly.

## 6. The LabVIEW Interface

The tarball provides 78 VIs for accessing real-time fifos and shared memory. These can be broken down as follows:

1. *rtf\_open.vi*, *mbuff\_open.vi*, *rtf\_close.vi*, *mbuff\_close.vi*. These are the analogous open and close calls for *rtf* and *mbuff* packages;

2. *rtf\_get\_<datatype>.vi* and *mbuff\_get\_<datatype>.vi*. These are the *get* operations for the data types specified in column 2 of Table 2 on page 3. For example, to get a signed 8-bit integer off a fifo, the appropriate VI is *rtf\_get\_int8.vi*.
3. *rtf\_put\_<datatype>.vi* and *mbuff\_put\_<datatype>.vi*. These are the *put* operations for the data types specified in column 2 of Table 2 on page 3. For example, to put a single precision float into shared memory, the appropriate VI is *mbuff\_put\_float32.vi*.
4. *rtf\_read\_<datatype>.vi* and *mbuff\_read\_<datatype>.vi*. These VIs bundle the open, read-loop and close VIs into an example for each of the data types.
5. *rtf\_write\_<datatype>.vi* and *mbuff\_write\_<datatype>.vi*. These VIs bundle the open, write-loop and close VIs into an example for each of the data types.
6. *test\_lfifo.vi* and *test\_lmem.vi*. These are the VIs that handle all data types for testing purposes. Make sure the front panel input parameters match those invoked by the real-time module *insmod* command or memory corruption can occur. These VIs are *only* available with the LabVIEW 6i tarball.

Note that the *rtf\_put\_string.vi* and *mbuff\_put\_string.vi* are the only two that add a NULL byte before the data transfer. The real-time core must be set up to accept the NULL byte also (just as the test code is).

### 6.1. Real-time Fifos

The *rtf\_open.vi* requires a fifo name and access mode as input parameters and returns the file descriptor of the opened fifo or a negative number on error. This error should be trapped in G-code.

The *rtf\_close.vi* accepts a file descriptor input, closes the file and returns the status.

The *rtf\_get\_<datatype>.vi* accepts the file descriptor input and a number of elements. It reads the fifo for the requested number of elements of the known data type and returns the file descriptor, a status value (-1 on error or number of bytes read on success) and the data in an array of the appropriate data type. Note that the data array is *dynamically re-sized* to hold all the incoming data so that the VI can hold a single value or a complete array of values. To re-iterate, the return value on success is the number of *bytes* read from the fifo and *not* the number of elements read.

Symmetrically, the *rtf\_put\_<datatype>.vi* accepts the file descriptor input, the number of data elements and an array of values of the appropriate data type and write them to the fifo. It returns the file descriptor and a status value (-1 on error, number of bytes written on success) which should be checked in G-code. Note that an input of 0 into the *Number of Elements* control is *ignored* and the whole data set is sent. For the *rtf\_put\_string.vi* a NULL terminating byte is also added to the data transfer.

For example, let us write 5 single precision floating point numbers from the kernel to user space. For this we can use the bundled up *rtf\_read\_float32.vi*. The (LabVIEW 6i) front panel for this example is shown in Figure 1 on page 7 and the G-code diagram is in Figure 2 on page 7. As we can see, the G-code traps errors returned by the *rtf\_open.vi* and *rtf\_get\_float32.vi*.

To execute this example from LabVIEW, first insert the (test) module:

```
% rmmmod test_rfifo
% insmod test_rfifo fifo=0 size=1024 dtype=9 nelm=5 mode="write"
```

OK?

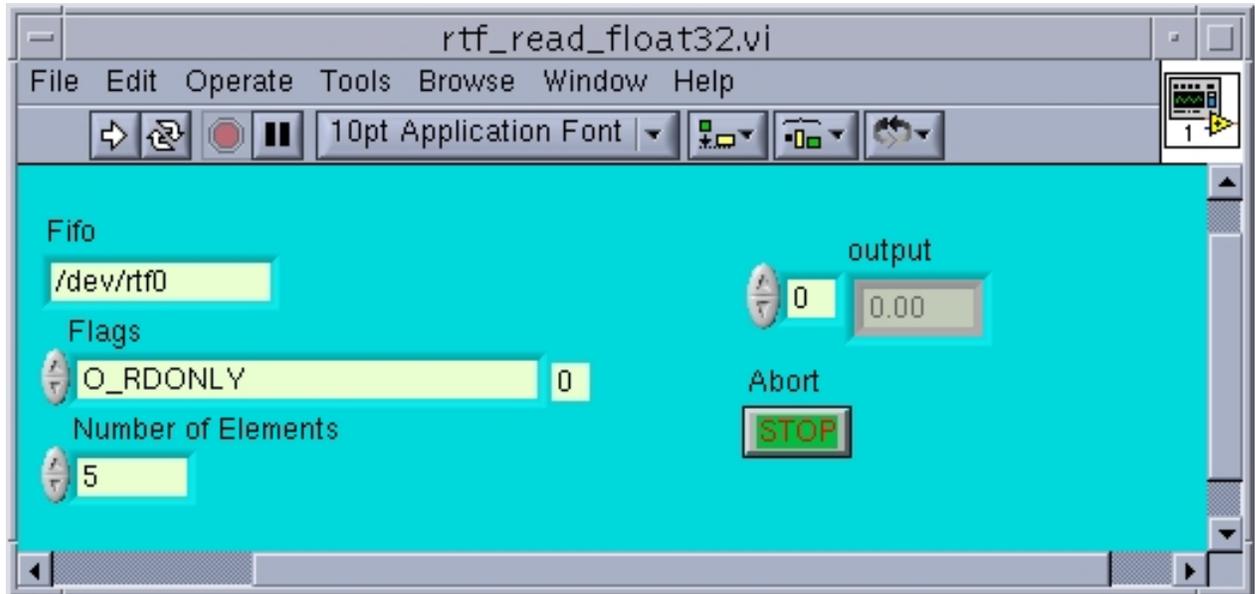


Figure 1.: rtf\_read\_float32.vi Front Panel

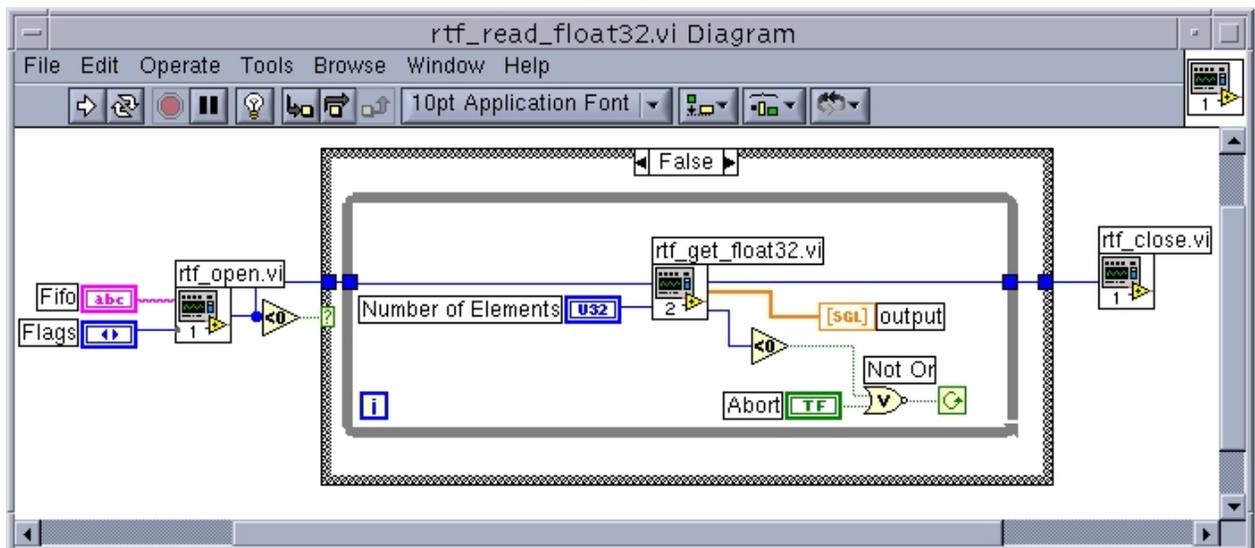


Figure 2.: rtf\_read\_float32.vi G-code Diagram

**DRAFT**

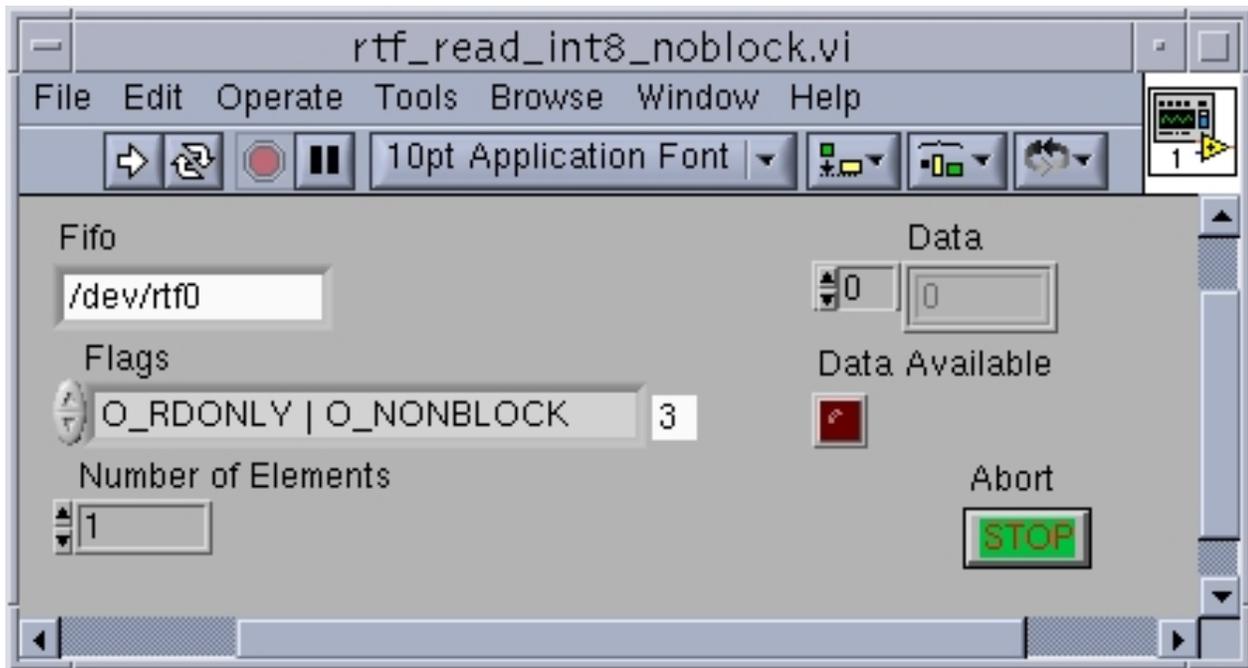


Figure 3.: rtf\_read\_int8\_noblock.vi Front Panel

Then run the VI in the usual way. The values in the output array should become 9.00, 18.00, 27.00, 36.00 and 45.00 respectively. The *put* or *write* VIs do the opposite of the *get* and *read* VIs respectively.

*Non-Blocking Reads* We provide no explicit traps for non-blocking reads but the library accepts that fifos can be opened in such a way. Such reads, typically, return a negative number when no data is available. In Figure 3 on page 8 we show an example of a VI for a non-blocking read of a single 8-bit integer from fifo 0. The associated G-code diagram is in Figure 4 on page 9 Note how we trap the return and indicate data is ready when the return value is positive.

We can run this VI after inserting the *test\_rfifo* module:

```
% rmmmod test_rfifo
% insmod test_rfifo fifo=0 size=1024 dtype=1 nelm=1 mode="write"
```

OK?

If you try this, note that the *Data Available* flag beats with a 1 Hz frequency in synchronization with the real-time module.

## 6.2. Shared Memory

The *mbuff\_open.vi* requires a section name, data type and number of elements (of the given type). The memory is allocated and the pointer to the memory is returned as a value into the integer *Memory Pointer* argument (not the address) since LabVIEW cannot return pointers *per se*. If the allocation fails, a negative number is returned in *Error Out* otherwise it is zero. Note that the internal pointer is declared as a *static* variable creating a possible race condition. This effect

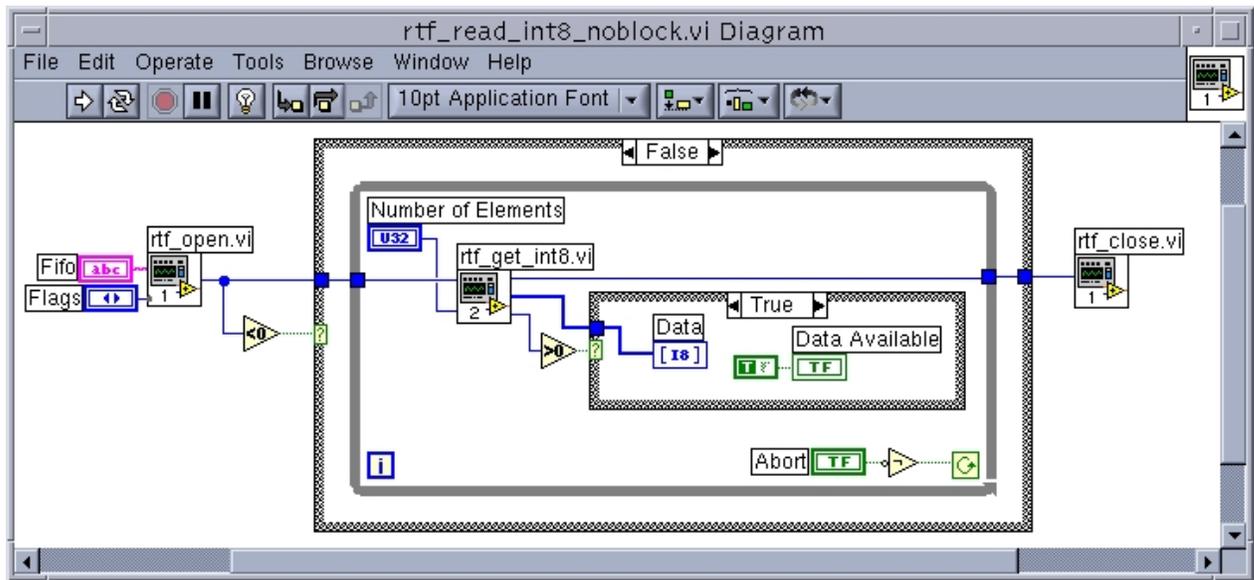


Figure 4.: rtf\_read\_int8\_noblock.vi G-code Diagram

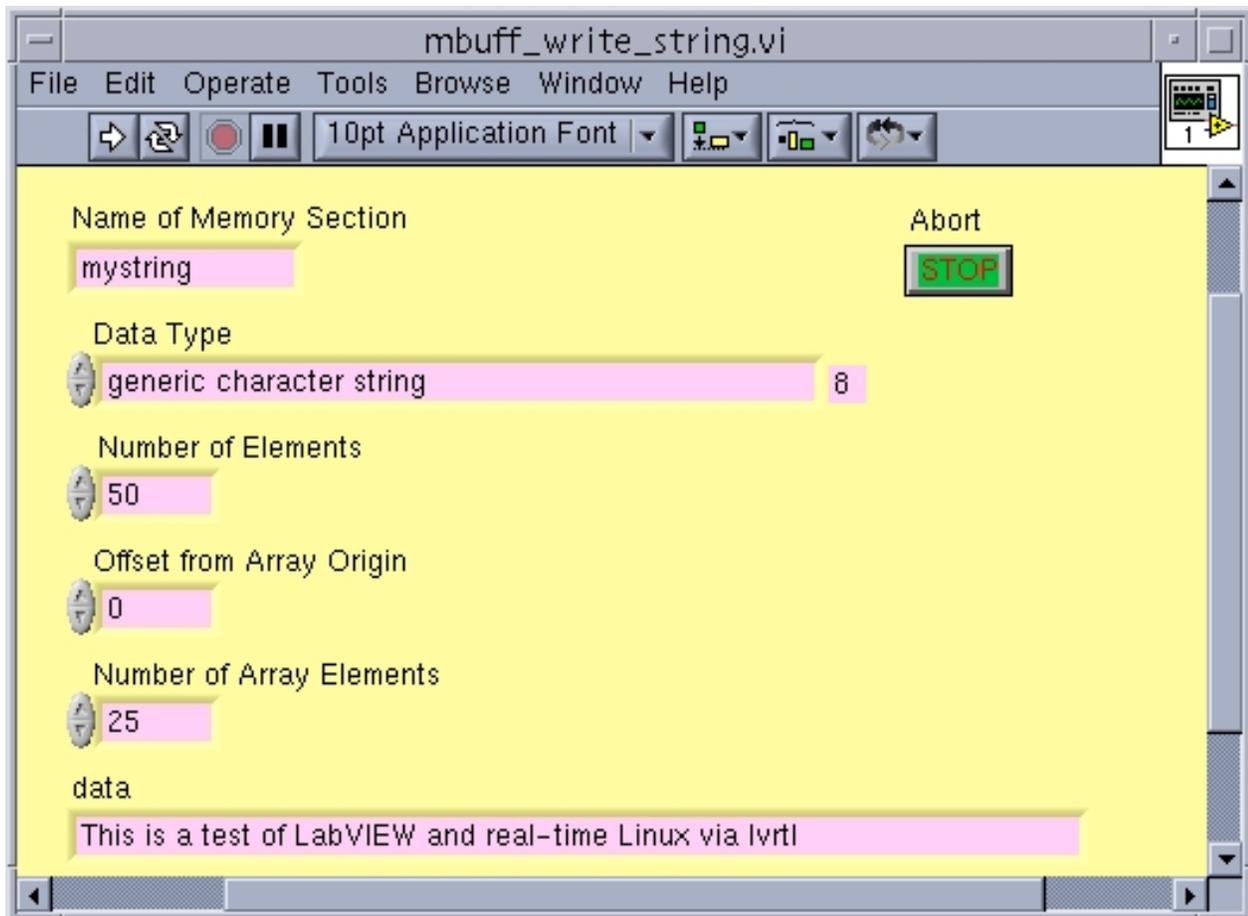
is mitigated by making the VI re-entrant and requiring that calls to *mbuff\_open.vi* are sequential wherever possible.

All other VIs, decode the input integer address and cast it to a pointer of the correct type so that the software knows the start address of the memory area. For the *mbuff\_close.vi*, the memory section name and the address are the only inputs and the memory is released via *mbuff\_free*. Since this function returns a *void*, no status check is possible so *mbuff\_close.vi* always returns 0.

The *mbuff\_get\_<datatype>.vi* accepts the address input and decodes it for the appropriate data type. It also accepts two other inputs: the offset from the start of the memory section and the number of data elements to read from the memory section. It returns a status (-1 on error, number of *bytes* read on success), the input address and the data. The data array is dynamically re-sized to accept all the values read.

Symmetrically, the *mbuff\_put\_<datatype>.vi* accepts the encoded address input, the offset, the number of data elements and an array of values of the appropriate data type and writes them to the memory section. Let us be clear as to what it writes and where: the input data array is read from 0 up to *delm* values (the number of data elements) and those values are written to the shared memory section starting at the offset from the base input address. Note that there is no checking the memory section upper boundary so putting values at a high offset where the number of data elements to put exceeds the end of the memory section could result in memory corruption. The VI returns the file descriptor and a status value (-1 on error, number of *bytes* written on success) which should be checked in G-code. Note that an input of 0 into the *Number of Elements* control is *ignored* and the whole data set is sent. For the *mbuff\_put\_string.vi* a NULL terminating byte is also added to the data transfer.

For example, let us write a string to the real-time core. For this we can use the bundled up *mbuff\_write\_string.vi*. The front panel for this example is shown in Figure 5 on page 10 and the G-code diagram is in Figure 6 on page 11. As we can see, the G-code traps errors returned by the *mbuff\_open.vi* and *mbuff\_put\_string.vi*.

Figure 5.: `mbuffer_write_string.vi` Front Panel

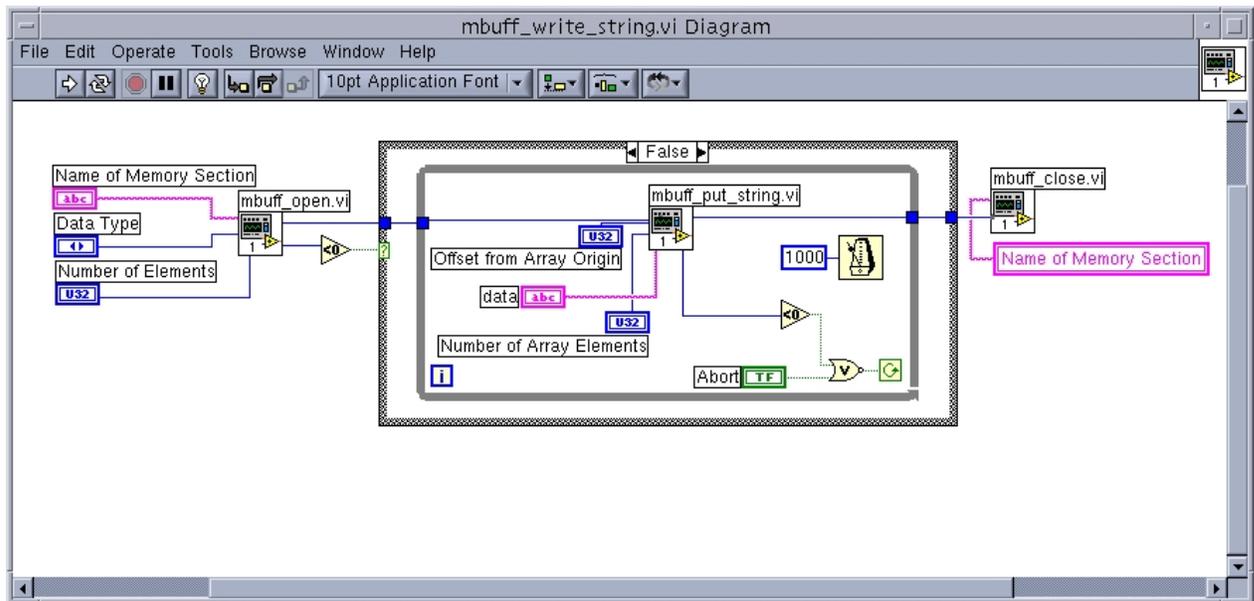


Figure 6.: mbuf\_write\_string.vi G-code Diagram

To execute this example from LabVIEW, first insert the (test) module:

```
% rmmmod test_rmem
% insmod test_rmem sname="mystring" dtype=48 nelm=50 mode="read"
```

Then run the VI in the usual way. Although the input data string is ‘This is a test of LabVIEW and real-time Linux via lvrtd’, only the first 25 characters are sent to the real-time core. Thus the real-time core gets the string ‘This is a test of LabVIEW’ only. If we increment the *Offset from Array Origin*, we move this substring along in the memory buffer. This can be verified with *dmesg*.

The *put* or *write* VIs do the opposite of the *get* and *read* VIs respectively.

## 7. Structured Data

The VIs described above are generic inasmuch as they handle fundamental data types and dynamically re-size arrays to handle multi-valued data. The question remains, though, as to structured data. Clearly, if the structure contains elements of a single data type, this is handled by the appropriate data type VI, specifying the number of elements in the structure as the number of elements to get off the fifo. What, though, about dissimilar data types in a structure? Consider the following structure which is to be put on a fifo:

```
struct mystruc {
    int myint;
    float myfloat;
    char mychar[40];
}
```

There are two approaches. First, one could write one's own VI and add code to the shared library. We believe that there is enough detail in the documentation (Other 2000) and the examples given with this software to do this.

Alternatively, one could wire together the three VIs *rtf\_get\_int32.vi*, *rtf\_get\_float32.vi* and *rtf\_get\_string.vi* with appropriate inputs. Clearly, this requires three reads of the fifo to completely obtain the structured data.

## 8. Document Revision History

23 August 2000, PND: Original version.

29 August 2000, PND: Updated to reflect 5.1 differences and non-blocking read VI.

**Acknowledgments.** Linux is a registered trade mark of Linus Torvalds. LabVIEW is a trade mark of National Instruments Corporation.

## References

Daly, P. N., Mahoney, T. J., and Küpper, J. 2000, 'RTLDOC L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> Template and Style File' in *Real Time Linux Documentation Project*, **1**, P. N. Daly and J. Küpper, eds., Real Time Linux Community Press

Other, A. N. 2000, *Using External Code in LabVIEW*, July 2000 Edition, National Instruments Corporation, Part Number 370109A-01