A Sample Control Application Employing RTLinux

Peter Wurmsdobler
Centre de Transfert des Microtechniques
39, avenue de l'observatoire, 25000 Besançon, FRANCE
voice: +33.3.81.47.70.20 fax: +33.3.81.47.70.21
WWW: `http://www.ctm-france.com`
E-mail: `peterw@ctm-france.com`

19th February 2001

Contents

1 Introduction

A control engineer usually focuses on design issues, by simply assuming a measured or some measured values to be available at time k, and the output of any complex control algorithm to be passed to the plant at time k, too, or in the best case at time $k+1$ with a unit delay accounting for "some" computation time. The real implementation of this control design into a real-time operating system (RTOS) running on a given target hardware is then carried out either by proprietary software which conceals internal functioning, or by directly writing the code. The latter solution sometimes causes a "fears", because diving into the real-time operating system and swimming in bits and bytes becomes indispensable.

One reason is that there are no templates and entry points to real-time programming available due to the fact that most real-time systems are proprietary. Since GNU/Linux is open source as are the hard real-time modifications to the Linux kernel, an access to programming is easier. Therefore, an open source implementation of a simple real-time control applications based on RTLinux [5] is presented in this paper. This should make the engineer's access to any real-time application and RTLinux possible.

The aim of this text is to serve as introduction and reference to programming real-time applications employing real-time Linux as underlying operating system. Following the open source idea, the presented code can be used as template for more complex control, measure or any other real-time application.

2 A sample control application

Assuming that there is some control algorithm already defined in a discrete time scheme, e.g. some general predictive control, an adaptive state space controller or whatever, a software 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 adaption algorithm or some data display on the other, both with some interface between them for communication.

The control application here is an implementation of a simple second order SISO discrete time controller which should of course run in real-time, but with its parameters being adjustable from a graphical interface. Additionally, the user should have the possibility to trace step responses by entering different setpoints, and of course start and stop the control process.

2.1 Application architecture

In this application the separation of tasks is realized by a kernel module initialising a real-time thread which does the actual control work, and some user interface running in the non-real-time Linux user space doing the rest. (In contrast to some proprietary OSes, Linux makes use of processor modes with two different levels of operation, the kernel or superuser mode and the user modes with less privileges). Thus, the software for the control application is broken into three parts,

xcontrol: the X-Windows based graphical user interface,

rtl_control.o: Linux kernel module initialising the real-time thread.

rtl_board.o: Linux kernel module containing low level DAQ board driver functions.

Some RTLinux kernel modules are used for this application, `rtl_time.o` and `rtl_sched.o`, for "hard real" timing and scheduling, `rtl_fifo.o` for messages passed to FIFO buffers and the shared memory module `mbuff.o` [3] for data being exchanged between both `xcontrol` and `rtl_control.o`. The board specific object `rt_board.o` provides the necessary functions for the DAQ-board and has to be inserted to the kernel, too. Last but no least, the kernel module `rtl_control.o` implements the real-time controller. The user space application `xcontrol` is a normal Linux process as it is depicted in Fig. 2.1.

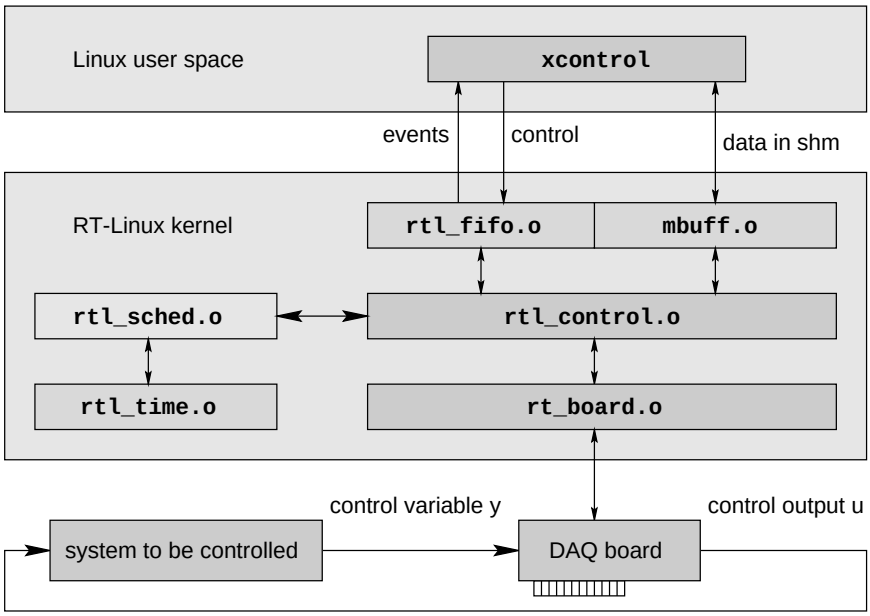

Figure 1: *Principle structure of the simple SISO control system with data and signal flow.*

In the following, the C-code of the real-time part is presented for the interested reader, the shared memory and FIFO definitions and all functions required to run the hard real-time application.

2.2 Real time module - user space application interface

The only way the user space application can talk to the kernel modules in Linux is by means of device files, `/dev/mbuff` for shared memory and `/dev/rtf*` for the FIFO buffers. Therefore, the interface between both software parts has to be described. For this simple application shared memory for data with its variables is defined in the file `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 for data tracing        */
  unsigned short int W;          /* user setpoint                  */
  unsigned short int w[SAMPLES]; /* setpoint values                */
  unsigned short int u[SAMPLES]; /* controler output               */
  unsigned short int y[SAMPLES]; /* control variable               */
  int a[LENGTH];                 /* digital controller denominator */
  int b[LENGTH];                 /* digital controller numerator   */
  }
  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. The FIFO messages used for control and event handling are defined in the file `fifos.h` by

```
#define FIFO_SIZE       (5000)
#define CONTROL_FIFO    (0)
#define CONTROL_FILE    "/dev/rtf0"
#define EVENT_FIFO      (1)
#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 date has to be traced and displayed. Since in this application only step responses are stored, the shared memory approach seems to be more reasonable.

2.3 Real time DAQ board driver

As for the DAQ-board module `rt_board.o`, the basic functions are the init and release, and the analogue set and get functions being called from the measurement object `rt_control.o`. These functions are defined in the file `rt_board.h`:

```
static int rt_board_init(void);
static void rt_board_release(void);
extern unsigned short int rt_board_aget( unsigned short int channel );
extern void rt_board_aset( unsigned short int channel, unsigned short int value );
```

These functions have to be programmed, if there is no module available on the net or by the manufacturer.

2.4 Real time module

Based on the real-time DAQ board functions, `rt_control.o` is responsible for the data and control flow. To start with, `rt_control.o` defines the control thread, some variables and a small in-line function for sending messages like:

```
void *rt_control_function(void *t)   /* control thread function     */
pthread_t control_thread;            /* control thread structure    */

int measure;                         /* measurement flag             */
shm_t *shm;                          /* shared memory               */
unsigned int index;                  /* index in shm                */
int e[LENGTH];                       /* deviation buffer           */
int u[LENGTH];                       /* controller output buffer    */
unsigned int k;                      /* current index               */
unsigned short int W;                /* control variable setpoint   */
unsigned short int Y;                /* measured control variable   */
unsigned short int U;                /* controler output            */
```

What the control engineer has to do then, is to write some C-code for his control algorithm. This is done here for a simple IIR-filter within the thread's function as:

```
void *rt_control_function( void *t )
  {
  unsigned char message;
  unsigned int l,h;

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

    /* get the value from the board
     */
    Y = rt_board_aget(INPUT_CHANNEL);

    /* control deviation
     */
    e[k] = (int) ( W - Y );

    /* THE control algorithm, a simple filter
     */
    u[k] = (shm->b[0]) * e[k];
    for ( l=1; l<LENGTH; l++ )
      {
      h = ( LENGTH + k - l ) % LENGTH;
      u[k] += ( (shm->b[l]) * e[h] - (shm->a[l]) * u[h] );
      }
    u[k] /= (shm->a[0]);
    if ( u[k] < -32768 ) u[k]  = -32768;
    if ( u[k] >  32767 ) u[k]  =  32767;

    /* control variable output
     */
```

```
    U = (unsigned short int)( u[k] + 32768 );

    /* set the control variable
     */
    rt_board_aset( OUTPUT_CHANNEL, U );

    /* if a measurement of step response is set,
     */
    if (measure == 0)
      {
      /* we store the values in shm
       */
      shm->w[index] = W;
      shm->y[index] = Y;
      shm->u[index] = U;
      index++;
      if (index == (shm->N) )
        {
        rtf_put( EVENT_FIFO, &message, 1 );
        measure = -1;
        }
      if (index == 1) W = shm->W;
      }
    k = (k+1) % LENGTH;
    }
  }
```

Note that this is a thread which is always alive once initialised, not a function like an interrupt service routine being called from some instance. In contrast, the thread passes control to the scheduler and is resumed periodically. In order to start the scheduler, the message handler

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

  while( rtf_get( fifo, &message, 1 ) > 0 )
    {
    switch(message)
      {
      case START_CONTROL:
        /*
         * start control by making the thread periodic with the sample time TS
         */
        pthread_make_periodic_np( control_thread, gethrtime(), TS );
        break;

      case STOP_CONTROL:
        /*
         * stop control by setting resume time to infinity and period to 0
         */
        pthread_make_periodic_np( control_thread, HRTIME_INFINITY, 0 );
        break;

      case TRIGGER_MEASURE:
```

```
      /*
       * reset index, enable tracing
       */
      index = 0;
      measure = 0;
      break;

    default:
      rtf_put( EVENT_FIFO, &message, 1 );
    }
  }
return 0;
}
```

is installed listening on a control FIFO. Finally, the mandatory `init_module(void)` function declares what to do once inserted, and the `cleanup_module(void)` what has to be done when the module is removed:

```
int init_module(void)
  {
  int l;
  pthread_attr_t attr;            /* thread attributs    */
  struct sched_param param;       /* scheduler parameter */

  /* Initialise rt-fifos
   */
  if ( rtf_create( CONTROL_FIFO, FIFO_SIZE ) < 0 )
    {
    printk( "Could not install control fifo\n" );
    return -ENODEV;
    }
  if ( rtf_create( EVENT_FIFO, FIFO_SIZE ) < 0 )
    {
    printk( "Could not install event fifo\n" );
    return -ENODEV;
    }

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

  /* Initialise shared memory
   */
  if ( shm_allocate( SHM_NAME, SHM_SIZE, (void **) &shm ) < 0 )
    {
    printk(" Init shared memory failed\n ");
    return -ENOMEM;
    }

  /* and now we initialise the kernel thread.
```

```
   */
  pthread_attr_init( &attr );
  pthread_attr_setcpu_np( &attr, 0 );
  sched_param.sched_priority = 1;
  pthread_attr\setschedparam( &attr, &param );
  if ( pthread_create( &control_thread, &attr, rt_control_function, (void *)1 ) )
    {
    printk( "Init control thread failed\n" );
    return -EAGAIN;
    }

  printk( "Init module sucessfull\n" );
  return 0;
  }

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

    /* Release shared memory
     */
    shm_deallocate( shm );

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

    printk( "Cleanup module sucessfull\n" );
    }
```

2.5 User space application

As the visible main application, `xcontrol` (Fig. 2) provides in general a main window with some sub-windows showing the inputs and outputs of the simple SISO control, some buttons in order to start and stop control, and some text fields in order to enter setpoint and controller parameters. This X-Windows interface is programmed using the GTK+ widget set library [4] in combination with a scientific plot widget [1].

2.6 Compiling and operation

Given that RTLinux [5] has been installed correctly and the kernel has been compiled with the real-time option, both the real-time kernel module and the user space application are compiled using `gcc` with the appropriate options.

Hence, the real-time module is inserted into the kernel by `insmod rtl_control.o` calling the function `init_module` which initiates the DAQ-board, creates the FIFOs, installs the message handler for the control FIFO, initiates shared memory, and, lastly creates the real-time control

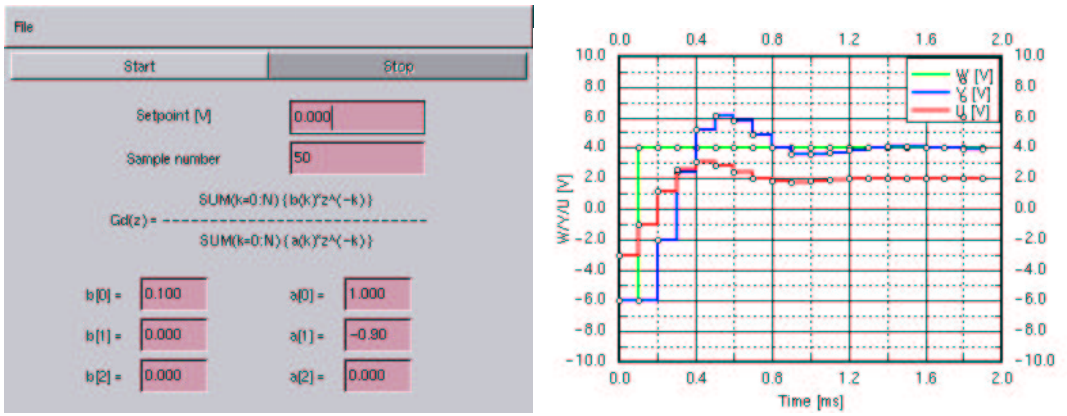

Figure 2: *The graphical user interface (left) and the measurement window (right) with setpoint W, measurement variable Y and control variable U, for a proportional plant $K = 2$ and the controller running at 10 kHz sampling rate.*

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 control FIFO. If `rtl_control` should be stopped, the entire process can be killed with a `rmmod rtl_control`.

After the kernel module has been inserted successfully, the user space application is simply started by executing `xcontrol`. At start-up this applications maps the shared memory 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. GTK+ also allows to define a handler to act on a file descriptor, in this case for the event FIFO with messages arriving from the real-time module.

If the user presses the start button, some initial values are calculated and put into shared memory. Afterwards, a `START_CONTROL` is sent to the real-time module through the control FIFO and the application is waiting. In contrast, the real-time module will execute the appropriate function in its message handler, i.e. the start control function `rt_control_start()` which will set the control thread periodic at a given period corresponding to the sampling time.

The function call `pthread_make_periodic_np( control, gethrtime(), TS )` tells the real-time scheduler to schedule the control thread periodically. Once scheduled, the control thread resumes after the last `pthread_wait_np()` and continues until the next `pthread_wait_np()`.

In order to trigger a step response, a user can enter a setpoint which will change the respective values in shared memory and then send the message `TRIGGER_MEASURE`. On its side, the real-time module will execute `rt_control_trigger_measure()` which causes the control thread to write data in shared memory. If the tracing is finished, a message `MEASURE_READY` will be sent through the event FIFO to the user space program which will act according to a message handler, e.g. get data from shared memory and display the step response.

If the user presses the stop button, the message `STOP_CONTROL` is sent to the control FIFO. This message will be received by the real-time module and the function `rt_control_stop()` will stop the periodic thread by calling the function `pthread_make_periodic_np( control, HRTIME_INFINITY, 0 )` which tells the real-time scheduler to schedule the thread only after a "long long" time.

3 Conclusion

A simple second order discrete time controller has been implemented employing RTLinux as hard real-time operating system with a kernel thread for the control algorithm itself and a graphical user interface for parameter adjustment and display. This application is not a highly sophisticated one, neither in terms of C, 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 idea of implementing a control algorithm in real-time Linux and to build his own applications.

References

[1] Feiguin, Adrian E. (1999). Gtkplot, a high quality scientific plot widget, `http://www.ifir.edu.ar/grupos/gtk/`.

[2] Mantegazza, Paolo (1999). RTAI, real-time application interface, Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Italy, `http://www.aero.polimi.it/`.

[3] Motylevski, Thomasz (1999). mbuff, a kernel shared memory driver, `http://crds.chemie.unibas.ch/`.

[4] Peter Mattis, Spencer Kimball, Josh MacDonald (1999). Gtk+, the GIMP ToolKit, `http://www.gtk.org/`.

[5] Yodaiken, Victor (1999). RTLinux, real-time Linux, FMSLab, Soccoro, New Mexico, USA, `http://www.rtlinux.org/`.

A V2 API

The presented application is based upon the v2 of RTLinux.

A.1 Timing functions

A basic requirement for all scheduling is the high resolution timer with the respective get time functions.

```
typedef long long hrtime_t;
hrtime_t gethrtime(void);
```

A.2 Interrupt functions

In order to register an interrupt for the hard real-time executive the following functions must be used.

```
int rtl_request_global_irq(unsigned int irq,
    unsigned int (*handler)(unsigned int, struct pt_regs *));
void rtl_hard_enable_irq(unsigned int ix);
void rtl_hard_disable_irq(unsigned int ix);
int rtl_free_global_irq(unsigned int irq);
```

A.3 Thread functions

As smallest schedulable entities threads can be created, handled and deleted.

```
typedef RTL_THREAD_STRUCT *pthread_t;
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg);
int pthread_make_periodic_np (pthread_t p,
    hrtime_t start_time, hrtime_t period);
int pthread_suspend_np (pthread_t thread);
int pthread_wakeup_np (pthread_t thread);
int pthread_wait_np(void);
void pthread_exit(void *retval);
int pthread_delete_np (pthread_t thread);
```

A.4 Shared memory functions

If the user wants to share memory between user space and the kernel threads theses functions have to be used.

```
int shm_allocate(const char *name, unsigned int size, void **shm);
int shm_deallocate(void * shm);
```

A.5 FIFO functions

FIFOS can be used to exchange data between user and real-time space.

```
int rtf_create(unsigned int fifo, int size);
int rtf_create_handler(unsigned int fifo,
    int (*handler)(unsigned int fifo) ;
int rtf_put(unsigned int fifo, void * buf, int count ):
int rtf_get(unsigned int fifo, void * buf, int count );
int rtf_destroy(unsigned int fifo);
```