

RT-Linux for a torque testing stage

Peter Wurmsdobler

`peterw@cetehor.com`

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

Abstract

In this paper the use of real time linux for data acquisition and control of a micro mechanic torque testing stage is presented. More precisely, a real time linux kernel module is in charge of real time tasks taking values from a DAQ board, saving values to shared memory and vice versa. The communication with the user space application is done by means of FIFO buffers passing control and event messages. The GTK+ based graphical user interface itself presents measured data and periodic signals to be output to the micro motor.

1 Introduction

As the physical dimensions for most industrial products become smaller and smaller, micro mechanical parts, e.g. small actuators, become more and more important. In particular, there is a need for micro motors, i.e. motors in the millimeter scale. One could suspect that a millimeter sized motor is not a micro application, as the word “micro” indicates, but this field makes mechanical parts in the micro-meter scale necessary. Such motors typically have an outer diameter of up to 5 mm. Nevertheless they contain parts in the scale of $10\ \mu\text{m}$ as for the teeth height of cogwheels in a micro gearbox.

In any case, the performance of such systems has to be assessed somehow. CTM, a research center in the field of micro technology (Centre de Transfert des Microtechnique) is currently developing a testing stage in order to characterize micro mechanical components in terms of their torque and power consumption or even other values and properties. This is done in the following operation modes:

- For active components, a strong, so-called master motor drives the probe motor in order to impose a certain speed. In this case, the torque exchanged between both motors depends on the control of the probe motor which is in most cases a brushless DC motor. Therefore, the control signals have to be synchronised to the coder signals generated by the master motor. Measurement of torque and coil currents is done as function of the angular rotor position. This type of measurement is referred to as “active steady state”.
- For passive components as micro ball bearings, the master motor drives the component in order to impose a certain speed. In this case, measurement of friction torque is done as function of the angular rotor position similarly to the first case. This type of measurement is referred to as “passive steady state”.

- For transitional phases, some motor signals are captured as function of time. The rotor just accelerates according to the control signal being applied to it, with the angular position, current and torque being of interest. This type of measurement is referred to as “transient”.

Speaking in words of software, all different measurement modes demand a fast reacting computer and software system. In the case of the steady state mode, a coder with 500 impulsions per revolution generates interrupts at 66.666 kHz, i.e. in a $15\ \mu\text{s}$ time interval for a rotation speed of 8,000 rpm. On the other hand, in order to capture the transitional phase of a watch motor of the Lavet type, sampling times of $10\ \mu\text{s}$ are needed. For a DAQ board with a buffer, this should not be a problem, but in order to output a defined signal at the same time, a real time interrupt handling is indispensable.

2 Experimental setup

Given the different measurement approaches, the system has been build at CTM as for its mechanics, electronics and of course its computer hardware. Figure 1 shows in principal how the test rig looks like.



Fig. 1: *Micro torque testing stage with its mechanics (right), electronics (middle) and the computer (at this time a 486/33).*

2.1 Mechanical setup

As it can be roughly seen in Fig. 1 on the right side beneath the top plate, a small motor is held by a clamping device which is directly mounted on the torque sensor (a cylindrical shiny box). This torque sensor maps a torque of $\pm 50\ \mu\text{Nm}$ to a linear displacement by means of laser triangulation and a linear silicon spring. The sensor equipment is mounted on a xyz translation table to make the alignment to the master motor or an angular position sensor.

On the top of the rig, the master motor equipped with an angular coder drives the micro mechanic component. In the case of “active steady state” mode, the micro motor is controlled synchronously to the master motor, in case of “passive steady state” mode only measurement is done synchronously to an angular position. In case of “transient” mode, the master motor is replaced by an angular position sensor.

2.2 Electrical interface

All electronics necessary to drive the test rig is built into a rack which can be seen in the middle of Fig. 1. It contains the amplifier of the torque signal converting the laser spot displacement to $\pm 5V$, the power stage for the master motor mapping $\pm 5V$ to $\pm 8,000$ rpm as speed setpoint and the power stage for the micro motor with three channels of $\pm 5V$. Additionally, the coil currents of the micro motor are measured mapping ± 250 mA to $\pm 5V$. All signals are available by an analogue interface connector on the bottom, also including the digital coder signal as output and the enable signals for active devices as input.

2.3 Computer requirements

In order to output 4 signals, one for speed setpoint and three for the coils of the micro motor, respectively, a ACL6126 12bit output board has been used. A PCL818 has been employed to measure the torque, three currents and the motor speed at 12bit and with $10\mu s$ conversion time. However, the ISA bus access is slow and the sample time is longer, if all interrupt handling and data transfer is accounted for, approximately $40\mu s$ per channel.

Since the timer-counter on the PCL818 did not work any more, down sampling of pulses coming from the angular coder by means of hardware was not possible. In addition, the PCL818 in external trigger mode, started a ADC when externally triggered and issued an interrupt afterwards. Therefore, the parallel port has been used for creating interrupts on coder pulses for motor synchronisation purposes.

Concerning the computer, a P200 was used with 32MB RAM, a 3Com3C905 ethernet adapter, Matrox Mystique PCI, and everything usually built in a desktop computer as mouse, etc. On the OS side, NMT RTLinux 2.2.10-RTL_BETA11 has easily been compiled for this system, with `xsvga3.3.3` and `fvwm2` running as graphical user interface.

3 The micro torque testing software

The software for the torque testing stage is broken into two parts, a graphical interface as user space application, and a real time module responsible for measurement and control (Micro-Couple-Mètre = Micro torque testing stage):

- `Xmcm`, the X-Windows Micro-Couple-Mètre interface,
- `rtl_mcm.o`, the real time linux Micro-Couple-Mètre kernel module.

The logical structure from a software point of view is explained in Figure 2 with the user space application `Xmcm`, the board specific objects, `rt_pc1818.o` for the PCL818 DAQ board, `rt_acl6126.o` for the ACL6126 output board and the real time measurement functions in `rt_mcm.o` linked together to the kernel module `rtl_mcm.o`. The standard parallel port (SPP) functions responsible for the synchronisation are defined in `rt_mcm.o`.

Given that the kernel has been compiled with the real time option by NMT-RTL [1], two modules are insmoded at first, the shared memory module by Tomasz Motylewski [2] for all data being input and output, and `rtl_fifo.o` for FIFO buffers. Since there is just one “process”, which is in this case just a function, and not a task and is executed if an interrupt occurs (indicated by `irq.o`), no real time scheduler is used so far.

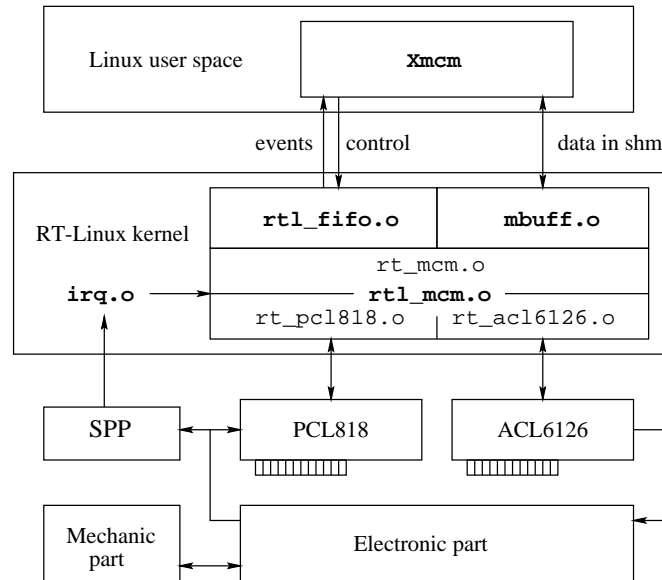


Fig. 2: Principle structure of the torque measurement system with flow of data and signals, `irq.o` symbolizes interrupt handling.

3.1 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. In fact, it is the kernel module allocating these resources in `init_module`, and the user space application mapping it. Concerning shared memory data with its variables are defined by

```
typedef struct
{
    unsigned char mcm_status;           /* status as defined in common.h */
    unsigned char mcm_case;            /* cases as defined in common.h */
    unsigned char measure_mode;        /* measure mode defined in common.h */
    unsigned char calibrate_mode;      /* calibrate mode defined in common.h */
    unsigned int speed_measurement;     /* {ENABLE,DISABLE} flag */
    unsigned int torque_measurement;    /* {ENABLE,DISABLE} flag */
    unsigned int current_measurement;   /* {ENABLE,DISABLE} flag */
    unsigned int sampling_time;         /* in microseconds */
    unsigned int down_sampling;        /* reduce interrupt rate */
    unsigned int input_length;         /* input buffer length */
    unsigned int input_index;          /* pointer to current value in input */
    unsigned int output_length;        /* output buffer length */
    unsigned int output_index;         /* pointer to current value in output */
    unsigned int output_offset;        /* offset at index = phase */
    unsigned short int speed_setpoint; /* {0,4095}, speed setpoint */
    unsigned short int amplitude;      /* {0,4095}, Probe motor voltage A */
    unsigned short int outputs[3][MAX_OUTPUTS]; /* {0,4095}, Probe motor voltages */
    unsigned short int speed[MAX_INPUTS]; /* {0,4095}, measured speed */
    unsigned short int torque[MAX_INPUTS]; /* {0,4095}, MTE Torque Tz */
    unsigned short int currents[3][MAX_INPUTS]; /* {0,4095}, Probe motor currents */
}
shm_t;
```

The vector length of the signals being measured is known in advance and the output is periodic. In particular, the size of the buffers is an integer multiple of the frame size (number of impulses per revolution) which makes up something like a ring buffer with `shm->input_index` and `shm->output_index` in shared memory pointing to the actual values.

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 by

```
#define START_MEASURE      ('a')    /* Xmcm -> rt_mcm */
#define STOP_MEASURE      ('b')    /* Xmcm -> rt_mcm */
#define START_CALIBRATE   ('c')    /* Xmcm -> rt_mcm */
#define STOP_CALIBRATE    ('d')    /* Xmcm -> rt_mcm */
#define SET_SPEED         ('e')    /* Xmcm -> rt_mcm */
#define SET_AMPLITUDE     ('f')    /* Xmcm -> rt_mcm */
#define MEASURE_READY     ('a')    /* rt_mcm -> Xmcm */
#define TRANSIENT_READY   ('b')    /* rt_mcm -> Xmcm */
#define CALIBRATE_READY   ('c')    /* rt_mcm -> Xmcm */
```

3.2 Real time module

As the DAQ-board modules (`rt_pcl818.o` and `rt_acl6126.o`) are concerned, the basic functions are the init and release, and the analogue set and get functions being called from the measurement object `rt_mcm.o`:

```
extern int rt_pcl818_init(void);
extern void rt_pcl818_release(void);
extern void rt_pcl818_aget( unsigned short int channel,
                           unsigned short int *value );
extern int rt_acl6126_init(void);
extern void rt_acl6126_release(void);
extern void rt_acl6126_aset( unsigned short int channel,
                           unsigned short int value );
```

Based upon these functions, `rt_mcm.o` is responsible for the data and control flow by using both shared memory and FIFO buffers to communicate with the graphical user interface. In `init_module` it creates the FIFOs, the message handlers for the control FIFO, initiates shared memory, registers real time interrupts (sic!) and initiates the DAQ-boards. Then it falls asleep and waits for something to happen for which there are two sources, either messages arriving at the FIFO, or interrupts.

In the first case, the application send control messages to a FIFO and the real time module will execute the appropriate functions in its message handler, e.g. to start a measure or calibration, or to set some output values before measurement. In particular, the start function will enable interrupts, program the timer on the DAQ-board by using the respective functions of the board. All this actions are not time critical.

The second way to make the module do something is by interrupts registered in `init_module`. For the case of a steady state mode measurement, the start function will set the speed set point for the master motor which will start to turn. Then the coder generates periodic pulses, the rising edge of which causes an interrupt issued by the parallel port hardware. Somehow, this will cause the rt-linux interrupt handling mechanism to launch the respective interrupt service routine:

```

unsigned int rt_mcm_spp_isr( unsigned int irq_number, struct pt_regs *p )
{
  ++daq_counter;
  if ( ( ( daq_counter % (shm->down_sampling) ) == 0 ) &&
        ( measure_lock == 0 ) && ( shm->mcm_status) == RUN ) )
  {
    measure_lock = -1;
    rtl_hard_enable_irq( RT_MCM_SPP_IRQ );
    rt_mcm_measure();
  }
  else
    rtl_hard_enable_irq( RT_MCM_SPP_IRQ );
  return 0;
}

```

Since interrupts can arrive in the worst case all $15 \mu\text{s}$, but one has to take into account approximately $40 \mu\text{s}$ per channel measurement, this interrupt service routine increases the `daq_counter` for synchronisation purposes (see Fig. 3) and just decides, whether there will be enough time to do some measurement (`shm->down_sampling` is calculated in user space). If not, it returns by enabling the interrupt before. If yes, measurement is locked and the measurement function is called. Then the actual indices in shared memory for a quasi ring buffer are calculated, and some measurement is done depending on the measurement by calling inline DAQ-functions like

```

inline void rt_mcm_get_currents( void )
{
  rt_pcl1818_aget( 3, &(shm->currents[0][shm->input_index] ) );
  rt_pcl1818_aget( 4, &(shm->currents[1][shm->input_index] ) );
  rt_pcl1818_aget( 5, &(shm->currents[2][shm->input_index] ) );
}

```

depending on `shm->mcm-case` and whether channels are enabled or not. At the same time some voltages are set according to the angular position of the rotor. Figure 3 explains the necessity of synchronisation and real time interrupt treatment.

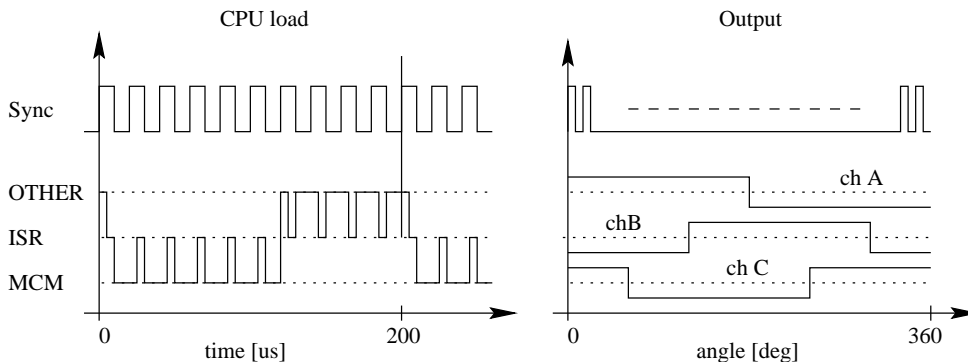


Fig. 3: *Principle of measurement and output synchronisation, in terms of time (left) and angle (right).*

The CPU is loaded either with the interrupt service routine (ISR), or the measurement task (MCM), or if there is some time left with other kernel or user tasks (OTHER). With real time linux, the treatment of the interrupts for staying synchronized can be guaranteed to high

frequencies, even at high load or if some measurement is going on. This is necessary in order to output the driving signal for the micro motor synchronously.

The procedure is similar for passive components, but even simpler. For transient phases, a similar procedure is used, but the timer (if it works, however with a different, but similar interrupt service routine) is already programmed with a reasonable sampling time. Since the measurement will take just a few milliseconds, the computer can stay blocked for this period and the measurement frequency can be as high as the ADC allows. For calibration, a similar mechanism is used, but at lower speed, so it is not very time critical.

3.3 User space application

As the visible main application, `Xmcm` provides in general a main window with some sub-windows showing the inputs and outputs of the micro torque testing stage, some control buttons in order to start and stop measurement and to start and stop calibrating, and some text fields in order to enter rotation speed, time range etc. This X-Windows interface is programmed using the GTK+ widget set library [3] in combination with a scientific plot widget [4]. All language specific texts are defined in a separate header file such that other languages can be compiled easily (necessary, because this is a European project funded by the European government).

At start-up this application maps the shared memory in its space, and opens the FIFO buffers. After this, all GTK widgets are created, like the start button or the plots for the torque to be measured. The user can now calibrate the sensors, enter some values like number of revolutions to be displayed, speed setpoint and select the channels he is interested in, torque, currents or more. If the user then presses the start button, some values are calculated and put into shared memory. Afterwards, a `START_MEASURE` is sent to the real time module and the user application is waiting.

After each revolution, the real time module sends a message to the non real time process by an event FIFO saying `MEASURE_READY`. From this point, a routine gets the last frame after the actual index in the ring buffer in order to avoid a read/write conflict. If the entire frame is fetched from shared memory, some mean values for the torque are calculated for a given speed. Then the event handler decides to display data and refreshes the screen.

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 stop interrupt generation and disable motors and the like. Then the user can print his desired torque versus speed plots.

4 Experimental results

At the time the software was developed, only a 486/33 was available which was highly motivating, especially to use real time Linux. With the common proprietary OS, measurements could be made up to 100 Hz, but the goal was to measure at least at 10 kHz. After installing real time Linux and doing programming work for the real time module, data acquisition was possible up to 25 kHz, i.e. a sampling time of 40 μ s. Of course, nearly no other Linux processes got the chance to run, but this was the limit for the 486/33.

Reducing sample time to 100 μ s, even some graphical data display was possible, even though not all data were displayed. For the display of the torque signal for one revolution at a rotational speed of 8,000 rpm, a refresh rate of the monitor of 166 Hz would be necessary. For sake of

simplicity, the user interface event handler takes only data and displays at a frequency to be set at compile time, typically at 1 s.

Since it could be shown, that real time linux is reliable and more powerful (a factor 250) as some world dominating operating system, a P200 could be bought and the system was installed once more on this computer. For this computer some measurements of the interrupt reaction have been carried out. By using a signal generator to issue the interrupts and a toggle of a value being output at the DAQ-board, the system worked reliably up to 150 kHz, i.e. a sampling time of $6.6\mu\text{s}$. For this case, a latency between the raising edge of the interrupt generating signal and the voltage on the output of the DAQ board of $4.5\mu\text{s}$ with a variance of $\pm 1\mu\text{s}$ could be measured. The entire sequence of PCI bridges, ISA delay interrupt handling and a little bit calculation makes up this delay, but it could not be measured in detail.

Since at 150 kHz there is no CPU power left to do anything, and the motor speed is in any case lower ($15\mu\text{s}$ for 8,000 rpm), some measurement can be done in parallel, but in the remaining time. So at an interrupt frequency of 50 kHz, i.e. $20\mu\text{s}$ or 6,000 rpm, a periodic signal well synchronized to the master motor could be output in order to drive the micro motor. At the same time some measurement is going on with some down sampling, i.e. the measurement function is carried out for a period of some interrupts occurring (see Fig. 3).

5 Conclusion

A torque testing stage has been developed at CTM for which some measurement software was needed, especially in order to do synchronized measurement and to output an external signal. It could be seen that real time linux is a reliable means to carry out this job. The combination of a low level measurement and control task running as kernel module with a convenient state of the art graphical user interface based on the open source GTK+ widget set solved the problem to a highly satisfying extent for our clients.

Acknowledgements

This work has been carried out within the HAFAM project (Handling and Assembly of Functionally Adapted Microcomponents), funded by the European Government in the Research Network of Training and Mobility for Young Researchers, contract NR ERB FMRX-CT97-0141.

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/>