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, U S A
pnd@noao.edu

Abstract

Hard real-time Linux variants, RTLinux and RTAI, both use fifos and shared memory to communicate with user applications. In this work, we describe a new package, *lvrtl*, which allows LabVIEW VIs to use these mechanisms in a completely generic way. For the *get* operations, the incoming data is collected into a dynamically resized array thus handling scalar and array data for several different data types. There are symmetric *put* operations and a complete suite of VIs with which to build LabVIEW applications. The package also includes extensive test code under RTLinux 2.2 and RTAI 1.3.

1 Introduction

This paper 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. The code is freely available so you may modify it as you wish but please report bugs to the principal author. Note that *only* the LabVIEW $6i$ code will be developed further.

This package supercedes the earlier fifos package based upon code interface nodes [1], rather than shared libraries [2], which has now been withdrawn. With this new software you can read or write to a fifo or shared memory segment using fundamental data types with arbitrary array sizes. The limit on the size of data passed is either set by the fifo size or the amount of available system memory.

2 Installing the Software

This code, lvrtl 1.1.51[1] or lvrtl 1.1.60[2], has been developed and tested using the following infrastructure: RTlinux 2.2[3] and RTAI 1.3[4], Linux 2.2.14[5], mbuff 0.7.1[6] and fifos 0.5[7].

Note that, although we use the common *mbuff* and *fifos* packages, we believe that this code should work without modification using a standard RTLinux 2.2 distribution (which includes earlier releases of these packages).

For native RTAI 1.3, without using the common *mbuff* and *fifos* packages, some modest amount of re-coding and re-compiling would have to be done to use the *rtai_malloc* and *rtai_free* calls. In this case, a simple macro definition should suffice in *lvrtl.c* and the test code files ⋆.RTAI (although this approach has never been tested):

```
#ifdef RTAI
 #define mbuff.h rtai_shm.h
 #ifdef __KERNEL__
  #define mbuff_alloc(n,s) \
    rtai_kmalloc(nam2num(n),s)
  #define mbuff_free(n,a) \
    rtai_kfree(num2nam(n))
 #else
  #define mbuff_alloc(n,s) \
    rtai_malloc(nam2num(n),s)
  #define mbuff_free(n,a) \
    rtai_free(num2nam(n),a)
 #endif
#endif
```

Prior to installing the *lvrtl* package you must have a working hard real-time Linux system. Furthermore,

[1] ftp://orion.tuc.noao.edu/pub/pnd/lvrtl.1.1.51.tgz
[2] ftp://orion.tuc.noao.edu/pub/pnd/lvrtl.1.1.60.tgz
[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
[6] ftp://crds.chemie.unibas.ch/PCI-MIO-E/mbuff-0.7.1.tar.gz
[7] http://www.realtimelinux.org/CRAN/software/rtai_rtl_fifos-05.tar.gz

we assume you know how to load real-time Linux modules.
Once you have received a copy of the tarball unpack it into some suitable directory. Please edit the *makefiles* to suit your site—principally the location of the *mbuff* and *fifos* include files—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 *etc* pointing to the appropriate library. In this way you can easily change systems.

3 Command Line Interface

Both real-time fifos and shared memory can be tested from the command line to verify your real-time Linux installation (independent of LabVIEW). 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 1. Data types not handled are *floatExt* since Linux has no '16-byte double double' representation and the set of {*cmplx64*, *cmplx128*, *cmplxExt*} data types since these are just a pair of floats, doubles or extended precision doubles anyway.

C-type	DTYPE	Value	Code
signed char	int8	1	0x01
signed short	int16	2	0x02
signed int	int32	3	0x03
unsigned char	uint8	5	0x05
unsigned short	uint16	6	0x06
unsigned int	uint32	7	0x07
float	float32	9	0x09
double	float64	10	0x0A
char $\star$	string	48	0x30

TABLE 1: *LabVIEW Data Type Codes*

Both sets of modules have a frequency of 1 Hz (which you can change if you wish).

3.1 Real-time Fifos

The module *test_rfifo* accepts five command line parameters and the user application *test_ufifo* four parameters as shown in columns 1 and 2 of Table 2 respectively. Note that for the user application, one can specify the mode as read-only (*-mread*), write-only (*-mwrite*) or a non-blocking read (*-mnoblock*).

Module	Application	Interpretation
fifo	–f	Fifo number
size		Fifo size
dtype	–d	LabVIEW Data Type
nelm	–n	Number of elements
mode	–m	Access mode

TABLE 2: *command line parameters*

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:

```
% rmmod 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
```

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
```

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

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

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.

3.2 Shared Memory

The module *test_rmem* and the user application *test_umem* accept four command line parameters as shown in columns 1 and 2 of Table 3 respectively.

Module	Application	Interpretation
sname	–s	Memory Name
dtype	–d	LabVIEW Data Type
nelm	–n	Number of elements
mode	–m	Access mode

TABLE 3: *command line parameters*

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:

```
% rmmod 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
test_umem: received int8 (0x01) msg=1 value=1
test_umem: received int8 (0x01) msg=2 value=1
test_umem: received int8 (0x01) msg=3 value=1
```

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
```

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
```

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.

4 The LabVIEW Interface

Before we discuss the implementation under LabVIEW, it is useful to discuss how LabVIEW handles data. In the *cintools* directory there is an include file, *extcode.h*, which defines a string *handle*:

```
typedef struct {
 int32 cnt;    /* number of bytes to follow */
 uChar str[1]; /* cnt bytes */
} LStr, *LStrPtr, **LStrHandle;

#define LStrBuf(sp) \
  (&((sp))->str[0])
#define LStrLen(sp) \
  (((sp))->cnt)
#define LStrSize(sp) \
  (LStrLen(sp)+sizeof(int32))
```

We see, then, that the canonical LabVIEW *LStrHandle* is no more complicated than a Pascal-type string definition which is an integer length followed by the address of the first character. The macro *LStrBuf* points to the first character in the string, *LStrLen* gives the length of the string and *LStrSize* gives the total allocation for the string.

There are *no* other such definitions in *extcode.h* but our *lvrtl.h* defines the others, such as the one for a 32-bit signed integer in an analagous manner:

```
typedef struct {
 int32 cnt;    /* number of int32s */
 int32 num[1]; /* start of array */
} Lint32, LI32, *Lint32Ptr,
  *LI32Ptr, **Lint32Handle, **LI32Handle

#define Lint32Buf(sp) \
  (&((sp))->str[0])
#define Lint32Len(sp) \
  (((sp))->cnt)
#define Lint32Size(sp) \
  (Lint32Len(sp)+sizeof(int32))

#define LI32Buf(sp) \
  (&((sp))->str[0])
#define LI32Len(sp) \
  (((sp))->cnt)
#define LI32Size(sp) \
  (LI32Len(sp)+sizeof(int32))
```

Using such handles, we can dynamically manipulate data.

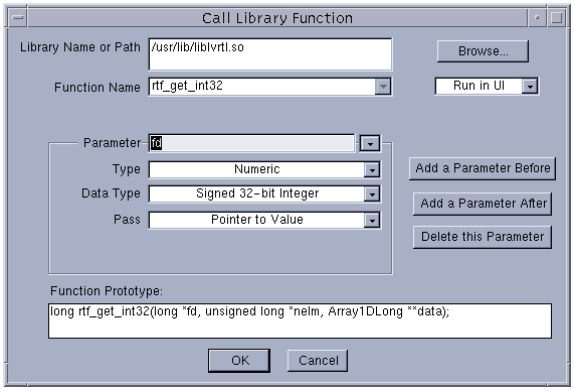

FIGURE 1: *Call Library Function Panel*

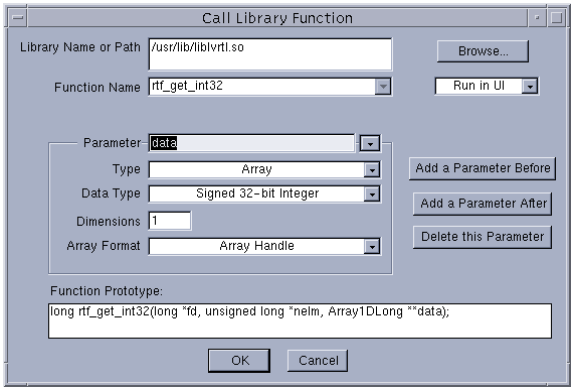

FIGURE 2: *Call Library Function Panel*

For example, we can design a VI called *rtf_get_int32* consisting of two numeric controls (file descriptor, *fd*, and number of data elements, *nelm*) and an array indicator (*data*). We can configure the *Call Library Function* panel to pass *fd* (and *nelm*) as pointers to values as shown in Figure 1 and the data array as an array handle as shown in Figure 2. Note that the function prototype is built as:

```
long rtf_get_int32( long *fd,
  unsigned long *nelm, Array1DLong **data );
```

This *Array1DLong* ⋆⋆ is, in effect, identical to our *Lint32Handle* defined above so our library code would look like this:

```
long rtf_get_int32 (
 long *fd,          // file descriptor
 uInt32 *nelm,      // number of elements
 Lint32Handle data  // data array
) {

 // initialize some local variables
 int err = 0, fb = (*nelm) * sizeof(uInt32);

 // return if fd is invalid
 if (*fd<0) return (*fd);

 // return if nelm is invalid
 if (*nelm<=0) return (-1L);

 // resize array to hold nelm values
 err=NumericArrayResize(
   iL,               // signed longs (int32)
   1,                // 1 dimension
   (UHandle *)&data, // address
   *nelm);           // number of elements
 if (err!=noErr) return (-err);

 // clear the re-sized array
 ClearMem((UPtr)*data,Lint32Size(*data));

 // read some data from fd
 err=read(*fd,Lint32Buf(*data),fb);
 if (err<=0||err!=fb) return (-err);

 // set the length
 Lint32Len(*data)=*nelm;

 // return any error
 return (long)err;
}
```

Note how we have used the LabVIEW functions *ClearMem* and *NumericArrayResize* [2] and the macros *Lint32Buf*, *Lint32Size* and *Lint32Len* to read the fifo.

5 The LabVIEW VIs

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

rtf_open.vi This is the open VI for real-time fifos;

rtf_close.vi This is the close VI for real-time fifos;

rtf_get_DTYPE.vi These are the *get* VIs for the DTYPE specified in column 2 of Table 1. For example, to get a signed 8-bit integer off a fifo, the appropriate VI is *rtf_get_int8*.

rtf_put_DTYPE.vi These are the *put* VIs for the DTYPE specified in column 2 of Table 1. For example, to put an unsigned 32-bit integer onto a fifo, the appropriate VI is *rtf_put_int32*.

rtf_read_DTYPE.vi These VIs bundle the open, read-loop and close VIs into an example for each specified DTYPE.

rtf_write_DTYPE.vi These VIs bundle the open, write-loop and close VIs into an example for each specified DTYPE.

mbuff_open.vi This is the open VI for shared memory.

mbuff_close.vi This is the close VI for shared memory.

mbuff_get_DTYPE.vi These are the *get* VIs for the DTYPE specified in column 2 of Table 1. For example, to get a single precision floating point from memory, the appropriate VI is *mbuff_get_float32*.

mbuff_put_DTYPE.vi These are the *put* VIs for the DTYPE specified in column 2 of Table 1. For example, to put a string into shared memory, the appropriate VI is *mbuff_put_string*.

mbuff_read_DTYPE.vi These VIs bundle the open, read-loop and close VIs into an example for each specified DTYPE.

mbuff_write_DTYPE.vi These VIs bundle the open, write-loop and close VIs into an example for each specified DTYPE.

test_lfifo.vi, 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 6*i* tarball.

Note that the *rtf_put_string* and *mbuff_put_string* VIs 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).

5.1 Real-time Fifo VIs

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_DTYPE* 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_DTYPE* accepts the file descriptor input, the number of data elements and an array of values of the appropriate data type and writes 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.

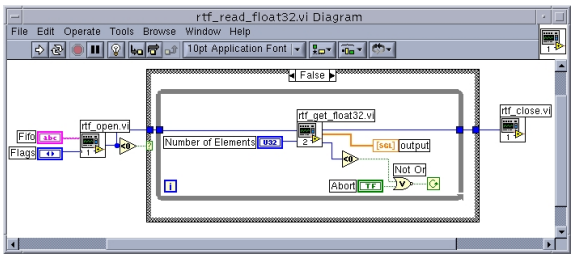

FIGURE 3: *rtf_read_float32 Diagram*

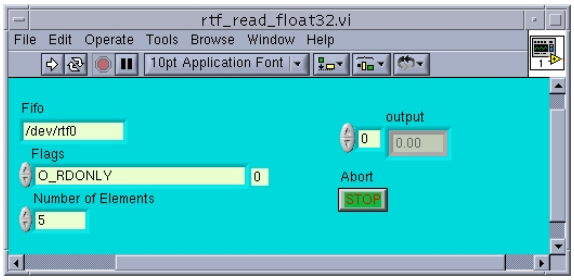

FIGURE 4: *rtf_read_float32 Front Panel*

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 6*i*) code for this example is shown in Figure 3 and the front panel is in Figure 4. As we can see, the G-code traps errors returned by the *rtf_open* and *rtf_get_float32*.

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

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

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.

5.1.1 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 5 we show the LabVIEW code for a non-blocking read of a single 8-bit integer from fifo 0. The associated front panel is in Figure 6. Note how we trap the return and indicate data is ready when the return value is positive.

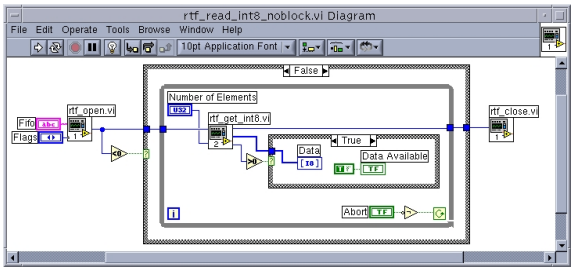

FIGURE 5: *rtf_read_int8 Diagram*

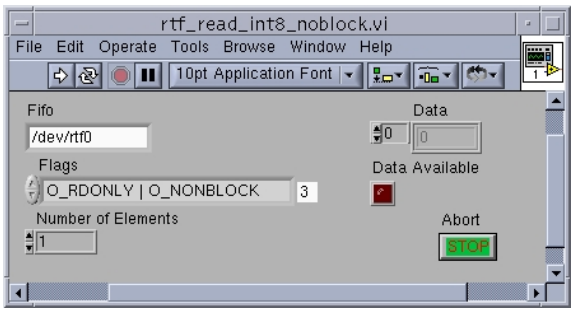

FIGURE 6: *rtf_read_int8 Front Panel*

We can run this VI after inserting the *test_rfifo* module:

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

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

5.2 Shared Memory VIs

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 is mitigated by making the VI re-entrant and requiring that calls to *mbuff_open* 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* always returns 0.

The *mbuff_get_*DTYPE 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_*DTYPE 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.

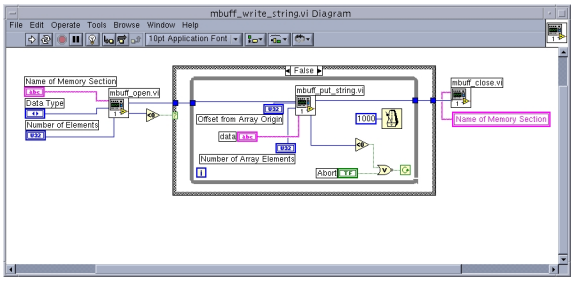

FIGURE 7: *mbuff_write_string Diagram*

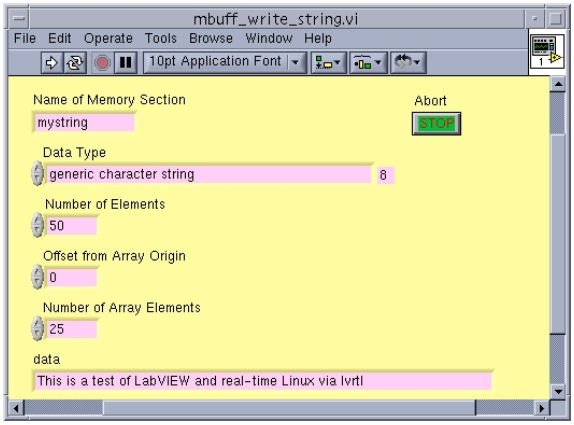

FIGURE 8: *mbuff_write_string Front Panel*

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 LabVIEW diagram for this example is shown in Figure 7 and the front panel is in Figure 8. As we can see, the G-code traps errors returned by the *mbuff_open* and *mbuff_put_string*.
To execute this example from LabVIEW, first insert the (test) module:

```
% rmmod 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 lvrtl', 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.

6 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 [3] and the examples given with this software to do this.
Alternatively, one could wire together the three VIs *rtf_get_int32*, *rtf_get_float32* and *rtf_get_string* with appropriate inputs. Clearly, this requires three reads of the fifo to completely obtain the structured data.

Acknowledgements. Linux is a registered trade mark of Linus Torvalds. LabVIEW is a trade mark of National Instruments Corporation. *NOAO* is operated by the *Association of Universities for Research in Astronomy Inc.* (AURA), under cooperative agreement with the *National Science Foundation* (NSF).

References

[1] Daly, P. N., Schumacher, G., Mills, D. and Ashe, M. C. 1999, *Real Time Linux at the NOAO*, Proc. 1st Real-time Linux Workshop, Vienna.

[2] Other, A. N. 2000, *Using External Code in LabVIEW*, July 2000 Edition, National Instruments Corporation, Part # 370109A-01. Vienna.

[3] Daly, P. N. 2000, *Interfacing Real-time Linux and LabVIEW*, Real Time Linux Documentation Project, 1, P. N. Daly and J. Küpper, eds., Real Time Linux Community Press.