A SAMPLE CONTROL APPLICATION EMPLOYING REAL TIME LINUX

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
E-mail: peterw@cetehor.com

Abstract: In this paper a sample control application employing realtime linux as hard realtime operating system is presented. More precisely, a kernel module is in charge of getting a value from a DAQ-board, carrying out the control algorithm and putting the result out to the DAQ board. At the same time, these values are put into shared memory for display by a graphical user interface based on GTK+. This user application can set control parameters and adjust a setpoint in shared memory, as well as start and stop the control process through FIFO-buffers.

Keywords: realtime operating system, linux, control application

1. INTRODUCTION

Linux has become a powerful operating system challenging the OS and computer market, because it is open source and adaptable to different hardware and computing problems. In particular, there are some groups spread over the world developing modifications to the linux kernel in order to provide a hard realtime operation system, e.g. FSMLab's RTlinux (Yodaiken, 1999) or DIAPM's RTAI (Mantegazza, 1999). Both are already used in many, moreorless specific and sophisticated realtime applications which mainly are done by software engineers.

In contrast, 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 realtime 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 indispensible.

Since Linux is open source as are the hard realtime modifications, so an open source implementation of a sample control application based on real time linux is presented in this paper. This should make the control engineer's access to any controller implementation and realtime linux easier. Additionally, the resulting code can be used as template for more complex control and realtime applications.

2. RTLINUX - THE FMSLAB REALTIME LINUX IMPLEMENTATION

In this section one realtime variant for the linux operating system is presented briefly, RTLinux. The core idea behind RTLinux is to run a full featured operating system as one thread of a real time executive. In a logical sense, "tasks" and "interrupts" are seperated in two classes, the realtime ones running directly on behalf of the executive, and the non-realtime ones being

executed within the common operating system, with some means of communication between these two priority spaces. "realtime" in this context is meant to be *hard realtime* as its deadlines have to be met in any case. In order to meet this goal, the general rule is to keep as many services in non-realtime linux as possible if they are not inherently hard realtime, like data display or storage.

As general purpose operating system Linux optimizes *average* performance, but is not appropriate for hard realtime. The RTLinux modifications to the kernel, however, put a thin layer underneath the linux operating system. This core hard realtime mechanism takes over control of the low level interrupt handling from Linux: in some cases Linux kernel code is modified to make this takeover work smoothly. The low latency interrupt handlers cannot be premepted by linux actin then only on emulated interrupts. All other functionalities and communication means as schedulers, timers, POSIX-io, FIFOs, shared memory, semaphores and mutexes can be added by inserting a kernel module at runtime.

With the core functionality realtime ISRs can be installed already as simple real time "application". Once the modules for time and scheduling are inserted into the kernel, both "task" and "interrupt" priority spaces are created. From now on the thin hard realtime layer constitutes a single process per CPU running on the bare computer hardware with linux being in the following just a low priority thread among realtime threads and interrupt service routines (ISRs). Linux as "task" works as usual in its own process space as illustrated by Figure 1.

Form the users's point of view, the computer can be used as full featured workstation in normal mode. In case of a realtime event, however, linux is completely preemted and the realtime threads or interupt service routines are treated immediately. Once they have finished and there is nothing else to do in the "hard realtime world", linux is scheduled again. In order to make the RTLinux idea understand, a simple application using a minimum API is presented in the following.

3. CONTROL ALGORITHM IMPLEMENTATION

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 realtime, 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.

3.1 *Application architecture*

In this application the separation of tasks is realized by a kernel module initializing a real time thread which does the actual control work, and some user interface running in the non-realtime 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 two parts,

- `xcontrol`, the X-Windows based graphical user interface,
- `rtl_control.o`, linux kernel module initializing the realtime thread.

Four 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` (Motylevski, 1999) for data being being exchanged between both `xcontrol` and `rtl_control.o`. Based on functions offered by these modules, the board specific object `rt_bmc1000.o` for the BMC1000 DAQ-board and the object containing the realtime control functions `rt_control.o` are linked together to the kernel module `rtl_control.o`. The user space application `xcontrol` is a normal linux process as it is depicted in Fig. 2.

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

3.2 *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

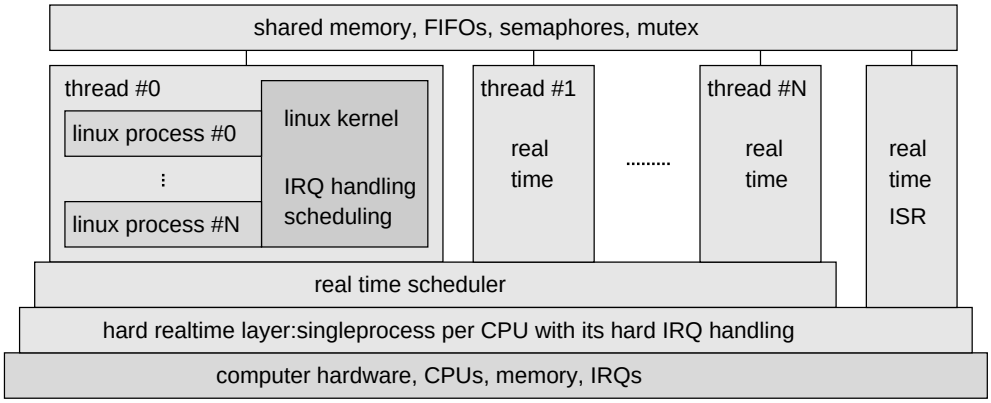

Figure 1. Principle structure of RTLinux realtime implementation.

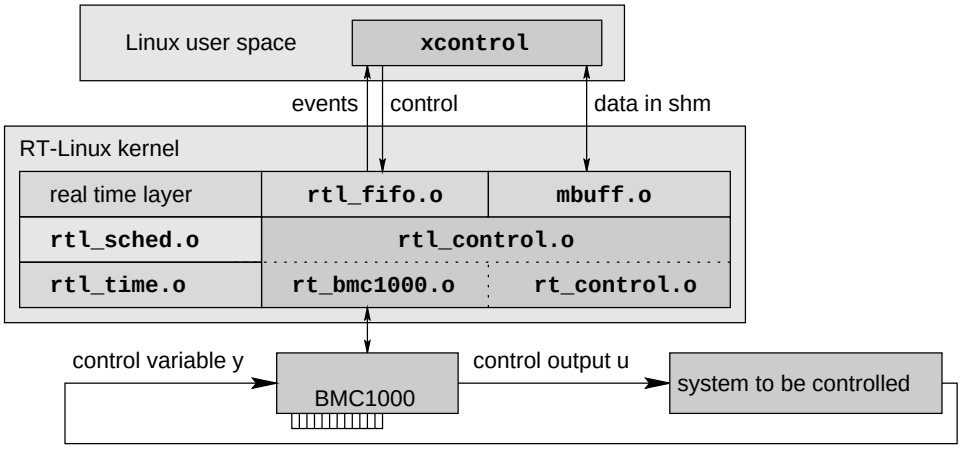

Figure 2. Principle structure of the simple SISO control system with data and signal flow.

```
#define SHM_DEV_FILE    ( "/dev/mbuff" )
#define SHM_NAME        ( "control" )
#define SHM_SIZE        ( sizeof(shm_t) )
typedef struct
  {
  unsigned int N;                /* length      */
  unsigned short int W;          /* user setp.  */
  unsigned short int w[SAMPLES]; /* setpoint    */
  unsigned short int u[SAMPLES]; /* control     */
  unsigned short int y[SAMPLES]; /* position    */
  int a[LENGTH];                 /* denominator */
  int b[LENGTH];                 /* numerator   */
  }
  shm_t;
```

Messages can be passed to and from the realtime module causing a message handler to be executed both in the realtime 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') /* X -> rtl */
#define STOP_CONTROL    ('b') /* X -> rtl */
#define TRIGGER_MEASURE ('c') /* X -> rtl */
#define MEASURE_READY   ('a') /* rtl -> X */
#define INVALID_MESSAGE ('b') /* rtl -> X */
```

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.

3.3 *Real time module*

As for the the DAQ-board module `rt_bmc1000.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` and being defined in the file `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 programed, if there is no module available on the net or by the manufacturer. Based on these 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 inline function for sending messages like:

```
void rt_control_event_msg( unsigned char message )
    {
    rtf_put( EVENT_FIFO, &message, 1 );
    }
pthread_t control;           /* control thread  */
int measure;                 /* measurment flag */
shm_t *shm;                  /* shared memory   */
unsigned int index;          /* index in shm    */
int e[LENGTH];               /* deviat. buffer  */
int u[LENGTH];               /* control buffer  */
unsigned int k;              /* current index   */
unsigned short int W;        /* setpoint        */
unsigned short int Y;        /* measure var.    */
unsigned short int U;        /* control var.    */
```

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

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

    /* get the value from the board
     */
    Y = rt_bmc1000_aget();

    /* 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_bmc1000_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) )
        {
        rt_control_event_msg( MEASURE_READY );
        measure = -1;
        }
      if (index == 1) W = shm->W;
      }
    k = (k+1) % LENGTH;
    }
  }
```

Note that this is a thread which is always alife once initialized, 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:
        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;
  }
```

is installed listening on a control FIFO which executes the functions

```
void rt_control_start(void)
  {
  /* start control by making the thread periodic
   * with the smaple time TS in ns
   */
  pthread_make_periodic_np(
    control, gethrtime(), TS );
  }

void rt_control_stop(void)
  {
  /* stop control by setting resume time to
   * infinity and period to 0
   */
  pthread_make_periodic_np(
    control, HRTIME_INFINITY, 0 );
  }

void rt_control_trigger_measure(void)
  {
  /* reset index, enable tracing
   */
  index = 0;
```

```
  measure = 0;
  }
```

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;            /* attributs */
  struct sched_param param;       /* parameter */

  /* Initialise the PCL818 DAQ board
   */
  if ( rt_bmc1000_init() )
    {
    printk( "Calling rt_pc1000_init failed\n" );
    return -ENOSYS;
    }
  rt_bmc1000_mux_select( INPUT_CHANNEL,
      RT_BMC1000_ADC_PM10 );

  /* 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 initialize 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,  &attr,
    rt_control_thread, (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 );

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

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

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

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

3.4 *User space application*

As the visible main application, `xcontrol` provides in general a main window with some subwindows (Fig. 3) 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 (Peter Mattis, 1999) in combination with a scientific plot widget (Feiguin, 1999).

3.5 *Compiling and operation*

Given that RTLinux (Yodaiken, 1999) has been installed correctly and the kernel has been compiled with the realtime 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 realtime 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 realtime 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

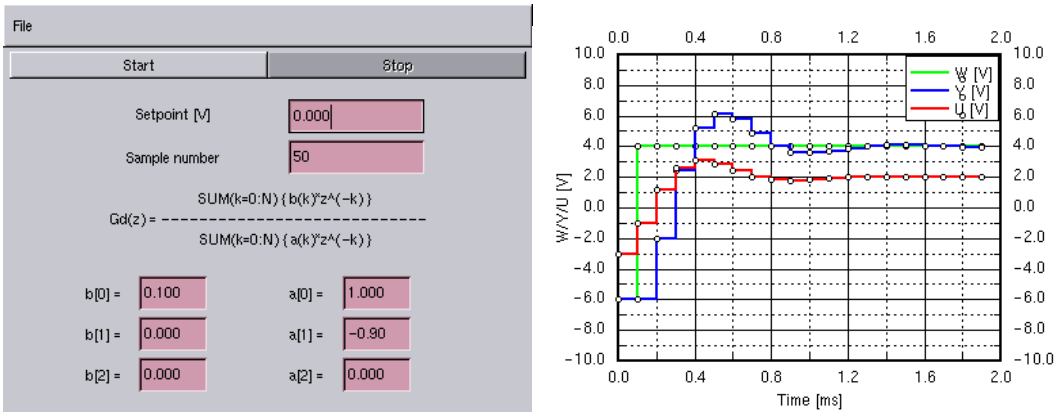

Figure 3. The graphical user interface (left) and the measurement window (right) with setpoint W, measurement variable Y and control variable U, for a proportianal plant $K = 2$ and the controller running at 10 kHz sampling rate.

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 realtime 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 realtime module through the control FIFO and the application is waiting. In contrast, the realtime 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 realtime 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 repective 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 realtime 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 realtime scheduler to schedule the thread only after a "long long" time.

4. CONCLUSION

A simple second order discrete time controller has been implemented employing RTLinux as hard realtime 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 realtime linux and to build his own applications.

5. REFERENCES

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

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

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

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

Yodaiken, Victor (1999). RTLinux, real time linux, FMSLab, Soccoro, New Mexico, USA, `http://www.rtlinux.org/`.