

A simple control application with Real Time Linux

Peter Wurmsdobler

`peterw@thinkingnerds.com`

`http://www.thinkingnerds.com/nerds/peterw/peterw.html`

Abstract

In this paper a simple control application using real time linux is presented. A linux kernel module is responsible for getting a value from a DAQ-board, carrying out the control algorithm and outputting the result to the DAQ board. At the same time, values are put into shared memory for being displayed by a graphical user interface based on GTK+. Additionally, this user application can set control parameters and adjust a setpoint in shared memory, or start and stop the control process by FIFO-buffers.

1 Introduction

Real time linux is already used in many, moreorless specific real time applications. However, putting a control concept in practice can be difficult for somebody being not familiar with a RTOS. A control engineer is no necessarily a software engineer and vice versa. Thus there is a gap between software engineers and control engineers just applying the software.

Especially a control engineer who made some courses on digital control expects a measured or some measured values to be available at time k . From this point on it is his turn, because simulation is patient and any control algorithm can successfully be tested using MATLAB or some other simulation tools. Within a simulation the output of any wonderful algorithm is passed to the simulation model at time k , or in the best case at time $k + 1$ with a unit delay accounting for “some” computation time. One day, however, the control engineer has to do a real implementation, and in this moment, he has to bother himslef with “real” questions. How can I make a value be available for my algorithm at time k ? What’s about jitter? What happens, if the result is not ready at time $k + 1$?

The simpliest way will be to buy some proprietary solutions and trust in them. If the nuclear power plant explodes, a scapegoat can easily be found. However, if the engineer wants to understand what is going on or increase the performance at the same target hardware by optimizing for his application, then diving into the real time operating system and swimming in bits and bytes becomes indispensable. The real time application based on linux presented in this paper should make an access to real time linux easier for a control engineer by implementing a simple discrete time controller. Additionally, the resulting code can be used as template for more complex control and real time applications.

2 Control algorithm implementation

Assuming that there is some control algorithm already defined in a time discrete scheme, e.g. some general predictive control, an adaptive state space controller or whatever, a soft-

ware architecture for its implementation has to be defined. Time critical and less time critical tasks have to be separated, like the low level control feed back loop on the one hand, and some adaptation algorithm or some data display on the other, both with some interface between them for communication.

The application here implements only a simple discrete time controller with its parameters to be adjustable from a graphical user space application. Additionally, the user should have the possibility to trace step responses by entering different set points, and of course start and stop the control process by fancy buttons. In the following, it will be explained how this can be done with real time linux.

2.1 Application architecture

In real time linux the separation of tasks is realized by kernel modules doing the real control and real time work within a kernel thread, and some user space applications doing the rest. The communication can be done using first-in-first-out buffers (FIFOs), and shared memory. Thus, the software for the simple single-input-single-output (SISO) control is broken into two parts, one simple graphical interface being non-real time, and one small real time module:

- `xcontrol`, the X-Windows control interface,
- `rtl_control.o`, the real time linux control kernel module.

The logical structure from a software point of view is explained in Figure 1 with the user space application `xcontrol`, the board specific objects `rt_bmc1000.o` for the BMC1000 DAQ board and the real time control functions in `rtl_control.o` linked together to the kernel module `rtl_control.o` (both `rtl_control.o` and `rt_control.o` are used interchangeably in the next time). This is what the user has to make the code for.

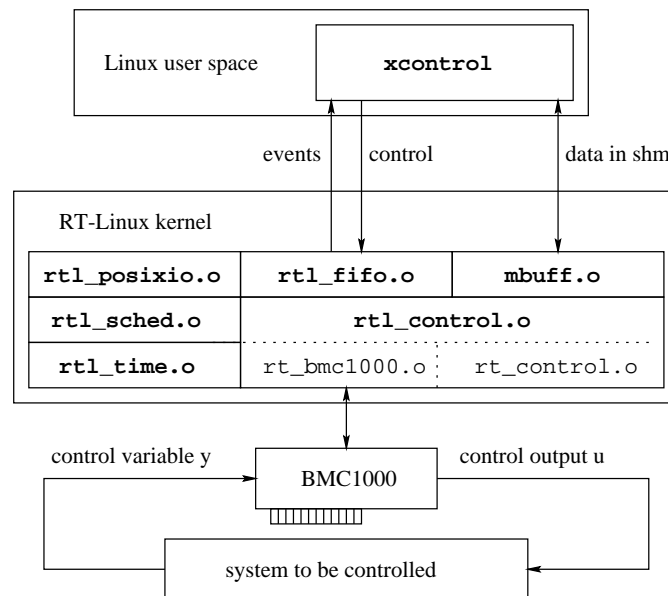


Fig. 1: Principle structure of the simple SISO control system with flow of data and signals.

Given that the kernel has been compiled with the real time option by NMT-RTL [1] four modules are inserted at first, the shared memory module by Tomasz Motylewski [2] for all data being input and output, `rtf_posixio.o` and `rtf_fifo.o` for FIFO buffers, `rtl_sched.o` and `rtl_time` for real time scheduling and timing, respectively.

2.2 Real time module - user space application interface

The only way the user space application can talk to the kernel modules is by means of device files, `/dev/mbuff` concerning shared memory and `/dev/rtf*` as for the FIFO buffers. Therefore, the interface between both software parts has to be defined. For this simple application shared memory for data with its variables is defined in a file like `shm.h` by

```
#define SHM_DEV_FILE ( "/dev/mbuff" )
#define SHM_NAME ( "control" )
#define SHM_SIZE ( sizeof(shm_t) )
typedef struct
{
    unsigned int N;          /* length of measured vectors */
    unsigned short int w;    /* {0,65535}, setpoint */
    unsigned short int u[SAMPLES]; /* {0,65535}, control current */
    unsigned short int y[SAMPLES]; /* {0,65535}, measured position */
    int a[LENGTH];          /* scaled control denominator of ORDER+1 */
    int b[LENGTH];          /* scaled control numerator of ORDER+1 */
}
shm_t;
```

Messages can be passed to and from the real time module causing a message handler to be executed both in the real time module and the user application. FIFO messages used for control and event handling are defined in a file like `fifos.h` by

```
#define FIFO_SIZE (5000) /* byte size for fifo buffer */
#define CONTROL_FIFO (0) /* for passing messages to rt_control */
#define CONTROL_FILE "/dev/rtf0"
#define EVENT_FIFO (1) /* for triggering events in Xcontrol */
#define EVENT_FILE "/dev/rtf1"
#define START_CONTROL ('a') /* xcontrol -> rtl_control */
#define STOP_CONTROL ('b') /* xcontrol -> rtl_control */
#define TRIGGER_MEASURE ('c') /* xcontrol -> rtl_control */
#define MEASURE_READY ('a') /* rtl_control -> xcontrol */
#define INVALID_MESSAGE ('b') /* rtl_control -> xcontrol */
```

In fact, for data to be measured, FIFOs could be used instead of shared memory, but this is eventually more of interest if all data have to be traced and displayed. Since in this application only step responses are taken, the shared memory approach seems to be more reasonable.

2.3 Real time module

As the DAQ-board module `rt_bmc1000.o` is concerned, the basic functions are the `init` and `release`, and the analogue `set` and `get` functions being called from the measurement object `rt_control.o` and being defined in some file like `rt_bmc1000.h`:

```
extern int rt_bmc1000_init(void);
extern void rt_bmc1000_release(void);
extern unsigned short int rt_bmc1000_aget( void );
extern void rt_bmc1000_aset( unsigned short int channel,
                             unsigned short int value );
extern void rt_bmc1000_mux_select( unsigned int channel,
                                   unsigned int range_code );
```

These functions have to be programmed, if there is no module available on the net or by the manufacturer. Based upon these functions, `rt_control.o` is responsible for the data and control flow by using both shared memory and FIFO buffers to communicate with the graphical user interface. To start with, `rt_control.o` defines the control thread and some variables and a small inline function for sending messages like:

```
pthread_t control;          /* THE control thread, or task */
static int measure;        /* flag if measurement is on */
static shm_t *shm;         /* shared memory (sic!) */
static unsigned int index; /* index in shared memory */
static int e[LENGTH];      /* control deviation buffer */
static int u[LENGTH];      /* control signal buffer */
static unsigned int k;     /* index k of actual value */
static unsigned short int Y; /* THE measured value, input */
static unsigned short int U; /* THE control current, output */
static unsigned short int W; /* THE setpoint */

inline void rt_control_event_msg( unsigned char message )
{
    rtf_put( EVENT_FIFO, &message, 1 );
}
```

What the control engineer has to do then, is to write some C-code for his general and everything predicting control algorithm. This is done here for a simple IIR filter as (just look at the periodic part):

```
void *rt_control_thread( void *t )
{
    unsigned int l,h;

    while(1) /* this is the periodic part */
    {
        pthread_wait_np(); /* yield to scheduler */

        /* AND HERE STARTS THE CONTROL ALGORITHM */

        /* get the value from the board at channel preset in init_module
        */
        Y = rt_bmc1000_aget();

        /* 32768 is cancelled out if setpoint is {0,65535}.
        */
        e[k] = (int) ( W - Y ); /* this is the control deviation ! */

        /* THE control algorithm, a simple IIR filter
        */
        u[k] = (shm->b[0]) * e[k];
```

```

for ( l=1; l<LENGTH; l++ )
{
    h = ( LENGTH + k - 1 ) % LENGTH;
    u[k] += ( shm->b[l] ) * e[h] - ( shm->a[l] ) * u[h ] );
}
u[k] /= (shm->a[0]);
k = (k+1) % LENGTH;

/* control variable output, 32768 is added to shift to {0,65535}.
*/
U = (unsigned short int)( u[k] + 32768 );

/* set the control current on board and the loop is closed
*/
rt_bmc1000_aset( OUTPUT_CHANNEL, U );

/* if a measurement of step response is set,
*/
if (measure == 0)
{
    /* we store the values in shm
    */
    shm->y[index] = Y;
    shm->u[index] = U;

    /* increase index and stop measurment of step response
    */
    index++;
    if (index == (shm->N) )
    {
        rt_control_event_msg( MEASURE_READY );
        measure = -1;
    }
}
}
}

```

Note that this is a thread which is always alive once initialized, not a function like an interrupt service routine being called from some instance. In contrast, the thread can yield control to the scheduler or being resumed by it if the time is mature. It is nice, if the control thread is defined, but the module should be able to react to some messages by a function like

```

static int rt_control_message_handler(unsigned int fifo)
{
    unsigned char message;

    while( rtf_get( fifo, &message, 1 ) > 0)
    {
        switch(message)
        {
            case START_CONTROL:
                rt_control_start();
                break;
            case STOP_CONTROL:
                rt_control_stop();
                break;
        }
    }
}

```

```

        case TRIGGER_MEASURE:
            rt_control_trigger_measure();
            break;
        default:
            rt_control_event_msg( INVALID_MESSAGE );
    }
}
return 0;
}

```

by executing the following functions (explained later):

```

static void rt_control_start(void)
{
    /* cleaning the buffers and reset k
    */
    for (k=0; k<LENGTH; k++)
    {
        e[k] = 0;
        u[k] = 0;
    }
    k = 0;

    /* start the control process by making the thread periodic
    * with the sample time TS times 1000 to yield ns.
    */
    pthread_make_periodic_np( control, gethrtime(), (TS*1000) );
}

static void rt_control_stop(void)
{
    /* stop the control process by setting resume time to
    * infinity and period to 0, like in thread_create_np.
    */
    pthread_make_periodic_np( control, HRTIME_INFINITY, 0 );
    rt_bmc1000_aset( OUTPUT_CHANNEL, 32768 );
}

static void rt_control_trigger_measure(void)
{
    /* reset index and enable one shot measure, then set the
    * set point. This is for tracing step responses.
    */
    index = 0;
    measure = 0;
    W = shm->w;
}

```

Since all these basic functions have been defined, `insmod rtl_control.o` will make the `init_module` function to be called:

```

int init_module(void)
{
    int l;
    pthread_attr_t attr;          /* control thread attributes */
    struct sched_param sched_param; /* control thread scheduler parameter */
}

```

```

/* Initialise the PCL818 DAQ board and set input channel
*/
if ( rt_bmc1000_init() )
{
    printk( "%s Calling rt_pc1000_init failed\n", RT_FILTER_ID );
    return -ENOMEM;
}
rt_bmc1000_mux_select( INPUT_CHANNEL, RT_BMC1000_ADC_PM10 );

/* Initialise rt-fifos
*/
if ( rtf_create( CONTROL_FIFO, FIFO_SIZE ) < 0 )
{
    printk( "%s Could not install fifo %d\n",
            RT_FILTER_ID, CONTROL_FIFO );
    return -ENOMEM;
}
if ( rtf_create( EVENT_FIFO, FIFO_SIZE ) < 0 )
{
    printk( "%s Could not install fifo %d\n",
            RT_FILTER_ID, EVENT_FIFO );
    return -ENOMEM;
}

/* Initialise message handler
*/
if ( rtf_create_handler( CONTROL_FIFO, rt_control_message_handler ) )
{
    printk( "%s Could not install rt_control_message_handler\n",
            RT_FILTER_ID );
    return -ENOMEM;
}

/* Initialise shared memory using Tomasz Motylewski's mbuff.o
*/
if ( shm_allocate( SHM_NAME, SHM_SIZE, (void **) &shm ) < 0 )
{
    printk( "%s Initialising shared memory failed\n", RT_FILTER_ID );
    return -ENOMEM;
}

shm->w = 32535;
shm->N = SAMPLES;
shm->a[0] = 1;
shm->b[0] = 1;
for (l=1; l<LENGTH; l++)
{
    shm->a[l] = 0;
    shm->b[l] = 0;
}

/* and now we initialize the kernel thread, with attributes
* scheduler parameters and then create it.
*/
pthread_attr_init( &attr );
pthread_attr_setcpu_np( &attr, 0 );
sched_param.sched_priority = 1;
pthread_attr_setschedparam( &attr, &sched_param );
if ( pthread_create( &control, &attr, rt_control_thread, (void *)1 ) )

```

```

    {
        printk( "Initializing real time control thread failed" );
        return -ENOMEM;
    }
    printk( "%s Init module sucessfull\n", RT_FILTER_ID );
    return(0);
}

```

This function initiates the DAQ-board, creates the FIFOs, installs the message handlers for the control FIFO, initiates shared memory, and, it creates the real time control thread. Note that after creation (look at `rtl_sched.c` the thread is already executed the first time and stops at `pthread_wait_np()` yielding again to the real time scheduler.

After all this is done the module falls asleep and waits for something to happen, like a message arriving from the FIFO. In this case, the application sends control messages to a FIFO and the real time module will execute the appropriate functions in its message handler, e.g. to start a control or do something other defined for each message. In particular, the start control function `rt_control_start()` will set the control thread periodic at a given period which corresponds to THE sample time.

Hence the function `pthread_make_periodic_np(control, gethrtime(), (TS*1000))` tells the real time scheduler to schedule the control thread periodically. What happens then? The control thread resumes after the last `pthread_wait_np()` and does his duty until the next `pthread_wait_np()`.

By calling `pthread_make_periodic_np(control, HRTIME_INFINITY, 0)` the real time scheduler is told to schedule the task after a “long long” time.

The mechanism of starting, executing and stopping the control process could be even done with interrupts of a timer of a DAQ board. In this case, it is the interrupt service routine does a wakeup of the control thread, not the scheduler.

If `rtl_control` should be stopped, the entire process can be killed with a `rmmod rtl_control` by which the `cleanup_module` of the respective module is called. In this function all resources being allocated are released by calling

```

void cleanup_module(void)
{
    /* Delete control thread
    */
    pthread_delete_np( control );

    /* Release shared memory using Tomasz Motylewski's mbuff.o
    */
    shm_deallocate( shm );

    /* Release rt-fifos
    */
    rtf_destroy( CONTROL_FIFO );
    rtf_destroy( EVENT_FIFO );

    /* Release the PC1000 DAQ board
    */
    rt_bmc1000_release();

    printk( "%s Cleanup module sucessfull\n", RT_FILTER_ID );
}

```

2.4 User space application

As the visible main application, `xcontrol` provides in general a main window (Fig. 2) with some sub-windows (Fig. 2) showing the inputs and outputs of the simple SISO control, some control buttons in order to start and stop control, and some text fields in order to enter sample time and controller parameters. This X-Windows interface is programmed using the GTK+ widget set library [3] in combination with a scientific plot widget [4].

At start-up this applications maps the shared memory in its space, and opens the FIFO buffers by simple `open` commands as for any unbuffered file. After this, all GTK widgets are created, like the start button or the plots for the control variable and the control output (see the GTK+ manual how to do this, or just use `glade` [5] as first step). GTK+ also allows to define a handler to act on a file, in this case for the event FIFO with date arriving from the real time module, which can be done similarly, too.

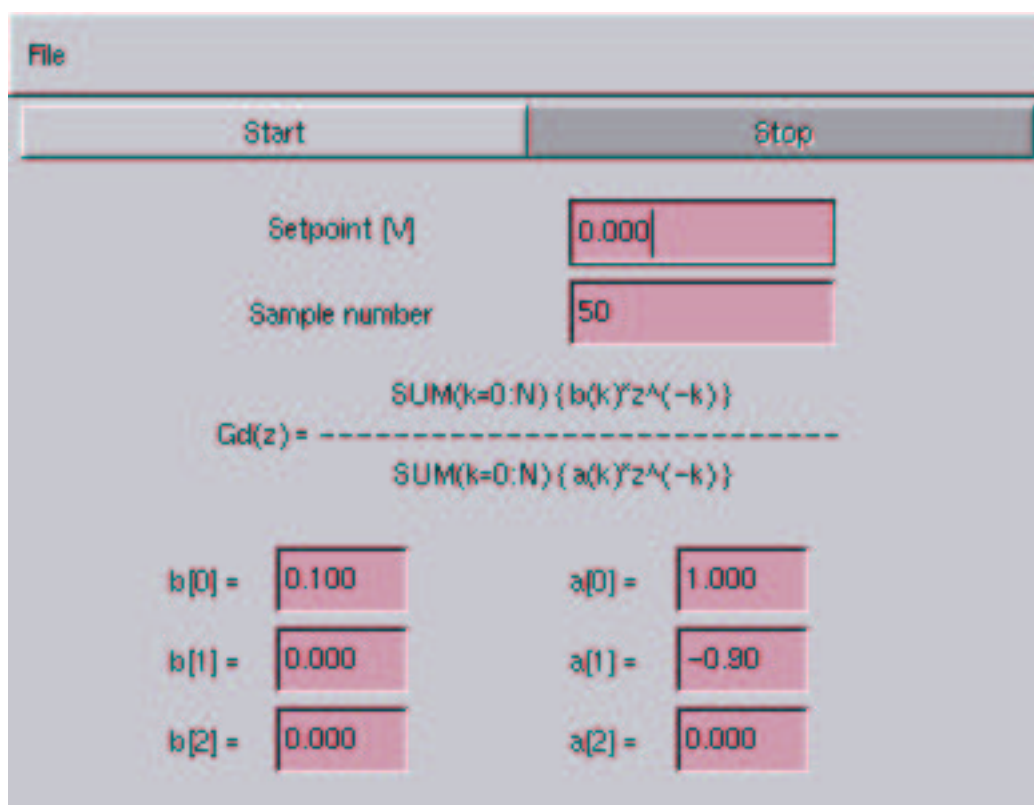


Fig. 2: *The graphical user interface.*

If the user then presses the start button, some values are calculated and put into shared memory. Afterwards, a `START_CONTROL` is sent to the real time module and the application is waiting. If the user presses the stop button, a stop message is sent. This message will be received by the real time module and the appropriate function will suspend the periodic thread.

3 Conclusion

The application presented here is neither a high end one in terms of c-programming, nor in terms of control engineering. Arriving at the end of this paper, however, a control engineer even

not too familiar with `c` should be able to understand the concept of implementing his control algorithm in real time linux.

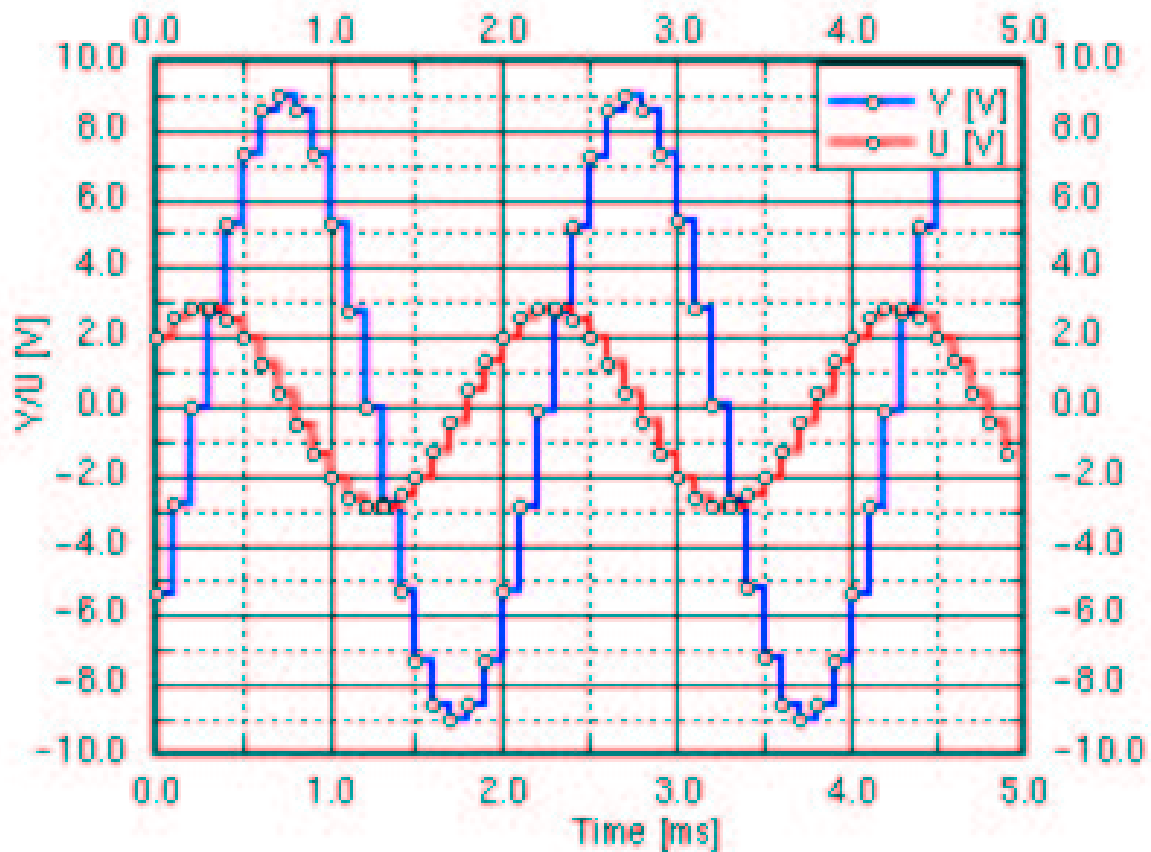


Fig. 3: *The input Y and output U , only with a sinusoidal input of 500 Hz at 10 kHz sampling rate.*

References

- [1] rt-linux, THE real time linux.
<http://www.rtlinux.org>, <http://www.fsmlabs.com>
- [2] mbuff, a shared memory module.
<http://crds.chemie.unibas.ch/PCI-MIO-E/mbuff-0.6pre7.tar.gz>
- [3] GTK+ the graphical toolkit.
<http://www.gtk.org/>
- [4] GTKplot, a widget for scientific plots.
<http://www.ifir.edu.ar/grupos/gtk/>
- [5] A GTK+ user interface builder.
<http://glade.pn.org>