DM510 - Operating Systems
Project 3: Kernel Module

Jeff Gyldenbrand (jegyl16)
Supervisor: Daniel Merkle
Southern University of Denmark

April 23, 2018

Contents

1 Introduction

This project is done in collaboration with Jonas Sørensen (joso216) and Simon D. Jørgensen (simjo16).

The goal of this project is to make a kernel module which implements a driver that exposes two character devices to user-space. This driver solves the producer-consumer problem, meaning the devices can both produce (write) and consume (read) data.

More specifically, two devices is to be implemented: dm510-0 and dm510-1. When a process writes to dm510-0, the data has to be stored in a buffer (buffer 1) that resides in kernel-space. In case this buffer is full, it must wait until another process reads from device dm510-1, and thereby emptying the buffer. When a process reads from dm510-0, the data is read from buffer 0, and now, if this buffer is empty, the process has to wait until another process writes to dm510-1. See fig:1 for graphical representation of this explanation.

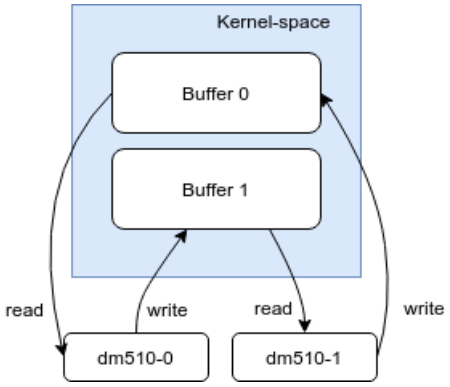

fig:1

2 Design decisions

With the scull driver as inspiration source are large sections of the code, and features directly used from here, with minor changes. For more transparent code, the buffer is divided into its own header file: buffer.h, also, the ioctl-commands has its own header: dm510_iotcl-commands.h.

2.1 Devices

First the devices is setup and initialized, meaning, the device region is registred and memory is allocated for the buffers. Then a open-function handles whenever processes tries to open af device file, and a relase-function is invoked whenever a process closes a device-file. Furthermore a read-function handles the case where a process attempts to read from an already opened device. And lastly a write-function handles whenever a process writes to a device. Furthermore a systemcall for ioctl is made to ..

2.2 Buffers

In the buffer header several functions is made. Firstly an initialization of a new buffer of given size with the buffer_init-function. A buffer_free_space-function to see how much

space we can write to. A buffer_resize-function to support scaling of the buffers. A buffer_free-function to free the buffer from memory. And A buffer_write-function and buffer_read-function to handle the read- and writes of the buffers.

3 Implementation

3.1 Devices

3.2 Buffers

The buffers consists of the struct buffer, and has a char pointer rp (read) and wp (write). The buffer is initialized in buffer_init-function with kmalloc, and setting both rp and wp to the beginning of the buffer.

To see how much space we can write to, this functions checks if read- and write-pointer is at same spot, hence, the whole buffer can be written to, else it calculates the write-space between them. See listing 1.

Listing 1: buffer.h :: buffer_free_space

```
1  if (head->rp == head->wp)
2    return head->size - 1;
3  return ((head->rp + head->size - head->wp) % head->size) - 1;
```

For the buffer_resize-function memcpy is invoked to copy complete memory blocks, and return a pointer to the new destination, and in this way, resize a buffer.

To free a buffer it is simply resized to zero. This is possible because if krealloc's new_size is 0 and p is not a NULL pointer, the object pointed to is freed. See listing 2.

Listing 2: buffer.h :: buffer_free

```
1        kfree(head->buffer);
2        head->buffer = NULL;
3        return 0;
```

The buffer_write-function checks where the read- and write-pointer is relative to each other, to deside where to write a given input. This is the same principle for the buffer_read-function, to deside where to read from and to.

3.3 Processes (sleep / awakening)

When the module is initialized in the dm510_init_module-function a queue is initialized for each device. See listing 3.

Listing 3: dm510_dev.c :: dm510_init_module

```
1        for ( i = 0; i < DEVICE_COUNT; i++) {
2                init_waitqueue_head(&devices[i].inq);
3                init_waitqueue_head(&devices[i].outq);
4                mutex_init(&devices[i].mutex);
5                dprintf("Device(\%d) = (\%d, \%d)",i,(i \% BUFFER_COUNT), ((i + 1)
                      \% BUFFER_COUNT));
```

```
6               devices[i].read_buffer = buffers + (i \% BUFFER_COUNT);
7               devices[i].write_buffer = buffers + ((i + 1) \% BUFFER\_COUNT);
8               DEBUG_CODE(printk(""););
9               frame_device_setup(devices+i, global_device+i );
10      }
```

If the read- and write-pointer at any given time, in the dm510_read-function, is at the same spot, wait_event_interruptible is invoked. Adding the current thread to the queue, and calls the scheduler to schedule a new thread. See listing 4

Listing 4: dm510_dev.c :: dm510_read

```
1 if(wait_event_interruptible(dev->inq,(*rp != *wp))){
2       return rerror(-EAGAIN, "Reader was interrupted while sleeping.");
3 }
```

This is also handled in the buffer_write-function.

4 Test

Nine test has been made. The first test, as seen in the video at time-frame: 0:37 - 0:44, is a read- and write-test, with the program ./readwrite "I love kernel panic", to see that its possible to send a message and read it.

The second test is at time-frame: 0:59 - 1:08, to see when read from an empty buffer with blocking. Notice a ctrl-c is applied to interrupt.

The third test is at time-frame: 1:15 - 1:21, to see the same case as the second test, only with a non-blocking. This time an error is prompted: Error (-1).

The fourth test is at time-frame: 1:29 - 1:37, to see the case where more readers than allowed is applied. As shown, only 9 readers are allowed, the 10th reader prompts an -1 error: invalid pointer.

The fifth test is at time-frame: 1:45 - 1:48, to see the case where more than one writer is applied. Only one writer is legal, hence, the second writer prompts an error -1: invalid pointer.

The sixth test is at time-frame: 1:56 - 2:06, to see the case where a buffer that is full is written to, with blocking. Notice that ctrl-c is applied to interrupt.

The seventh test is at time-frame: 2:12 - 2:17, to see the same case as the sixth test, only with non-blocking. This time it prompts "Buffer is full"

The eigth test is at time-frame: 2:26 - 2:29, this test is the given moduletest. As shown it opens with success on r and w. Result 1 matches expected result 1, and result 2 matches expected result 2.

The final test is at time-frame: 2:37 - 3:12, to see that its possible to resize the buffer. In this case its resized to first "2024". Then to check that it is not possible to resize the buffer below used space, an string 'hello friend' is echoed into the device dm510-0, and its

attempted to resize the buffer to size "5". As shown en error is prompted: "Can't contract buffer size 5 below used space 14".

5 Discussion

Multiple processes to use a device simultaneously is not possible because only one writer per device can exist. If more writers is created, they will be rejected. Its possible to have more than one reader. But this is also handled.

6 Conclusion

A kernel module with a driver that exposes two character devises to user-space has been developed. It handles the producer-consumer problem in that, the devices can both write and read data, stored from buffers in kernel-space. Furthermore a process cant read from an empty buffer, but actively waits for another process to write to said buffer, before reading from it. And vice verca, a process cant write to a full buffer, so it wait to the buffer is emptied by another process. As shown in the test-section the major corner-cases is handled too. Every requirements has been met.

7 Appendix

7.1 Source-code

dm510_dev.c

```
1  /* Prototype module for second mandatory DM510 assignment */
2  #ifndef __KERNEL__
3  #  define __KERNEL__
4  #endif
5  #ifndef MODULE
6  #  define MODULE
7  #endif
8
9  //#define DEBUG
10 #include <linux/cdev.h>
11 #include <linux/module.h>
12 #include <linux/kernel.h>
13 #include <linux/fs.h>
14 #include <linux/uaccess.h>
15 /* Prototypes - this would normally go in a .h file */
16
17 //own header file for buffer, to make code readable
18 #include "buffer.h"
19 #include "dm510_ioctl_commands.h"
20 static int dm510_open( struct inode*, struct file* );
21 static int dm510_release( struct inode*, struct file* );
22 static ssize_t dm510_read( struct file*, char*, size_t, loff_t* );
23 static ssize_t dm510_write( struct file*, const char*, size_t, loff_t* );
24 long dm510_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
25
26 #define DEVICE_NAME "dm510_dev" /* Dev name as it appears in /proc/devices */
27 #define MAJOR_NUMBER 254
28 #define MIN_MINOR_NUMBER 0
29 #define MAX_MINOR_NUMBER 1
30
31 /* end of what really should have been in a .h file */
32
33 /* file operations struct */
34 static struct file_operations dm510_fops = {
35        .owner  = THIS_MODULE,
36        .read   = dm510_read,
37        .write  = dm510_write,
38        .open   = dm510_open,
39        .release = dm510_release,
40   .unlocked_ioctl = dm510_ioctl
41 };
42
43 struct frame {
44        wait_queue_head_t inq, outq;    /* read and write queues */
45        struct buffer * read_buffer;
46        struct buffer * write_buffer;
47        int nreaders, nwriters;         /* number of openings for r/w */
48        struct fasync_struct *async_queue; /* asynchronous readers */
49        struct mutex mutex;           /* mutual exclusion semaphore */
50        struct cdev cdev;
51 };
```

```
52
53  static struct frame devices[DEVICE_COUNT];
54  static struct buffer buffers[BUFFER_COUNT];
55  static size_t max_processes = 10;
56
57  dev_t global_device = MKDEV(MAJOR_NUMBER,MIN_MINOR_NUMBER);
58
59  static int frame_device_setup(struct frame * dev, dev_t device){
60          cdev_init(&dev->cdev, &dm510_fops);
61          dev->cdev.owner = THIS_MODULE;
62          return cdev_add(&dev->cdev, device, 1);
63  };
64
65  #define BUFFER_DEFAULT_SIZE 4096
66
67  int dm510_init_module( void ) {
68          int i, result;
69          result = register_chrdev_region(global_device,DEVICE_COUNT,DEVICE_NAME);
70          if(result){
71                  return rerror(result, "Failed to register chrdev_region.");
72          }
73          for (i = 0; i < BUFFER_COUNT; i++) {
74                  result = buffer_init(buffers+i, BUFFER_DEFAULT_SIZE);
75                  if(result < 0) return rerror(result, "Could not allocate memory
                        for buffer(%d).", i);
76          }
77          for ( i = 0; i < DEVICE_COUNT; i++) {
78                  init_waitqueue_head(&devices[i].inq);
79                  init_waitqueue_head(&devices[i].outq);
80                  mutex_init(&devices[i].mutex);
81                  dprintf("Device(%d) = (%d, %d)",i,(i % BUFFER_COUNT), ((i + 1) %
                        BUFFER_COUNT));
82                  devices[i].read_buffer = buffers + (i % BUFFER_COUNT);
83                  devices[i].write_buffer = buffers + ((i + 1) % BUFFER_COUNT);
84                  DEBUG_CODE(printk(""););
85                  frame_device_setup(devices+i, global_device+i );
86          }
87
88          return 0;
89  }
90
91  /* Called when module is unloaded */
92  void dm510_cleanup_module( void ) {
93          int i;
94          for(i = 0; i < DEVICE_COUNT ; i++){
95                  if(devices[i].write_buffer) cdev_del(&devices[i].cdev);
96          }
97          for(i = 0; i < BUFFER_COUNT ; i++){
98                  if(buffers[i].buffer) buffer_free(buffers+i);
99          }
100         unregister_chrdev_region(global_device,DEVICE_COUNT);
101
102 }
103
104
105 /* Called when a process tries to open the device file */
```

```
106 static int dm510_open( struct inode *inode, struct file *filp ) {
107         struct frame * dev;
108         dprintf("Open");
109         dev = container_of(inode->i_cdev, struct frame, cdev);
110         filp->private_data = dev;
111         if(mutex_lock_interruptible(&dev->mutex)){
112                 return rerror(-ERESTARTSYS, "Mutex lock was interrupted.");
113         }
114 
115         if (filp->f_mode & FMODE_READ){
116                 if(dev->nreaders >= max_processes ){
117                         mutex_unlock(&dev->mutex);
118                         return rerror(-ERESTARTSYS, "Too many readers, only %d are
                                allowed.", max_processes);
119                 } else{
120                         dev->nreaders++;
121                 }
122         }
123         if (filp->f_mode & FMODE_WRITE){
124                 if(dev->nwriters >= 1){
125                         mutex_unlock(&dev->mutex);
126                         return rerror(-ERESTARTSYS, "Amount of writers exceeded the
                                 allowed capacity of 1.");
127                 }else{
128                         dev->nwriters++;
129                 }
130 
131         }
132         mutex_unlock(&dev->mutex);
133 
134         return nonseekable_open(inode, filp);
135 }
136 
137 
138 /* Called when a process closes the device file. */
139 static int dm510_release( struct inode *inode, struct file *filp ) {
140         struct frame * dev = filp->private_data;
141         dprintf("Release");
142         //scull_p_fasync(-1, filp, 0);
143         mutex_lock(&dev->mutex);
144         if (filp->f_mode & FMODE_READ && dev->nreaders)
145                 dev->nreaders--;
146         if (filp->f_mode & FMODE_WRITE && dev->nwriters)
147                 dev->nwriters--;
148         mutex_unlock(&dev->mutex);
149 
150         return 0;
151 }
152 
153 
154 /* Called when a process, which already opened the dev file, attempts to read
        from it. */
155 static ssize_t dm510_read( struct file *filp,
156     char *buf,     /* The buffer to fill with data */
157     size_t count,  /* The max number of bytes to read */
158     loff_t *f_pos ) /* The offset in the file       */
```

```
159 {
160         dprintf("Read.");
161         struct frame * dev = filp->private_data;
162         char **rp = &dev->read_buffer->rp;
163         char **wp = &dev->read_buffer->wp;
164 
165         if (mutex_lock_interruptible(&dev->mutex)){
166                 return rerror(-ERESTARTSYS, "Mutex lock was interrupted by outside
                        source.");
167         }
168 
169 
170 
171         while (*rp == *wp) {
172                 mutex_unlock(&dev->mutex);
173                 if (filp->f_flags & O_NONBLOCK){
174                         return rerror(-EAGAIN, "File-pointer could not be blocked
                            .");
175                 }
176                 dprintf("Read Sleeping.");
177                 if(wait_event_interruptible(dev->inq,(*rp != *wp))){
178                         return rerror(-EAGAIN, "Reader was interrupted while
                            sleeping.");
179                 }
180                 dprintf("Reader Awoken.");
181                 if(mutex_lock_interruptible(&dev->mutex)){
182                         return rerror(-ERESTARTSYS, "Mutex lock was interrupted by
                            outside source.");
183                 }
184         }
185         count = buffer_read(dev->read_buffer,buf,count);
186         mutex_unlock (&dev->mutex);
187         wake_up_interruptible(&dev->outq);
188         dprintf("Read : Done.");
189         return count;
190 }
191 
192 /* Called when a process writes to dev file */
193 static ssize_t dm510_write( struct file *filp,
194     const char *buf,/* The buffer to get data from */
195     size_t count, /* The max number of bytes to write */
196     loff_t *f_pos ) /* The offset in the file       */
197 {
198 
199         struct frame * dev = filp->private_data;
200         dprintf("Write.| wp = %d, rp = %d" ,
201         dev->write_buffer->wp - dev->write_buffer->buffer,
202         dev->write_buffer->rp - dev->write_buffer->buffer);
203         if (mutex_lock_interruptible(&dev->mutex))
204                 return rerror(-ERESTARTSYS, "Mutex lock was interrupted by outside
                        source.");
205 
206         if(count > buffers->size){
207                 return rerror(-EMSGSIZE, "Message exceeded the size of the buffer
                    .");
208         }
```

```
209
210         while (buffer_free_space(dev->write_buffer) < count) {
211                 //DEFINE_WAIT(wait);
212
213                 mutex_unlock(&dev->mutex);
214                 if (filp->f_flags & O_NONBLOCK){
215                         return rerror(-EAGAIN, "File-pointer could not be blocked
                                .");
216                 }
217                 if(wait_event_interruptible(dev->outq,
218                         (buffer_free_space(dev->write_buffer) >= count))){
219                         return rerror(-EAGAIN, "Writer was interrupted while
                                sleeping.");
220                 }
221
222                 if (mutex_lock_interruptible(&dev->mutex))
223                         return rerror(-ERESTARTSYS, "Mutex lock was interrupted by
                                outside source.");
224         }
225         count = buffer_write(dev->write_buffer,(char*)buf,count);
226         dprintf("Write : Waking Reader. | wp = %d, rp = %d" ,
227         dev->write_buffer->wp - dev->write_buffer->buffer,
228         dev->write_buffer->rp - dev->write_buffer->buffer);
229         int i;
230         for(i = 0 ; i < DEVICE_COUNT ; i++){
231                         wake_up_interruptible(&devices[i].inq);
232         }
233         //wake_up_interruptible(&dev->inq);
234
235         mutex_unlock (&dev->mutex);
236         dprintf("Write : Done.");
237         return count;
238 }
239
240
241 #define GET_BUFFER_SIZE 0
242 #define SET_BUFFER_SIZE 1
243 #define GET_MAX_NR_PROC 2
244 #define SET_MAX_NR_PROC 3
245
246 /* called by system call icotl */
247 long dm510_ioctl(
248     struct file *filp,
249     unsigned int cmd, /* command passed from the user */
250     unsigned long arg ) /* argument of the command */
251 {
252         switch(cmd){
253                 case GET_BUFFER_SIZE:
254                 return buffers->size;
255
256                 case SET_BUFFER_SIZE:{
257                         int i;
258                         for(i = 0 ; i < BUFFER_COUNT ; i++){
259                                 int used_space = buffers[i].size - buffer_free_space
                                        (buffers+i);
260                                 if(used_space > arg) {
```

```
261                                     return rerror(-EINVAL, "Buffer(%d) has %lu
                                            amount of used space, cannot be reduced
                                            to size %lu.", i, used_space, arg);
262                             }
263                     }
264                     for(i = 0 ; i < BUFFER_COUNT ; i++) {
265                             buffer_resize(buffers+i,arg);
266                     }
267             }
268             break;
269
270             case GET_MAX_NR_PROC:
271             return max_processes;
272
273             case SET_MAX_NR_PROC:
274             max_processes = arg;
275             break;
276
277             case GET_BUFFER_FREE_SPACE:
278             return buffer_free_space(buffers+arg);
279
280             case GET_BUFFER_USED_SPACE:
281             return buffers[arg].size - buffer_free_space(buffers+arg);
282     }
283
284
285     return 0; //has to be changed
286 }
287
288 module_init( dm510_init_module );
289 module_exit( dm510_cleanup_module );
290
291 MODULE_AUTHOR( "Jonas Ingerslev Soerensen, Jeff Gyldenbrand, Simon Dradrach
        Joergensen" );
292 MODULE_LICENSE( "GPL" );
```

dm510_ioctl_commands.h

```
1  #ifndef DM510_IOCTL_COMMANDS
2  #define DM510_IOCTL_COMMANDS
3
4  #define GET_BUFFER_SIZE 0
5  #define SET_BUFFER_SIZE 1
6  #define GET_MAX_NR_PROC 2
7  #define SET_MAX_NR_PROC 3
8  #define GET_BUFFER_FREE_SPACE 4
9  #define GET_BUFFER_USED_SPACE 5
10
11 #define DEVICE_COUNT 2
12 #define BUFFER_COUNT 2
13
14 #endif /* end of include guard: DM510_IOCTL_COMMANDS */
```

buffer.h

```
1  #ifndef DM510_BUFFER_H
2  #define DM510_BUFFER_H
```

```
3
4  #include <linux/slab.h>
5  #include <linux/errno.h>
6  #include "debug.h"
7
8  struct buffer{
9         char *buffer;
10   size_t size;
11        char *rp, *wp;
12   struct mutex mutex;
13 };
14
15 // return how much space we can write to. If read- and write-pointer is at same
       spot -> it means
16 // the whole buffer can be written to. Else, calculate the write-space between
       them.
17 size_t buffer_free_space(struct buffer * head){
18   if (head->rp == head->wp)
19     return head->size - 1;
20   return ((head->rp + head->size - head->wp) % head->size) - 1;
21 }
22
23 //In accordance with IOCTL declaraion section 1, the buffer shall support
       scaleing.
24 int buffer_resize(struct buffer * head, size_t size){
25        void * pointer;
26        mutex_lock(&head->mutex);
27
28        pointer = kmalloc(size*sizeof(*head->buffer),GFP_KERNEL);
29
30        if(!pointer) return rerror(-ENOMEM);
31        if(head-> wp == head-> rp){
32               head->wp = head->rp = pointer;
33
34        } else if(head-> wp > head-> rp){
35               size = head->wp - head->rp;
36               memcpy(pointer,head->rp,size);
37               head->wp = pointer + size;
38               head->rp = pointer;
39
40        } else {
41               size = (head->buffer + head->size) - head->rp;
42               memcpy(pointer,head->rp,size);
43               head->rp = pointer;
44               memcpy(head->rp + size, head->buffer, head->wp - head->buffer);
45               head->wp = (head->rp + size) + (head->wp - head->buffer);
46
47        }
48        head->buffer = pointer;
49        kfree(head->buffer);
50        mutex_unlock (&head->mutex);
51        return 0;
52 }
53
54 // initialise a new buffer of given size.
55 int buffer_init(struct buffer * head, size_t size){
```

```
56          void * pointer;
57          DEBUG_CODE(if(head->buffer) dprintf("To prevent memory leaks, the buffer
                pointer should be NULL."););
58
59          pointer = kmalloc(size * sizeof(*head->buffer),GFP_KERNEL);
60
61          if(!pointer) return rerror(-ENOMEM);
62
63          head->wp = head->rp = head->buffer = pointer;
64          head->size = size;
65
66          return 0;
67  }
68
69  // allocate memory for buffer
70  struct buffer * buffer(size_t size){
71    struct buffer * head = kmalloc(sizeof(*head), GFP_KERNEL);
72          buffer_init(head,size);
73    return head;
74  }
75
76  // free buffer by resizing to zero. This is possible because if
77  // krealloc's new_size is 0 and p is not a NULL pointer, the object pointed to is
        freed.
78  int buffer_free(struct buffer * head){
79          kfree(head->buffer);
80          head->buffer = NULL;
81          return 0;
82  }
83
84  size_t buffer_write(struct buffer * buf, char * seq, size_t size){
85          size_t new_size;
86          mutex_lock(&buf->mutex);
87          //dprintf("(%lu).wp : %lu" , (size_t)buf, (size_t)(buf->wp - buf->buffer))
                ;
88          if(buf->wp < buf->rp){
89                  new_size = min((size_t)(buf->wp - buf->rp) - 1, size);
90                  copy_from_user(buf->wp,seq,new_size);
91                  buf->wp += new_size;
92
93          }else{
94                  const size_t a = (buf->buffer + buf->size) - buf->wp;
95                  const size_t b = (buf->rp - buf->buffer) % buf->size;
96                  new_size = min(a,size);
97                  copy_from_user(buf->wp,seq,new_size);
98                  size -= new_size;
99                  if(0 < size){
100                         new_size = min(b,size);
101                         buf->wp = buf->buffer;
102                         copy_from_user(buf->wp,seq,new_size - 1);
103                 }
104                 buf->wp += new_size;
105         }
106         //dprintf("-> %lu\n" , (size_t)(buf->wp - buf->buffer));
107         mutex_unlock (&buf->mutex);
108         return new_size;
```

```
}
size_t buffer_read(struct buffer * buf, char * seq, size_t size){
        size_t new_size = 0;
        mutex_lock(&buf->mutex);
        //dprintf("(%lu).rp : %lu" , (size_t)buf, (size_t)(buf->rp - buf->buffer))
            ;
        if(buf->rp < buf->wp){
                new_size = min((size_t)(buf->wp - buf->rp), size);
                copy_to_user(seq,buf->rp,new_size);
                buf->rp += new_size;

        } else {
                const size_t a = (buf->buffer + buf->size) - buf->rp;
                const size_t b = buf->rp - buf->buffer;

                new_size = min(a,size);
                copy_to_user(seq,buf->rp,new_size);
                size -= new_size;
                if(0 < size){
                        new_size = min(b,new_size);
                        buf->rp = buf->buffer;
                        copy_to_user(seq,buf->rp,new_size);
                }
                buf->rp += new_size;

        }
        //dprintf("-> %lu\n" , buf->rp - buf->buffer);
        mutex_unlock (&buf->mutex);
        return new_size;
}
#endif /* end of include guard: DM510_BUFFER_H */
```

debug.h

```
#ifndef DM510_DEBUG_H
#define DM510_DEBUG_H
  #include <linux/slab.h>
  #ifdef DEBUG
  # define DEBUG_CODE(code) do {code} while (0);
  #else
  # define DEBUG_CODE(code) do {} while (0)
  #endif
  # define dprintf(...) DEBUG_CODE(printk("%s/%d : ",__FILE__,__LINE__);printk(
      __VA_ARGS__ );)
  # define rerror(errno,...) ({dprintf(KERN_ERR "Error (%d) : ",errno) ;
      DEBUG_CODE(printk(KERN_CONT __VA_ARGS__ );); errno;})
#endif /* end of include guard: DM510_DEBUG_H */
```