Case Studies for RT Linux

Case Studies for RT Linux:
by Michal Sojka

Second Edition
Published February 2005

Table of Contents

Preface ..... **i**
Document presentation ..... **ii**
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 3
1.3.3. Periodic threads 3
1.4. Assignment 4
2. Debugging Techniques **5**
2.1. Introduction 5
2.2. Debugging by Printing 5
2.3. Analyzing Crashes 5
2.3.1. Crashes in Non-Real-Time Code 5
2.3.2. Crashes in Real-Time Code 7
2.3.3. Finding an Error in Source Code 7
2.4. Assignment 8
3. Controller of DC motor **9**
3.1. Introduction 9
3.2. Real-Time Controller 9
3.2.1. The Motor Data Structure 9
3.2.1.1. Locking of the Motor Structure 10
3.2.2. Initialization 10
3.2.3. PWM Generation 10
3.2.4. Action Value Calculation 11
3.2.4.1. Fixed Point Arithmetic 12
3.2.5. Position Measuring 12
3.2.5.1. IRQ Handler 12
3.2.5.2. Measure Thread 13
3.2.6. Compilation 14
3.3. User-Space Part of Controller 14
3.4. Conclusion 15
3.5. Assignment 15
4. CANping – a simple LinCAN testing application **17**
4.1. Introduction 17
4.2. CANping manual 17
canping 17
4.3. Implementation details 18
4.3.1. Usage of the LinCAN driver 19
4.3.1.1. Opening the Driver 19
4.3.1.2. Setting Filters 19
4.3.1.3. Reading and Writing 19
4.3.1.4. Closing 20
4.3.2. Other parts of the application 20
4.3.2.1. Lists 20
4.3.2.2. Waiting for the thread completion 21
4.3.2.3. Signal handling 21
4.4. Assignment 22
A. Schematics of Motor Driver Board **23**

List of Tables

1. Project Co-ordinator ii
2. Participant List ii
A-1. Bill of Materials 24

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 (high priority) 3
1-6. The output of threads3 example. 4
3-1. Declaration of struct motor 9
3-2. Code of the `pwm` function 11
3-3. Code of the `regul` function 11
3-4. Code of `irq_handler` function 12
3-5. Measurement of position 13
3-6. User-space part of the controller 14
A-1. Motor controller scheme 23
A-2. Scheme of connectors for motor controller 23

Preface

This document contains several case studies for RT-Linux and some OCERA components. It covers RT-Linux basics as well as quite advanced topics, but it is not exhaustive. There are many topics not covered by this document, so the reader has to study other documents in order to learn how to use RT-Linux and/or OCERA components.

The following chapters are organised as follows. First, there is an introduction presenting what is the case study about and what the reader will learn. In the following sections a problem is analysed. For each case study, there is one or more programs solving the problem and these programs are explained in detail. The source code of these programs is available so the reader can modify it and look what their change caused. At the end, there are assignments for students. These can be used as exercises in a school.

If these case studies are used in school for teaching RT-Linux and OCERA, the teacher should explain the problem in a similar way how it is written here and then let the students to solve individual tasks e.g. those mentioned in the assignment sections.

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 Politécnica 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 then follows a description of how to create threads and change their priorities and at the end a section about periodic threads is included.

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, file systems and so on. The main difference between RT-Linux module and an 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 source code of a simple Linux module is in Figure 1-1. It contains two functions: `init_module`, which is called when the 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. After running **rmmod threads0**, you will see a `Cleanup` message. If you write RT-Linux application you use these functions to initialize and deinitialize your application.

1.3. Threads

RT-Linux implements a POSIX API for a threads manipulation. If you are familiar with a POSIX threads library (`pthread`) used by user-space application, it should not be problem for you to work with threads in RT-Linux. A thread is created by calling the `pthread_create()` function. The third parameter of `pthread_create()` is a 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. In Figure 1-2 is code that creates three threads. Every thread does the same thing (function `thread_code()`), but different parameters 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.

The behaviour of `rtl_printf()` depends on whether you have RT-Linux compiled with the option "rtl_printf uses printk" set or not. If this option is set, messages are printed to the console only when Linux runs, which may by sometimes long time from the instant when the message was produced. For purposes of these case studies it is better to not set this option. In this case you can see in "real-time" what the CPU does.

If you uncomment line marked by ❶, the behaviour changes. Now, all threads wait at beginning of its execution and Linux has a chance to create 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 necessary 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. This can be guaranteed by assigning a high priority to this thread.

The example `threads2.c` does the same as `thread1.c`, but we set different thread priorities. Due to the priority changes, the order of lines at the program output differs from the 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

The output the program is as follows.

```
Thread 1 started
Thread 2 started
Thread 3 started
Message: this is thread 1
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
Message: this is thread 3
Message: this is thread 3
Message: this is thread 3
```

The thread 2 has the highest priority and the thread 3 has the lowest priority. The first message is printed by the middle priority thread 1 because it is started short time before the thread 2.

1.3.3. Periodic threads

Most of control applications execute periodic tasks. They same code is run repeatedly in regular intervals. This is called a *periodic thread* and RT-Linux has support for this type of threads. After spawning, the thread should call `pthread_make_periodic_np()` to setup its period. Then it enters a loop in which the `pthread_wait_np()` function is called. This function assures that the thread waits to the start of the next period. In Figure 1-5, there is code of periodic threads 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;
}
```

```
void *low_prio_thread(void *arg)
{
        int i;
        pthread_make_periodic_np(pthread_self(), gethrtime(), 2*1000*MS);
        while (!terminate) {
                for (i = 0; i < 5; i++) {
                        rtl_printf(".");
                        do_some_work();
                }
                pthread_wait_np();
        }
        return (void *)0;
}
```

Figure 1-5. A periodic thread (high priority)

This example contains two threads. One thread is assigned high priority and the second low priority. The period of the high priority thread is 300 ms and this thread has 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[1].

```
#..#..#.#####..#..#.####.#..#..#####..#..#.#####..#..#.####.#..#..###
##..#..#.#####..#..#.####.#..#..#####..#..#.#####..#..#.####.#..#..##
###..#..#.#####..#..#.####.#..#..#####..#..#.#####..#..#.####.#..#..#
####..#..#.####
```

Figure 1-6. The output of threads3 example.

1.4. Assignment

Create an application that will regularly switch on and off LEDs connected to parallel port. Use three LEDs and every LED will blink with a different period.

Notes

1. The '#' characters appear regularly only if you have RT-Linux compiled without the "rtl_printf uses printk" option.

Chapter 2. Debugging Techniques

2.1. Introduction

This chapter provides basic information on debugging RT-Linux modules. More precisely, it covers analysis of real-time program crashes and teaches the reader how to acquire as much information as possible from kernel *oops* messages.

The debugging of RT-Linux code is not as simple as the debugging of a user-space application. The reason of this is the absence of an interactive debugging facility in the 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 the appropriate tool.

2.2. 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 a trivial error, which can be discovered by looking at sources or by the 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 behaviour 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 **dmesg** command. On the other hand it has a drawback too. When Linux hasn't chance to run (RT-Linux threads have always higher priority than Linux), no messages are written to a console and to the kernel log as well.

The second case where we need debugging is the 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 fix the code, which caused the error. Later we can use the debugging by printing technique to see values of variables or to print other useful information, to find out what have exactly caused the crash.

2.3. Analyzing Crashes

In most cases, the crash of an application is caused by a fault when accessing an invalid virtual memory address. In such a 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 the module initialization or cleanup function. In this case the kernel kills the process, in whose context the kernel was 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` parameter.

2.3.1. Crashes in Non-Real-Time Code

Let's look at the `buggy0.c` example. There is a function `buggy`, which tries to write the number 123 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 can see that we are accessing a memory at virtual 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 a command **dmesg | ksymoops** (which uses 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   fffffeb <_EIP+0xfffffeb>
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 the value of an instruction pointer register (EIP) and in which function this value is. We can 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, there 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 can see that the `buggy` function was called from `my_fuction` and this was called from `init_module`. The reason why the stack trace is not exact is that the Linux kernel uses an approximation algorithm for the trace extraction.

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 2.3.3, *Finding an Error in Source Code*.

2.3.2. 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 as easy as in the previous section. Let's study example `buggy1.c`. There 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 executes the bad instruction, an oops message similar to that one in a previous example is printed. In this oops message we should notice a value of the EIP register. On my computer I can see:

```
 printing eip:
c8852065
```

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

2.3.3. Finding an Error in Source Code

Now we have exact information about an instruction that caused 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 disassembles 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 can see that on 5th byte after beginning of the `buggy` function is instruction `mov %eax,0x0` and this instruction belongs to C source line `*p = 123;`.

2.4. Assignment

Students will be given an arbitrary RT-Linux application with source code, which contains some errors which prevent the application from functioning properly. The student's goal is to find and fix those errors.

Notes

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

Chapter 3. Controller of DC motor

3.1. Introduction

This chapter contains a description of a real RT-Linux application. The goal of this case study is to write an RT-Linux application that controls velocity of DC motor equipped with an IRC (Incremental Radial Counter) sensor. The motor is connected to a PC through a simple electronic circuit consisting of a motor driver and basic logic parts. This circuit is to be plugged in to the PC printer port. Schematics for the board can be found in Appendix A, *Schematics of Motor Driver Board*.

The application consists of two programs. One is a RT-Linux kernel module that builds 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.

In the following sections we are going to describe the parts of the application in a more detailed manner. It can be useful if you can look at the source code while reading this text. 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 will describe particular pieces of this module.

3.2.1. The Motor Data 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 called *methods* in object oriented terminology. 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 we call 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. Declaration 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. The 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 the motor structure that are used to lock various parts of the 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 for SMP machine, 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. We can do this because the 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*) from doing its work.

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

3.2.2. Initialization

Initialization is done in the function `motor_init`. From there the function (method) `start_motor` is called in order to initialize the motor structure, mutexes, FIFOs and so on. It also starts three threads that implement the controller. These threads are:

- the PWM thread, for generating PWM signal,
- the measure thread, which measures current position and speed at regular intervals and
- the 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 the `PWM_PERIOD` symbol. Default value is 1 ms.

```
void *pwm(void *arg)
{
        struct timespec delay;
        struct motor *motor = (struct motor *)arg;

        delay.tv_sec = 0;
        while (1) {
                pthread_mutex_lock(&motor->action_lock);
                if (motor->action > 0)
                        set_output(motor, MOTOR_FWD); ❶
                else if (motor->action < 0)
                        set_output(motor, MOTOR_REV);
                else
                        set_output(motor, MOTOR_STOP);

                if (abs(motor->action) < PWM_RESOLUTION) {
                        int nsec = PWM_PERIOD * abs(motor->action) / PWM_RESOLUTION;
                        pthread_mutex_unlock(&motor->action_lock);
                        delay.tv_nsec = nsec;
                        nanosleep(&delay, NULL); ❷
                        set_output(motor, MOTOR_STOP); ❸
                }
                else pthread_mutex_unlock(&motor->action_lock);

                pthread_wait_np(); ❹
        }
        return 0;
}
```

Figure 3-2. Code of the `pwm` function

In the beginning of the loop (Figure 3-2, ❶), one of the two output pins of the printer port connected to the motor driver is set. Which bit is set depends on sign of *action value*. After that the thread sleeps for some time that depends on an *action value* too (❷). After waking up, the output pin is reset to zero (❸). It remains unset until beginning of next period (❹).

3.2.4. Action Value Calculation

The action value (the value used by PWM thread) is calculated by a simple PSD (proportional-sum-differential) controller. The controller is implemented in the function `regul` (see Figure 3-3). This function blocks at the beginning (❶) 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 the next section.

```
void *regul(void *arg)
{
        struct motor *motor = (struct motor *)arg;
        struct motor_status status;
        int retval;
        int action;

        while (1) {
                /* wait for measured value */
                pthread_suspend_np(pthread_self()); ❶

                pthread_mutex_lock(&motor->lock);

                action = calc_action(motor) * PWM_RESOLUTION >> 16;

                pthread_mutex_lock(&motor->action_lock);
                motor->action = action; ❷
                pthread_mutex_unlock(&motor->action_lock);

                /* send status to user space program */
```

```
                status.velocity = motor->velocity;
                status.position = motor->position;
                status.action = motor->action;
                retval = rtf_put(motor->fifo_out, &status, sizeof(status)); ❸

                pthread_mutex_unlock(&motor->lock);
        }
        return 0;
}
```

Figure 3-3. Code of the `regul` function

The calculated value is then stored in the `motor` structure for use by the `pwm` thread (❷) and current status is sent to the user-space program via the RT FIFO (❸).

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 numbers. Sixteen least significant bits are used to represent a decimal part of a number and 16 most significant bits represent an integer part.

We can add or subtract these numbers like ordinary integers. When we want to multiply them, we can choose one 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.

Which method is used depends on expected values of operands and desired precision. If we for example know, one operand is always smaller than one and we use the third method and shift that operand rightwards by 16 bits, we get zero, which is probably not what we wanted.

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 the explanation of the 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 have to read a two-bit value of IRC sensor (❶) 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 (❸) or decremented. Also the actual time is remembered in variable `last_irq_time` (❷) and a time since last IRQ is computed and saved in variable `last_irq_interval`. See Figure 3-5 for graphic representation of this.

```
char aper_seq[4] = {0, 1, 3, 2};

unsigned irq_handler(unsigned int irq, struct pt_regs * regs)
{
        struct motor *motor = &my_motor;
        int aper;
        char pos, last_pos;
        hrtime_t now;

        /* This prevents other processor to manipulate our data. */
        rtl_spin_lock(motor->spinlock);

        /* deterine direction */
```

```
        aper = read_apertures(motor); ❶
        pos = aper_seq[aper];
        last_pos = aper_seq[motor->last_aper];

        now = clock_gethrtime(CLOCK_REALTIME);
        motor->last_irq_interval = now - motor->last_irq_time;
        motor->last_irq_time = now; ❷

        if      (pos == (last_pos+1) % 4) {
                motor->delta_pos++; ❸
                motor->dir = +1;
        }
        else if (pos == (last_pos+3) % 4) {
                motor->delta_pos--;
                motor->dir = -1;
        }
        else {
                rtl_printf("lost interrupt\n");
        }

        motor->last_aper = aper;

        rtl_spin_unlock(motor->spinlock);

        rtl_hard_enable_irq(irq); /* This irq is disabled in this
                                   * routine, so we have to reenable
                                   * it. */
        return 0;
}
```

Figure 3-4. Code of `irq_handler` function

3.2.5.2. Measure Thread

This periodic thread is implemented in function `measure`. Before describing this thread, let's look at Figure 3-5. This is a graphical explanation of what is done when position is measured. On the horizontal axis there is time and on the 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 the motor. The figure thus shows that, at beginning, the motor rotates with constant velocity in one direction, than it quickly rotates to the other direction and finally it rotates to the same direction as at beginning.

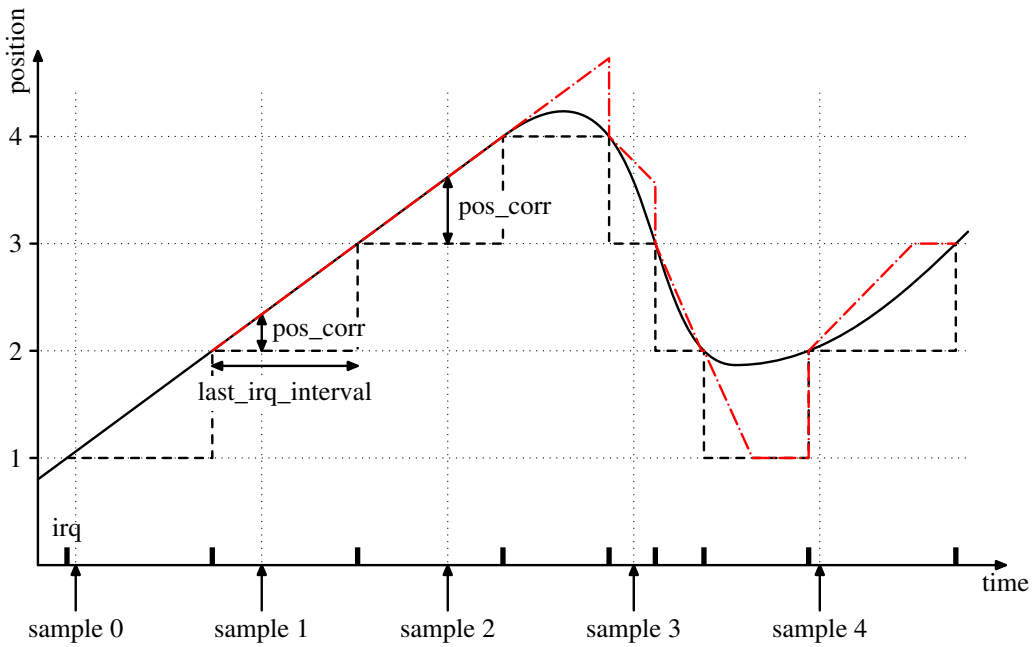

Figure 3-5. Measurement of position

The simpler version of a measuring algorithm would do only these steps (in regular intervals corresponding to sample marks on the horizontal axis):

- `delta_pos` is saved to temporal variable and zeroed. While 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 a value of `delta_pos`.

This method of position measuring gives us the results that are depicted in Figure 3-5 by the black dashed line. The inaccurate measuring of position in this simple version isn't big issue. The measuring of velocity is worse. When the motor rotates with constant velocity, the measured velocity isn't constant and oscillates between two adjacent values. As we use the measured velocity to close the feedback loop of the PSD controller, we need a better method of velocity measuring.

The method we use in our case study tries to estimate the real position depending on time elapsed since last IRQ. In each sampling period the correction of position is calculated (in Figure 3-5 denoted by pos_corr). 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 3-5 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

The 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 an user-space application, this library is automatically linked with the application. Since this is not true for compiling kernel modules and we have to 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 an actual velocity and position of the motor.

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

```
int main(int argc,char ** argv)
{
        int fifo_in;            /* to send desired value (input to motor.c) */
        int fifo_out;           /* to read motor state */
        float value = 10;

        struct motor_status mstate;
        struct timeval timeout;
        int retval;
        fd_set rdfs;

        if ((fifo_in = open("/dev/rtf0", O_WRONLY)) < 0) { ❶
                fprintf(stderr, "Error opening /dev/rtf0\n");
                exit(1);
        }

        if ((fifo_out = open("/dev/rtf1", O_RDONLY)) < 0) {
```

```
                fprintf(stderr, "Error opening /dev/rtf1\n");
                exit(1);
        }

        while (1) {             /* exit with Ctrl-C */
                timeout.tv_sec = 0;
                timeout.tv_usec = 10*1000; /* 10 ms */
                FD_ZERO(&rdfs);
                FD_SET(0, &rdfs); /* stdin */
                FD_SET(fifo_out, &rdfs); /* motor state */
                retval = select(fifo_out + 1,
                                &rdfs, NULL, NULL, &timeout); ❷

                if (FD_ISSET(0, &rdfs)) {
                        int i;
                        printf("Rotation: ");
                        fflush(stdout);
                        scanf("%f",&value);
                        i = (int)(value  * (1<<16)); /* fixed point aritmetic */
                        write(fifo_in, &i, sizeof(i));
                        printf("Rotation set to %.2f rps\n", value);
                        fflush(stdout);
                }
                if (FD_ISSET(fifo_out, &rdfs)) {
                        retval = read(fifo_out, &mstate, sizeof(mstate));
                        printf("\rPos: %8.2f Vel: %8.2f Action: %8d",
                               (double)mstate.position/(1<<16),
                               (double)mstate.velocity/(1<<16),
                               mstate.action
                                );
                        fflush(stdout);
                }
        }
        return 0;
}
```

Figure 3-6. User-space part of the controller

At the beginning two FIFOs are opened (❶). One is used for reading status from the real-time part and one for writing data to the real-time part. After that, the program enters an endless loop, so the user can quit the program only by pressing **Ctrl-C**[2]. The loop begins by waiting (with timeout) for either data from real-time part or input from keyboard via STDIN. This waiting is performed in `select()` system call (❷). After returning from this call we test if some file descriptor has ready 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 the extension of its functionality, but it wouldn't show us more RT-Linux features.

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

3.5. Assignment

1. Modify both the real-time and the user-space part to allow tuning of controller parameters (REG_P etc.) on-line without recompilation.

   You will need to define variables for each parameter and extend the communication procedures to allow sending of parameter change requests. Take care of proper use of mutexes!

2. Extend the real-time part in a way that the motor will rotate only 10 seconds after the user enters the new velocity. After the time elapses, the motor will automatically stop.

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 as well. This is common practice in Linux kernel and some other bigger C projects.
2. This is quite common practice in Unix programs.

Chapter 4. CANping – a simple LinCAN testing application

4.1. Introduction

This chapter describes a simple application called canping. Its primary goal was to test the LinCAN driver but it also nicely illustrates how to use LinCAN driver in real applications. In addition, it shows some maliciousness of the LinCAN driver.

Canping is a multithread application, which sends and receives messages in parallel and thus can be used for stress testing of the LinCAN driver. For basic usage, two running instances of canping, each one on a different host, are needed. One instance sends messages and waits for replies and the other instance waits for the sent messages and sends replays with an ID incremented by one. Canping can also be run twice on one host provided that we are either using a virtual CAN device or there are multiple CAN cards/chips in the host.

4.2. CANping manual

canping

Name

`canping` — Multi-thread LinCAN testing utility

Synopsis

canping [-m *num* | -s *num*] [-c *count*] [-d *device*] [-h] [-i *id*] [-l *length*] [-o] [-t *timeout*] [-y *count*]

Options

`-m` *num*

Start in the master mode and run *num* master threads in parallel.

`-s` *num*

Start in the slave mode and run *num* slave threads in parallel.

`-c` *count*

Every master thread will send only *count* messages and then finishes. Without this option messages are sent forever and CANping can be terminated by a signal e.g. by pressing a Ctrl+C key.

`-d` *device*

CAN device to be used. Default is `/dev/can0`.

`-i` *id*

Select ID of the lowest generated or responded message. Default is 1000. CANping in the master mode will generate messages with IDs *id*, *id*+2, *id*+4, etc. Each master thread will generate different ID. The slave threads will listen to these IDs and will answer with an ID increased by one.

`-l` *len*

Specify a length of the messages. Default is 8 and possible values are from 0 to 8.

`-t` *sec*

A timeout in seconds; this option specifies how long the master thread will wait for the response message from a slave. If the response won't arrive in *SEC* seconds, the message is considered as loosed.

`-v`

Increase verbosity. Without this option only summary statistics are displayed just before the program finishes. One -v means to display a global status (number of sent messages and timeouts) during program execution. Two -v options display a simple message for every packet in the format **ID:time** where time is measured in microseconds. Three -v options display more verbose information about each packet.

`-w` *msec*

Wait time in milliseconds before the master thread sends a next message. Default value is 1000 ms.

`-y`

Synchronize the master threads before sending the first message. When CANping is stared with the high number of master threads, usually the first created thread will begin by sending the messages before the last thread is created. Sometimes it may be useful that every thread will wait before sending anything to the other threads. Only when all threads are created and prepared for sending, sending can start.

Exit codes

The canping exit code depends on how the program finishes. This feature can be used for (semi)automatic testing of LinCAN driver. Exit codes are listed in the table bellow.

0	OK
1	Bad command line parameter
2	Problem with opening LinCAN driver
3	Problem with filter
4	Insufficient memory
5	Read syscall error
6	Write syscall error
7	Select syscall error
8	Flush (ioctl syscall) error

Examples

On one computer one runs slave with 10 threads:

```
root@pc104:~% canping -d /dev/can1 -s 10
```

and on another computer master is run with 10 threads too:

```
sojka@glab:~% canping -d /dev/can4 -m 10 -v -c 100 -w 10
Total count: 1000, Timeouts: 0
Summary statistics:
Id 1000: count = 100 mean = 8005.16 stddev = 691.01 min = 2022 max = 11133 [us] loss = 0% (0)
Id 1002: count = 100 mean = 8013.10 stddev = 641.19 min = 4013 max = 12885 [us] loss = 0% (0)
Id 1004: count = 100 mean = 8035.99 stddev = 575.36 min = 5996 max = 13081 [us] loss = 0% (0)
Id 1006: count = 100 mean = 8062.38 stddev = 626.46 min = 7005 max = 14061 [us] loss = 0% (0)
Id 1008: count = 100 mean = 8082.63 stddev = 740.59 min = 5963 max = 15020 [us] loss = 0% (0)
Id 1010: count = 100 mean = 8089.04 stddev = 831.64 min = 7018 max = 15970 [us] loss  = 0% (0)
Id 1012: count = 100 mean = 8105.02 stddev = 952.95 min = 7011 max = 16970 [us] loss = 0% (0)
Id 1014: count = 100 mean = 8129.57 stddev = 1037.39 min = 7020 max = 17906 [us] loss = 0% (0)
Id 1016: count = 100 mean = 8144.46 stddev = 1154.17 min = 7019 max = 18870 [us] loss = 0% (0)
Id 1018: count = 100 mean = 8157.85 stddev = 1259.84 min = 5994 max = 19842 [us] loss = 0% (0)
```

At the end of program execution, there are statistics for each packet type (ID). These contain mean round trip times (RTT), standard deviations of RTT, minimal and maximal RTT and packet losses (percentage and absolute).

4.3. Implementation details

The structure of canping is as follows. After the initialization and parsing of command line options (function `parse_optins()`), communicating threads are started. The code of these threads is made up by `master_thread()` and `slave_thread()` functions which are described in the following section.

After these threads are finished (either after pressing **Ctrl-C** or due to `-c` switch), summary statistics are written to stdout and the application finishes.

4.3.1. Usage of the LinCAN driver

This section will cover `master_thread()` function. Code of `slave_thread()` is very similar to `master_thread()`, so it will not be covered here. These functions represent typical usage of the LinCAN driver in applications and if you are looking for examples how to use the LinCAN driver, this is probably the most important section for you.

The `master_thread()` function is responsible for opening the driver, sending message with a particular ID and waiting for response message. Sending and receiving messages is done periodically in a loop.

4.3.1.1. Opening the Driver

At the beginning of every access to the LinCAN driver, the driver has to be opened. This is done by calling `open()`.

```
canfd = open(option_device, O_RDWR)
```

The value of the `option_device` variable specifies the name of CAN device e.g. `/dev/can0` and the second parameter contains additional flags. We are using a `O_RDWR` flag which means that we want to use the driver for both reading and writing (messages). You can also use the `O_NOBLOCK` flag in addition. For further details regarding this flag see the LinCAN documentation.

If the opening operation succeeds, the `canfd` variable contains a valid file descriptor, which is used for further communication with the driver. In the case of an error, we use `perror()` function to print the reason of the error and then we exit the program with appropriate exit code[1].

4.3.1.2. Setting Filters

After the device is open, the second step is to create a filter for message receiving. The filter assures this thread only receives messages it is waiting for.

```
/* setup filtering of received messages */
memset(&canfilt, 0, sizeof(canfilt));
canfilt.mask = 0xffffffff;
canfilt.id = pong_id; /* pong responces with increased id */
ret = ioctl(canfd, CANQUE_FILTER, &canfilt);
```

The filter is created by filling the canfilt_t structure and submitting it to the `CANQUE_FILTER` ioctl. Filters allow us to filter messages by various criterions. We use only filtering by an ID. The `mask` member tells which bits of message ID are relevant for filtering and the member `id` contains desired ID bit values. Since we need only to receive messages with one ID `mask` is set to all ones and `id` contains the ID.

After the filter is set up it is necessary to flush driver queues. The reason for this is that in the time between driver opening and filter setup, there can be some received messages in the queue, which doesn't match the filter criteria. Flushing the queue is done by calling `CANQUE_FLUSH` ioctl.

```
ret = ioctl(canfd, CANQUE_FLUSH, NULL);
```

4.3.1.3. Reading and Writing

As soon as these operations are done, everything is ready for message sending and reception. The message is sent by calling the `write()` function with canmsg_t structure as the second parameter. The canmsg_t structure should be filled according to the message we wish to send. This comprises of a message ID, message data bytes, the length of message etc. The structure for the ping message is filled by the following commands:

```
pingmsg.flags=0;
pingmsg.id=ping_id;
pingmsg.length = option_length;
for (i=0; i < option_length; i++) pingmsg.data[i] = i;
```

Later, the message is sent by:

```
ret = write(canfd, &pingmsg, sizeof(pingmsg));
```

After the message is sent, the program starts waiting for the response message. The `select()` system call is used for this purpose because it allows us to wait with timeout. If there is a received message, we can read it from the driver by calling `read()` function. Before doing that, it is important to zero `flags` field of canmsg_t structure. This is due to a feature of the driver. Pavel Pí[1]a describes this as follows:

> Adding "`msg.flags=0;`" before "`read()`" call is required, because random value could trigger RTR read patch in the driver. This obsolete driver read mode should be moved to its own IOCTL in future.

```
/* Read the message */
pongmsg.flags=0;
ret = read(canfd, &pongmsg, sizeof(pongmsg));
```

4.3.1.4. Closing

At the end, when it is not needed to work with the driver, it should be closed in order to remove all driver resources associated with our application. The driver is closed simply by calling `close()`.

```
close(canfd);
```

4.3.2. Other parts of the application

4.3.2.1. Lists

For managing the list of executing threads and their statistics a list implementation from Pavel Pí[1]a's uLan utils (ulut) package was used. This framework allows us very simple and efficient list handling.

First, list head should be declared:

```
typedef struct threads {
        ul_list_head_t head;
} threads_t;
```

Since we don't need any additional data, our list head has only one field of the type ul_list_head_t. Next we declare a list element:

```
typedef struct thread_data {
        pthread_t tid;
        long int canid;

        int count;
        double mean;            /* mean value of responses */
        double moment2nd;       /* used to compute variance of
                                 * responses */
        int min, max;           /* min/max response times */
        int timeout;            /* number of timeouts */

        ul_list_node_t node;
} thread_data_t;
```

This structure is used for storing data for every executed thread. In the next step we declare functions for list manipulation. These functions are created automatically by the `UL_LIST_CUST_DEC` macro.

```
UL_LIST_CUST_DEC(thread_list, threads_t, thread_data_t, head, node);
```

This declares some functions whose names start with the `thread_list_` prefix. We use two of these, namely `thread_list_init_head()` for list initialization and `thread_list_ins_tail()` for adding elements to the list.

We also use a macro `ul_list_for_each()` (declared in `ul_list.h`) which traverses through all the list elements in a loop.

4.3.2.2. Waiting for the thread completion

After the main thread starts all the communication threads, it is necessary to wait for their completion. This is done in `wait_for_threads()`. In the simplest case, waiting runs in a while loop and in every iteration is is waited for one thread using a `finish_sem` semaphore. After the `thread_count` iterations, we are sure all the threads finished. Whenever any thread finishes, it increments this semaphore by calling `sem_post()` and this is why we can wait for a particular number of threads to finish.

The simplest case of waiting code looks as follows:

```
while (thread_count > 0) {
        ret = sem_wait(&finish_sem);
        if (ret == 0) thread_count--;
}
```

In the canping application we use more difficult code because of printing of progress messages during waiting.

Finally, when all the threads are finished, we print summary statistics for each thread and free list elements:

```
ul_list_for_each_cut(thread_list, &master_threads, td) {
        print_stats(td);
        free(td);
}
```

4.3.2.3. Signal handling

In order to exit canping by pressing **Ctrl-C** (`SIGINT`) or by sending other signal such as `SIGTERM` it is necessary to write and register a signal handler. The handler is registered by:

```
siginterrupt(SIGINT, 1);
signal(SIGINT, term_handler);
siginterrupt(SIGTERM, 1);
signal(SIGTERM, term_handler);
```

This registers the `term_handler` function as the handler for the `SIGINT` and `SIGTERM` signals. The call to the `siginterrupt()` function tells the OS that an arrived signal should interrupt currently executed system call[2]. This is necessary because the master and slave threads executes `select()` or `read()` system call, which cause waiting for external event, which can never happen and we want the program to exit even if there is no event.

The signal handler looks as follows:

```
#define NOT_INTERRUPTED_SYSCALL (errno != EINTR && errno != ERESTART)
#define IS_FINISH_FLAG() (finish_flag)

void term_handler(int signum)
{
        if (!IS_FINISH_FLAG()) {
                finish_flag = 1;
                kill_all_threads(signum);
        }
}
```

Whenever the main thread receives a signal it sets `finish_flag` to prevent recursive call to the handler and then sends the same signal to all other threads. The other threads receive the signal and execute the handler. Because the `finish_flag` is set the handler does nothing. The only result of the signal is interruption of currently executed system call. As a consequence, the thread exits the send-receive loop and executes exit code.

You can notice the macro `IS_FINISH_FLAG`. This macro is defined only for debugging purposes so don't be confused by it.

4.4. Assignment

1. Create a simple chat application that will send messages over the CAN bus. The application will read the characters typed on keyboard and send them to the bus. In addition, it will receive CAN messages from other nodes on the bus and print then on screen.
2. Extend the chat application by displaying messages in color depending on the ID of sender. For color and screen management use for example the ncurses library (http://www.tldp.org/HOWTO/NCURSES-Programming-HOWTO/).

Notes

1. The exit code can be used by an automatic regression tests to detect the reason of failure.
2. The behaviour of signals is different in Linux 2.4 and Linux 2.6 with NPTL. The call to `siginterrupt()` is necessary in order to get the same behaviour for both 2.4 and 2.6.

Appendix A. Schematics of Motor Driver Board

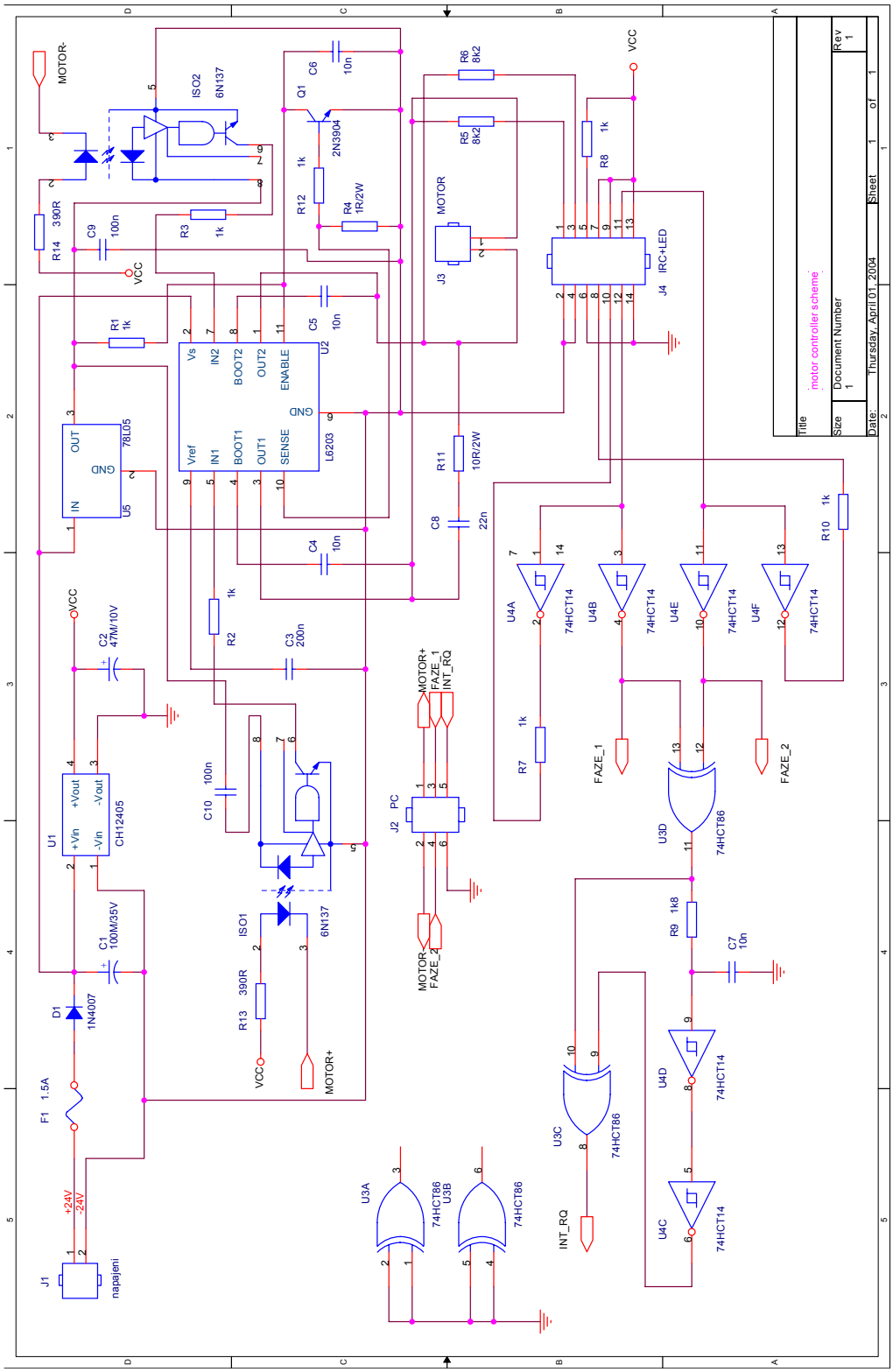

Figure A-1. Motor controller scheme

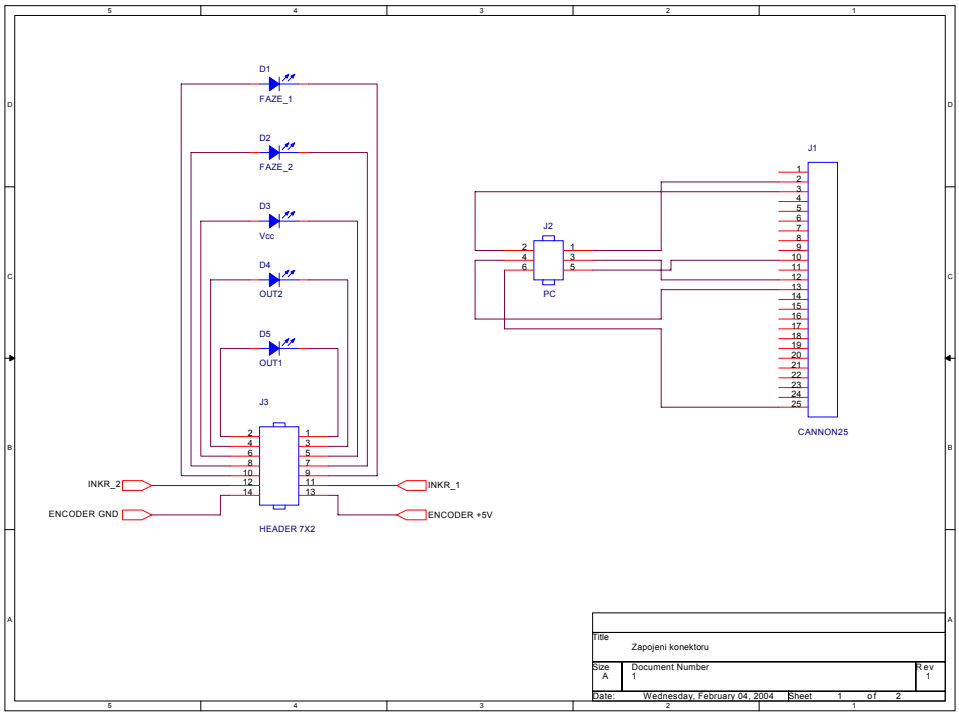

Figure A-2. Scheme of connectors for motor controller

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