Case Studies for RT Linux

Case Studies for RT Linux:
by Michal Sojka

Published April 2004

Table of Contents

Document presentation ..... **i**

1. RT-Linux Threads **1**

1.1. Introduction 1
1.2. Kernel Modules 1
1.3. Threads 1
1.3.1. Basic example 1
1.3.2. Thread priorities 2
1.3.3. Periodic threads 3

2. Debugging Techniques **4**

2.1. Debugging by Printing 4
2.2. Analyzing Crashes 4
2.2.1. Crashes in Non-Real-Time Code 4
2.2.2. Crashes in Real-Time Code 6
2.2.3. Finding an Error in Source Code 6

3. Controller of DC motor **8**

3.1. Introduction 8
3.2. Real-Time Controller 8
3.2.1. Motor Structure 8
3.2.1.1. Locking of the Motor Structure 9
3.2.2. Initialization 9
3.2.3. PWM Generation 9
3.2.4. Action Value Calculation 10
3.2.4.1. Fixed Point Arithmetic 10
3.2.5. Position Measuring 10
3.2.5.1. IRQ Handler 10
3.2.5.2. Measure Thread 10
3.2.6. Compilation 11
3.3. User-Space Part of Controller 12
3.4. Conclusion 12

A. Schematics of Motor Driver Board **13**

List of Tables

1. Project Co-ordinator i
2. Participant List i
A-1. Bill of Materials 14

List of Figures

1-1. The simplest Linux module (`threads0.c`) 1
1-2. Creation of RT-Linux threads (`threads1.c`) 1
1-3. Body of the thread 2
1-4. Setting of thread priorities 3
1-5. A periodic thread 3
3-1. Decalration of struct motor 8
3-2. Measurement of position 11
A-1. Motor controller scheme 13
A-2. Scheme of connectors for motor controller 13

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad PolitÃ©cnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Chapter 1. RT-Linux Threads

1.1. Introduction

This case study introduces RT-Linux basics—threads. First, the concept of Linux kernel modules is briefly explained and then follows a description of threads creation, chaniging priorities and periodic threads.

1.2. Kernel Modules

RT-Linux application is in fact a Linux kernel module. It is the same type of module, which Linux uses for drivers, filesystems and so on. The main difference between RT-Linux module and ordinary Linux module is that RT-Linux module calls functions, which are offered by RT-Linux kernel whereas ordinary module uses only Linux kernel functions.

The simplest Linux module is in Figure 1-1. It contains two functions: `init_module`, which is called when module is inserted to the kernel (usually by **insmod** or **modprobe** command), and `cleanup_module`, which is called before module is removed (usually by **rmmod**).

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
        printk("Init\n");
        return 0;
}

void cleanup_module(void)
{
        printk("Cleanup\n");
}

MODULE_LICENSE("GPL");
```

Figure 1-1. The simplest Linux module (`threads0.c`)

After you insert the module by running **insmod threads0.o**, you should see a message `Init` on your console, as well as after running **rmmod threads0**, you will see `Cleanup`. If you write RT-Linux application you will use these functions to initialize and deinitialize your application.

1.3. Threads

RT-Linux implements a POSIX API for threads manipulation. If you are familiar with POSIX threads library (`pthread`) in user-space, it should not be a problem for you to manage threads in RT-Linux. A thread is created by calling the `pthread_create()` function. The third parameter of `pthread_create()` is function which contains the code executed by the thread.

1.3.1. Basic example

Let's look at `threads1.c`, where is a simple example threads. In Figure 1-2 is code that creates three threads. Every thread does the same thing (function `thread_code`), but different parameteres are passed to each of them.

```
int init_module(void)
{
        pthread_create(&t1, NULL, &thread_code, "this is thread 1");
        rtl_printf("Thread 1 started\n");
        pthread_create(&t2, NULL, &thread_code, "this is thread 2");
        rtl_printf("Thread 2 started\n");
        pthread_create(&t3, NULL, &thread_code, "this is thread 3");
        rtl_printf("Thread 3 started\n");
        return 0;
}
```

Figure 1-2. Creation of RT-Linux threads (`threads1.c`)

```
void do_some_work(void)
{
        rtl_delay(1000000000);        /* 1 second */
}

void *thread_code(void *arg)
{
        int i;

        /* If this line isn't commented, the behaviour is diferent. */
/*         usleep(10000);                */              ❶

        for (i = 0; i < 3; i++) {
                rtl_printf("Message: %s\n", (char *) arg);
                do_some_work();
        }

        return (void *)0;
}
```

Figure 1-3. Body of the thread

The code of each thread is in Figure 1-3. If you run this RT-Linux application, you should see the following messages on your console:

```
Message: This is thread 1
Message: This is thread 1
Message: This is thread 1
Thread 1 has stared
Message: This is thread 2
Message: This is thread 2
Message: This is thread 2
Thread 2 has started
Message: This is thread 3
Message: This is thread 3
Message: This is thread 3
Thread 3 has started.
```

You may wonder, why the consecutive thread is started after the previous thread has finished. The explanation is very simple. Every RT-Linux thread has higher priority than Linux kernel (and user-space application too). The `init_module` function is executed in Linux context and because our real-time threads don't sleep, Linux have next chance to run only after the real-time thread finishes.

If you uncomment line marked by ❶, the behavior changes. Now, all threads wait at beginning of its execution and Linux has a chance to run remaining threads. After inserting the modified module, there should appear on your console something like that:

```
Thread 1 has started
Thread 2 has started
Thread 3 has started
Message: This is thread 1
Message: This is thread 3
Message: This is thread 3
Message: This is thread 3
Message: This is thread 2
Message: This is thread 2
Message: This is thread 2
Message: This is thread 1
Message: This is thread 1
```

1.3.2. Thread priorities

Often it is useful to set thread priorities. Threads with higher priorities can preempt threads with lower priorities. For example, we can have a thread controlling a stepper motor. In order to move the motor fluently, it is necessary to start this thread in strictly regular intervals. Assigning a high priority to this thread, we can guarantee this.

In the example `threads2.c`, the same thing is done as in the `thread1.c`, but we set different priorities. According to these priorities, the order of lines on program output differs from previous example. Setting of thread priority is done by code shown in Figure 1-4.

```
int init_module(void)
{
        pthread_attr_t attr;
        struct sched_param param;

        pthread_attr_init(&attr);
        param.sched_priority = 1;
        pthread_attr_setschedparam(&attr, &param);
        pthread_create(&t1, &attr, &thread_code, "this is thread 1");
        rtl_printf("Thread 1 started\n");
...
```

Figure 1-4. Setting of thread priorities

1.3.3. Periodic threads

Most of control applications do its task by periodically running the same piece of code. This is called a *periodic thread* and RT-Linux has support for this type of thread. After start, the thread should call `pthread_make_periodic_np` to setup its period. Then it enters a loop in which a `pthread_wait_np` function is called. This function assures that the thread waits to the start of next period. In Figure 1-5 there is a periodic thread from `threads3.c` example.

```
void *high_prio_thread(void *arg)
{
        pthread_make_periodic_np(pthread_self(), gethrtime(), 300*MS);

        while (!terminate) {
                rtl_printf("#");
                do_some_work();
                pthread_wait_np();
        }

        return (void *)0;
}
```

Figure 1-5. A periodic thread

This example contains two threads. One thread with high and one thread with low priority. The high priority thread has shorter period (300 ms) and have also less to do. The low priority thread has period of 2 seconds and has five times more to do. When an example is run, one can see (by watching the printing of # character), that the high priority thread runs very regularly regardless of whether the low priority thread does something or not.

Chapter 2. Debugging Techniques

This chapter provides basic information about debugging of RT-Linux modules. Especially it covers analysis of real-time program crashes and learn how to acquire as much information as possible from kernel *oops* messages.

Debugging of RT-Linux code is not so simple as debugging of a user-space application. The reason of this is absence of interactive debugging facility in Linux kernel. There are some patches to Linux kernel that provide some debugging possibilities and RT-Linux has also its own debugging facility[1], but usage of these interactive tools is out of scope of this document and interested reader should consult the documentation of an appropriate tool.

2.1. Debugging by Printing

There are several reasons why we need to debug our application. First, we may need to find out why our application doesn't behave the way we want. This is often trivial error, which can be discovered by looking at sources or by technique called *debugging by printing*. We use a `rtl_printf` function to print some relevant messages, which help us to find an error. It is worth noting that the behavior of `rtl_printf` depends on configuration of RT-Linux. If option `CONFIG_RTL_SLOW_CONSOLE` is set, all messages printed with `rtl_printf` are passed to `printk`. This has an advantage of appearing the messages in kernel log, which can be seen by running a **dmesg** command. On the other hand it has a drawback too. When Linux hasn't chance to run (RT-Linux threads always have higher priority), no messages are written to a console and to the kernel log as well.

The second case where we need debugging is a solving of application crashes. When an application crashes it is important to collect as much information concerning the crash as possible. After we have the right information, we can find where the crash was and try to correct what caused the error. Later we can use the debugging by printing technique to see values of variables and so on, to find out what exactly caused the crash.

2.2. Analyzing Crashes

In most cases, crash of an application is caused by a fault when accessing an invalid virtual memory address. In such case an exception is generated by the CPU and Linux handles it by printing a so called *oops message*. If we are lucky enough, the fault was not at real-time level but rather at Linux level, such as in module initialization or cleanup function. In this case the kernel kills the process, in whose context was the kernel running, in the time of the fault. This is often an **insmod** or **rmmod** command. It is important, that the kernel go on running. If we want to find out more information about the crash, it is always better to have a module compiled with debugging information. This can be accomplished by invoking **gcc** with `-g` prarameter.

2.2.1. Crashes in Non-Real-Time Code

Let's look at example `buggy0.c`. There is a function `buggy`, which tries to write something to the `NULL` address.

```
void buggy(void)
{
    int *p = NULL;
    *p = 123;
}
```

For educational purposes, this function is called from another function called `my_function`. When we try to insert `buggy0` module by running **insmod buggy0**, an oops message appears:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
c8852075
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[<c8852075>]    Not tainted
EFLAGS: 00010282
eax: 0000007b   ebx: c8852000   ecx: c02dd010   edx: c118a244
esi: 00000000   edi: 00000000   ebp: ffffffea   esp: c69c9f18
ds: 0018   es: 0018   ss: 0018
Process insmod (pid: 145, stackpage=c69c9000)
Stack: c8852085 c0118ebb c8852060 080add18 00000100 00000000 080add4c 00000094
       00000060 00000060 00000006 c691b1e0 c691a000 c691d000 00000060 c8841000
       c8852060 00000160 00000000 00000000 00000000 00000000 00000000 00000000
Call Trace: [<c8852085>] [<c0118ebb>] [<c8852060>] [<c8852060>] [<c0107617>]

Code: a3 00 00 00 00 c3 90 8d 74 26 00 e8 db ff ff ff 31 c0 c3 90
```

We see that we are accessing memory at address 0, which is the only meaningful information for this time. To obtain more information, we need to run this oops message through **ksymoops** utility. The easiest way of doing this is running command **dmesg|ksymoops** (using a pipe to pass output of dmesg to the ksymoops). Possible output of this command is shown here:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
c8852075
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[<c8852075>]    Not tainted
Using defaults from ksymoops -t elf32-i386 -a i386
EFLAGS: 00010282
eax: 0000007b   ebx: c8852000   ecx: c02dd010   edx: c118a244
esi: 00000000   edi: 00000000   ebp: ffffffea   esp: c69c9f18
ds: 0018   es: 0018   ss: 0018
Process insmod (pid: 145, stackpage=c69c9000)
Stack: c8852085 c0118ebb c8852060 080add18 00000100 00000000 080add4c 00000094
       00000060 00000060 00000006 c691b1e0 c691a000 c691d000 00000060 c8841000
       c8852060 00000160 00000000 00000000 00000000 00000000 00000000 00000000
Call Trace: [<c8852085>] [<c0118ebb>] [<c8852060>] [<c8852060>] [<c0107617>]
Code: a3 00 00 00 00 c3 90 8d 74 26 00 e8 db ff ff ff 31 c0 c3 90


>>EIP; c8852075 <[buggy0]buggy+5/10>   <=====              ❶

>>ebx; c8852000 <[rtl_sched]rtl_timer_list_lock+3278/32d8>
>>ecx; c02dd010 <contig_page_data+130/320>
>>edx; c118a244 <_end+e1e678/84ba494>
>>esp; c69c9f18 <_end+665e34c/84ba494>

Trace; c8852085 <[buggy0]init_module+5/10>                ❷
Trace; c0118ebb <sys_init_module+4bb/630>
Trace; c8852060 <[buggy0]my_function+0/0>
Trace; c8852060 <[buggy0]my_function+0/0>
Trace; c0107617 <system_call+47/50>

Code;  c8852075 <[buggy0]buggy+5/10>
00000000 <_EIP>:
Code;  c8852075 <[buggy0]buggy+5/10>   <=====
   0:   a3 00 00 00 00            mov    %eax,0x0   <===== ❸
Code;  c885207a <[buggy0]buggy+a/10>
   5:   c3                        ret
Code;  c885207b <[buggy0]buggy+b/10>
   6:   90                        nop
Code;  c885207c <[buggy0]buggy+c/10>
   7:   8d 74 26 00               lea    0x0(%esi,1),%esi
Code;  c8852080 <[buggy0]init_module+0/10>
   b:   e8 db ff ff ff            call   ffffffeb <_EIP+0xffffffeb>
Code;  c8852085 <[buggy0]init_module+5/10>
  10:   31 c0                     xor    %eax,%eax
Code;  c8852087 <[buggy0]init_module+7/10>
  12:   c3                        ret
Code;  c8852088 <[buggy0]init_module+8/10>
  13:   90                        nop
```

Let's look at some interesting lines:

❶ Here we can see a value of an instruction pointer (EIP) and in which function this vaue is. We see it is in function called buggy, which is 0x10 (16) bytes long and EIP points to the 5th byte of this function.

❷ Starting with this line, here is a stack trace. These lines show **possible** order of called functions. In our example the order shown here differs a little bit from real situation on the stack, but it can also help us. With a little imagination we see that the buggy function was called from my_fuction and this was called from init_module.

❸ On this line is shown exact instruction that caused the fault. Here the value of eax register was written to address 0x0.

Now we have all needed peaces of information and can continue with Section 2.2.3, *Finding an Error in Source Code*.

2.2.2. Crashes in Real-Time Code

When a fault occurs in real-time part, which is not associated with any process, oops message is printed and kernel doesn't go on running. This is a problem, because we can't gather information so easy as in previous section. Let's study example buggy1.c. Here is the function buggy too, but instead of being called from init_module, it is called from real-time thread thread1.

When inserting the module to the kernel, we should add *-m* to the **insmod** command. This causes the insmod to print symbols and addresses of inserted module. On my computer it gives:

```
# insmod -m buggy1.o
...
Symbols:
00000000 a buggy1.mod.c
00000000 a buggy1.c
c8852000 d __this_module
c8852000 D __insmod_buggy1_O/lib/modules/2.4.18-ocera-0.5/misc/buggy1.o_M40698F38_V132114
c8852060 T __insmod_buggy1_S.text_L156
c8852060 T buggy
c8852060 t .text
c8852070 T my_function
c8852080 T thread1
c88520a0 t init_module
c88520e0 t cleanup_module
c88521f0 d .bss
c88521f0 D t1
c88521f0 d .data
```

After the buggy() function invokes a fault, oops message similar to that in previous example is printed. In this oops message we should notice a value of the EIP register. On my computer I see:

```
 printing eip:
c8852065
```

When we compare this address with the list of symbol values, we find out this is five bytes after buggy symbol, which marks beginning of same named function.

2.2.3. Finding an Error in Source Code

Now we have exact information about an instruction that causes the fault. This is probably not enough to remove the bug, because we don't know what line in C source code this instruction belongs to. This can be determined by **objdump** utility. If we have our module compiled with debugging information, we can run

```
# objdump -S buggy1.o | less
```

This command dissassembles the module and prints assembler code intermixed with C source code. Here is a listing of buggy function:

```
Disassembly of section .text:
```

```
00000000 <buggy>:
void buggy(void)
{
        int *p = NULL;

        *p = 123;               /* attempt to write to bad memory
   0:   b8 7b 00 00 00          mov    $0x7b,%eax
   5:   a3 00 00 00 00          mov    %eax,0x0
                                * position */
}
   a:   c3                      ret
   b:   90                      nop
   c:   8d 74 26 00             lea    0x0(%esi,1),%esi
```

We see that on 5th byte after beginning of `buggy` function is instruction `mov %eax,0x0` and this instruction belongs to C source line `*p = 123;`.

Notes

1. The RT-Linux debugging facility is implemented in module `rtl_debug.o`.

Chapter 3. Controller of DC motor

3.1. Introduction

This article is documentation for a case study for RT-Linux. The goal of this case study is to control velocity of DC motor with IRC (Incremental Radial Counter) sensor. The motor is connected to a PC through a simple board consisting of a motor driver and basic logic circuits. This board is connected to a PC printer port. Schematics for the board can be found in Appendix A, *Schematics of Motor Driver Board*.

The case study consists of two programs. One is a RT-Linux kernel module that forms a real-time part of a controller and the second is a user-space program, which displays current status of the controller to the user and let him to change desired value of velocity. This program communicates with real-time part via real-time FIFO. The user-space program is very simple and if someone wants it could be for example nice GUI application.

In the following sections we are going to describe parts of the program in more detailed manner. It can be useful if you can look at the source codes when reading this. There are lots of comments in the source code and if you don't understand some part of this documentation, the comments may help you.

3.2. Real-Time Controller

The real-time controller is the main part of this case study. It is a RT-Linux module and in this section we describe particular pieces of this module.

3.2.1. Motor Structure

The heart of real-time part is a motor structure (struct motor, Figure 3-1). This structure is meant to represent one real motor. In our case study we have only one motor, but if someone wants us to control two or more motors it would be very simple to implement it. You can consider this as some sort of object oriented programming in plain C (not C++) language. We have a structure representing some *object* and also functions that works with this object. These functions are in object oriented terminology called *methods*. The only difference[1] between this approach and object oriented languages is that we have to explicitly pass the motor structure as a parameter when calling methods.

```
struct motor {
      int irq;                /* interrupt request number */
      int base;               /* base address of parallel port registers */

      /* This spinlock protects the following variables, which are
      * used in irq handling routine. */
      //pthread_mutex_t spinlock;
      rtl_spinlock_t spinlock;

      int delta_pos;          /* num. of irqs during last sample
      * period, signum determines
      * direction */
      int last_aper;
      int dir;                /* direction of motor rotation */
      hrtime_t last_irq_time; /* time of last irq */
      hrtime_t last_irq_interval; /* time between two last irqs */

      /* This lock protects action variable. We use special lock
      * only for this variable, because it is used quite often by
      * PWM thread and long locking of this variable would break
      * the PWM accurancy. */
      pthread_mutex_t action_lock;
      int action;             /* action value that controls PWM
      * (0-PWM_RESOLUTION) */
      /* This lock protects everithing below against other threads. */
      pthread_mutex_t lock;
```

```
        /* Some of the following variables store values in fixed point
        * aritmetic (FPA). Lower 16 bits is treated as decimal
        * part. */
        int reference;          /* desired value of velocity (FPA) */

        int velocity;           /* measured velocity (FPA) */
        int position;           /* measured position (FPA) */

        int last_pos_corr;      /* last correction of position based
        * on time calculations (FPA) */

        int sum_dev;            /* sum of regulator deviations (FPA) */
        int last_dev;           /* last regulator deviation (FPA) */

        pthread_t thr_pwm, thr_measure, thr_regul;
        int fifo_in, fifo_out;
}
```

Figure 3-1. Decalration of struct motor

3.2.1.1. Locking of the Motor Structure

In order to prevent multiple threads from manipulating motor structure concurrently, some locking is needed. Basic idea behind locking is that any thread accessing a resource (in our case the resource is a motor structure) must lock it. Whenever a thread has locked the resource, other threads have to wait for the first thread to unlock the resource.

There are thee locks in motor structure that are used to lock various parts of the motor structure:

spinlock
: is a lock of type rtl_spinlock_t which is used to protect variables against modification by an IRQ handler or, in SMP (Symmetric multi-processing) machine, by other processors. When accessing the motor structure in a regular thread (not in an IRQ handler) we lock the spinlock by calling `rtl_spinlock_irqsave()`. This forbids an IRQ reception on a CPU and, if compiled on SMP, locks the spinlock. In the IRQ handler we use `rtl_spinlock_lock()` to lock variables. This only locks the spinlock on SMP and on UP (uni-procesor) it does nothing, because other IRQs are already forbidden when one IRQ is handled. There isn't anyone else who could modify our data.

action_lock
: is used to lock an `action` variable. The `action` variable isn't protected by the third lock (see bellow) because longer locking of that lock would prevent the PWM (Pulse Width Modulation) thread (see Section 3.2.3, *PWM Generation*) doing its work.

lock
: this lock protects all variables not protected by any of above locks.

3.2.2. Initialization

Initialization is done in function `motor_init`. We call the function (method) `start_motor`, which initializes the motor structure, mutexes, FIFOs and so on. It also starts three threads that implement the controller. These threads are:

- PWM thread, for generating PWM signal,
- measure thread, which measures current position and speed at regular intervals and
- regul thread which calculates the action value.

3.2.3. PWM Generation

The PWM signal that drives the motor is generated by `pwm` function. There is an endless loop in this function. The loop body is executed once per time defined by `PWM_PERIOD` symbol. Default value is 1 ms.

In the beginning of the loop, one of the two output pins of printer port connected to motor driver is set. Which bit is set depends on sign of *action value*. Then the thread sleeps for some time that depends on an *action value* too. After waking up, the output pin is reset to zero. It remains reset until beginning of next period.

3.2.4. Action Value Calculation

Action value (the value used by PWM thread) is calculated by a simple PSD regulator. Regulator is implemented in function `regul`. This function blocks at beginning and when new value of velocity is measured, it is woken up. The real computation of action value is done in `calc_action` function. The calculations are very simple, but they use a trick that is described in next section.

3.2.4.1. Fixed Point Arithmetic

In order to represent decimal numbers in our control algorithm we use a trick called *fixed-point arithmetic*. This is very simple. We use an integer variable to represent decimal number. Sixteen least significant bits are used to represent decimal part of a number and 16 most significant bits represent integer part.

We can add or subtract these numbers like ordinary integers. When we want to multiply them, we can choose from the following methods:

- Multiply the values using 64 bit multiplication and shift the result to right by 32 bits.
- Before multiplication shift both operands to right by 8 bits.
- Shift one operand to the right by 16 bits.

3.2.5. Position Measuring

The measuring of position is the most complicated task in this case study. It can be solved in more simple way than is presented here, but our method gives better results in regulation and also allow us to demonstrate more RT-Linux features.

First, we will explain the simpler method and then extend it to the more complicated one. Before explanation of measurement procedure we should describe what the IRQ handler does.

3.2.5.1. IRQ Handler

Whenever signal from IRC sensor changes, an interrupt is generated. The interrupt handler must read two-bit value of IRC sensor and determine the direction in which the motor is rotating. According the direction a temporary variable `delta_pos` representing current position of a motor is incremented or decremented. Also the actual time is remembered in variable `last_irq_time` and a time since last IRQ is computed and saved in variable `last_irq_interval`. See Figure 3-2 for graphic representation of this.

3.2.5.2. Measure Thread

This periodic thread is implemented in function `measure`. Before describing this thread, let's look at Figure 3-2. This is a graphical explanation of what is done when position is measured. On a horizontal axis there is time and on vertical axis is motor position. The horizontal dotted lines represent position where IRC changes the output value. The thick black curve is real position of motor. The figure thus shows that, at beginning, the

motor rotates with constant velocity to one direction, than it quickly rotates to other direction and finally it rotates to the same direction as at beginning.

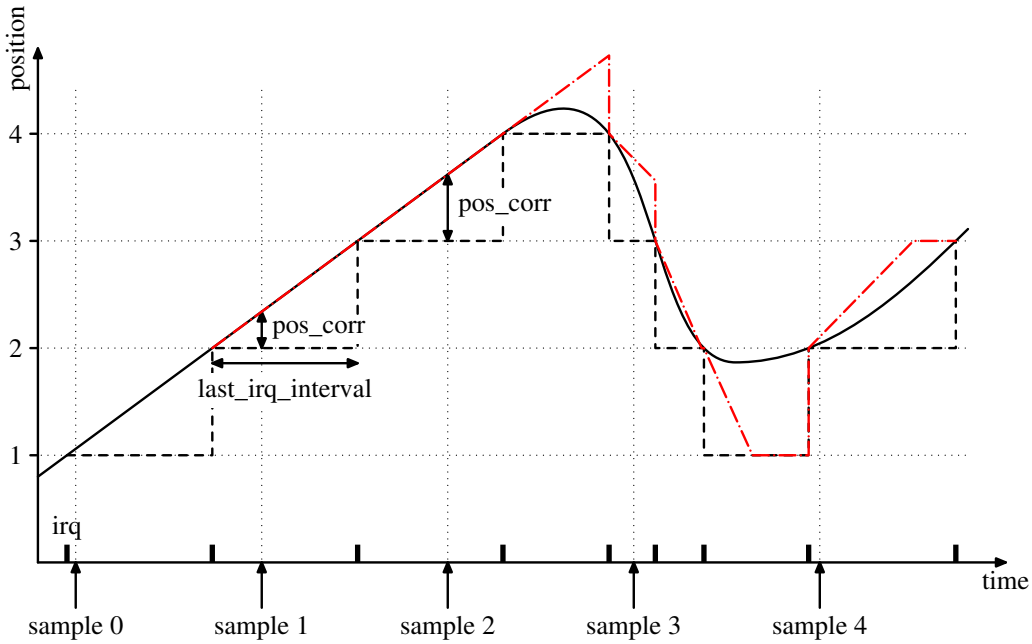

Figure 3-2. Measurement of position

The simpler version of an algorithm should do only these steps:

- `delta_pos` is saved to temporal variable and zeroed. When this is being done, interrupts should be disabled.
- The saved value of `delta_pos` (position change since last sample period) is added to `position`.
- Actual motor `velocity` is computed depending on value of `delta_pos`.

This method of measuring position gives us the results that are depicted in Figure 3-2 by the black dashed line. The inaccurate measuring of position in this simple version isn't big issue. Measuring of a velocity is worse. When motor rotates with constant velocity, the measured velocity isn't constant and oscillates between two adjacent values. Because we use measured velocity to close feedback loop in PSD controller, we need better measuring of velocity.

The better method tries to estimate a real position depending on time elapsed since last IRQ. In each sampling period the correction of position is calculated. The calculation is based on linear extrapolation of position between last two IRQs. Variables used in this calculation are: `last_irq_interval`, `last_irq_time` and current time. What the controller thinks about current position measured in this manner is depicted in Figure 3-2 by red dash-and-dot line.

Whenever the correction is bigger than one, we trim it to one. This can be seen for example before sample 4.

3.2.6. Compilation

Compilation of this case study is a little bit harder than of other RT-Linux projects. The reason of this is that we need 64-bit division. This functionality is contained in `libgcc` library. When compiling user-space application, this library is automatically linked with the application. This is not true with kernel modules and we have explicitly link the kernel module with `libgcc`. We can determine the exact name of the library by **gcc -print-libgcc-file-name** command. Then we use **ld** command to link the module with the library.

```
ld -r -o motor_ok.o motor.o $(gcc -print-libgcc-file-name)
```

3.3. User-Space Part of Controller

In this case study we want the user to have an ability to modify desired value of velocity. Also it is useful when a user can see actual velocity and position of motor.

Because this functionality isn't required to be hard real-time, is better not to include it in real-time part and do it as stand-alone user-space application. This application communicates with real-time part via RT FIFOs. Its source code is in `ctrl.c` file and is very simple.

At the beginning two FIFOs are opened. One is for reading status from real-time part and one for writing data to real-time part. Then there is an endless loop, so the user can quit the program only by pressing **Ctrl-C**[2]. At the beginning of the loop we wait (with timeout) for either data from real-time part or input from keyboard via STDIN. This waiting is performed in `select()` system call. After returning from this call we test if some file descriptor has data for us.

If there are data from STDIN, we print an issue to user to input desired rotation value. Then this value is sent to real-time part via FIFO. If there are data from input FIFO, we read them and store them to the structure of the same type as is used by real-time part. Then we print stored values.

3.4. Conclusion

In this case study we have developed functional controller of DC motor. There is space for extending its functionality, but it wouldn't show us more RT-Linux features. One could for example write support for changing parameters of PSD regulator by user or there could be a better (nicer) user-space program.

An important notice: When someone would solve the same task as in this case study in an industrial environment, he will probably use some hardware for PWM signal generation and/or position measuring. Almost every modern microprocessor dedicated to control applications can do these very easily.

Notes

1. In a fact, there are more differences. We don't have for example a polymorphism feature, but when we manually create virtual methods table, we can use this feature in C language too. This is common practice in Linux kernel and some other bigger C projects.
2. This is quite common practice in Unix programs

Appendix A. Schematics of Motor Driver Board

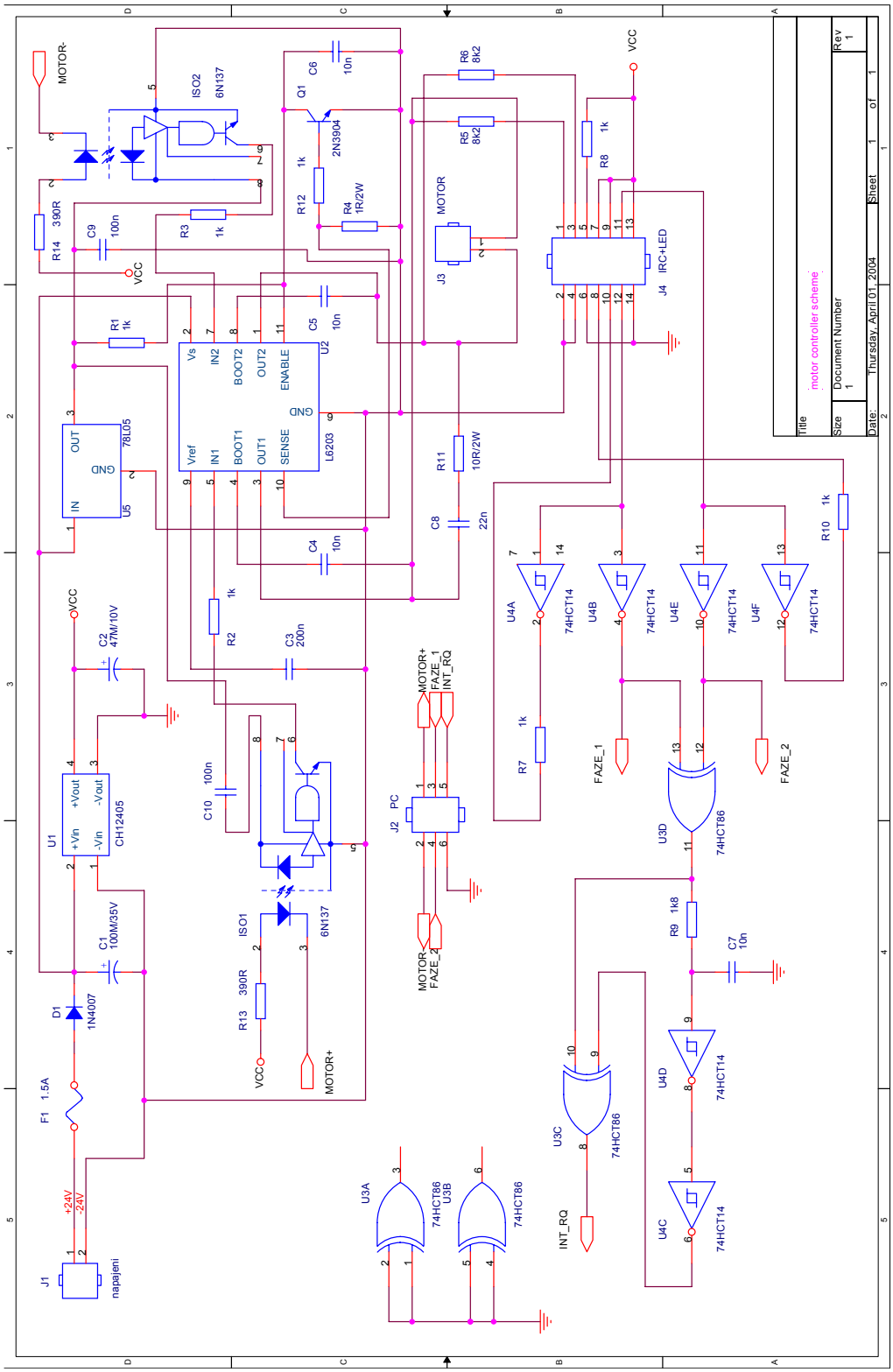

Figure A-1. Motor controller scheme

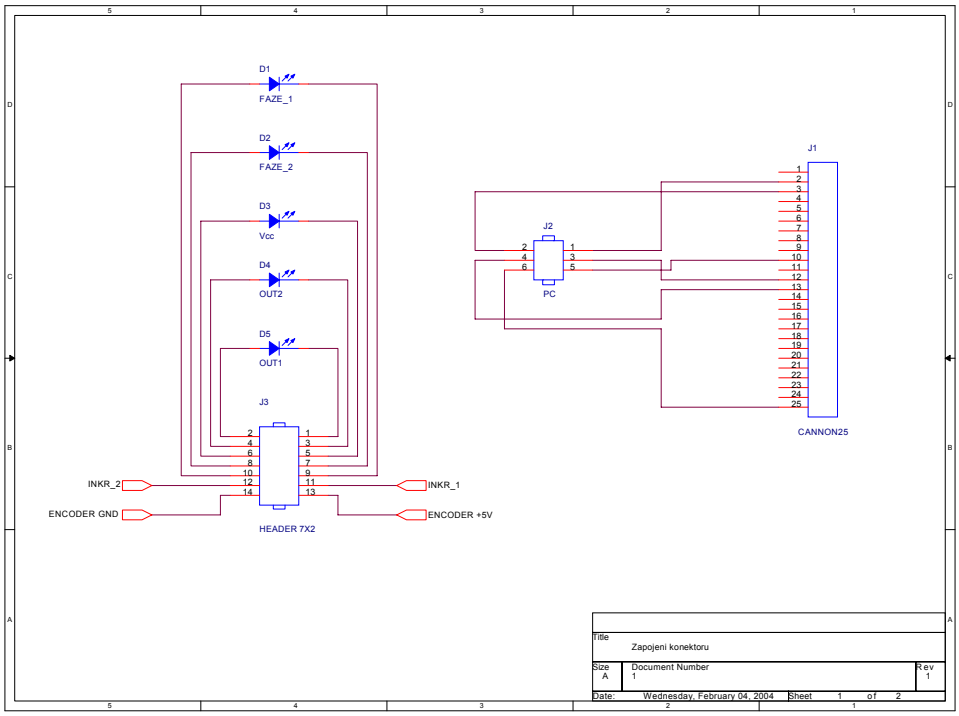

Figure A-2. Scheme of connectors for motor controller

Table A-1. Bill of Materials

Item	Quantity	Reference	Part
1	1	C1	100M/35V
2	1	C2	47M/10V
3	1	C3	200n
4	4	C4,C5,C6,C7	10n
5	1	C8	22n
6	2	C9,C10	100n
7	1	D1	1N4007
8	1	F1	1.5A
9	2	ISO2,ISO1	6N137
10	1	J1	power supply
11	1	J2	PC
12	1	J3	MOTOR
13	1	J4	IRC+LED
14	1	Q1	2N3904
15	7	R1,R2,R3,R7,R8,R10,R12	1k
16	1	R4	1R/2W
17	2	R5, R6	8k2
18	1	R9	1k8
19	1	R11	10R/2W
20	2	R14, R13	390R
21	1	U1	CH12405
22	1	U2	L6203
23	1	U3	74HCT86
24	1	U4	74HCT14
25	1	U5	78L05