Morse Code Driver

Course Operation System

Autor	Benjamin Hadorn
E-Mail	b_hadorn@bluewin.ch
Ablage/Website	http://www.xatlantis.ch
Datum	03.12.08
Version	0.3

Index

1 Introduction

This project was made as a part of the Operation System course in autumn 2008 to meet the course goals. The project is written by Benjamin Hadorn (Student No: 01-257-450).

1.1 Goals of the project

Following section has been copied from [OS].

The goal of this project is to develop a module for the Linux kernel implementing a character oriented device that reads some input from the users and blinks the corresponding morse code (MO) on the keyboard LED's. An user writer process writes data into one end of the device, and the a kernel thread reads reads it from the other end and produces the morse code. The device itself contains a small FIFO buffer used to store temporarily the data being transferred: when the buffer is full, a writer trying to put new data into the device is put asleep; when the buffer is empty, the kernel thread reader is put asleep. To avoid concurrency problems, only one writer at time can send data to the device.

1.2 Requirements

Following requirements were given by the project evaluation team:

Nr.	Requirement	Met
1	The module must implement a char device for the Linux kernel.	yes
2	Only one writer at time can access the device for writing.	yes
3	The size of the buffer, and the size of chunk of data read by the reader, can be set upon loading (module parameters).	yes (only the buffer size can be set)
4	The length (in milliseconds) of both dot and dashes must be configurable as module parameter.	Yes (only one time can be set)
5	When there is no data to convert to morse, the kernel reader is put asleep and the writer is woken up.	yes
6	When there is no space left for writing into the buffer, the writer is put asleep and the reader is woken up.	yes

7	You must implement a `/proc` interface to expose the current status of the device (available data, device status, ...)	yes
8	You must provide some helper scripts to create the necessary device files in `/dev`	yes
9	You must provide a makefile to compile your module	yes
10	you must provide an user-space program (with source code, in whatever language you like) to test and show the functionalities and behaviors of your module (basic functionalities, concurrency management, sleeping, etc.)	yes

The requirements met are described in the implementation section.

1.3 Terminology

The kernel module for morse code is called **`keymorse`**.

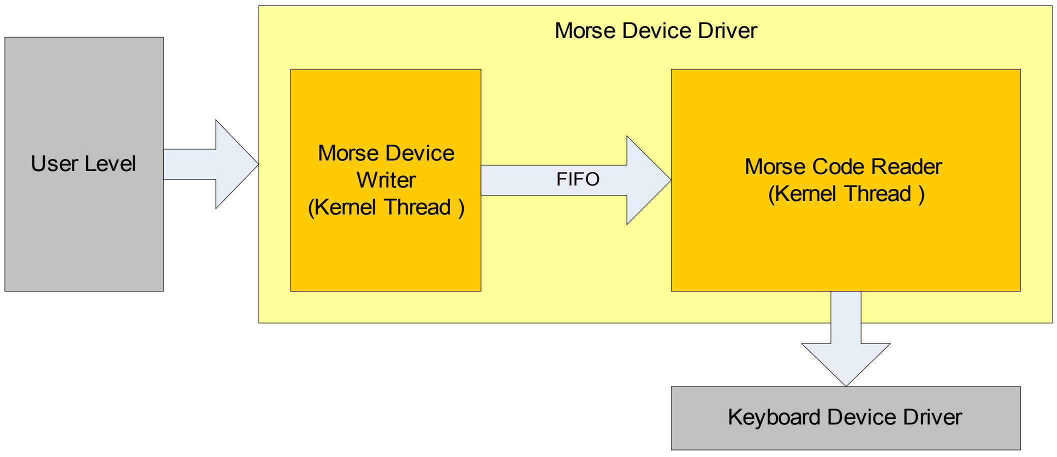
The device driver in `/dev` directory is called **`mo0`**.

2 Design

The module is designed using 2 threads, a writer thread and a reader thread processing the data and controlling the keyboard LED's.

The writer thread is provided by the kernel itself. Each write method is blocked until all data is written to the FIFO queue.

The reader thread is instantiated once loading the module to the kernel. Only one reader thread exists.


Morse Device Driver
User Level
Morse Device Writer (Kernel Thread)
FIFO
Morse Code Reader (Kernel Thread)
Keyboard Device Driver

Instead of communicating directly with the keyboard hardware the TTY-Keyboard driver is used to control the LED's.

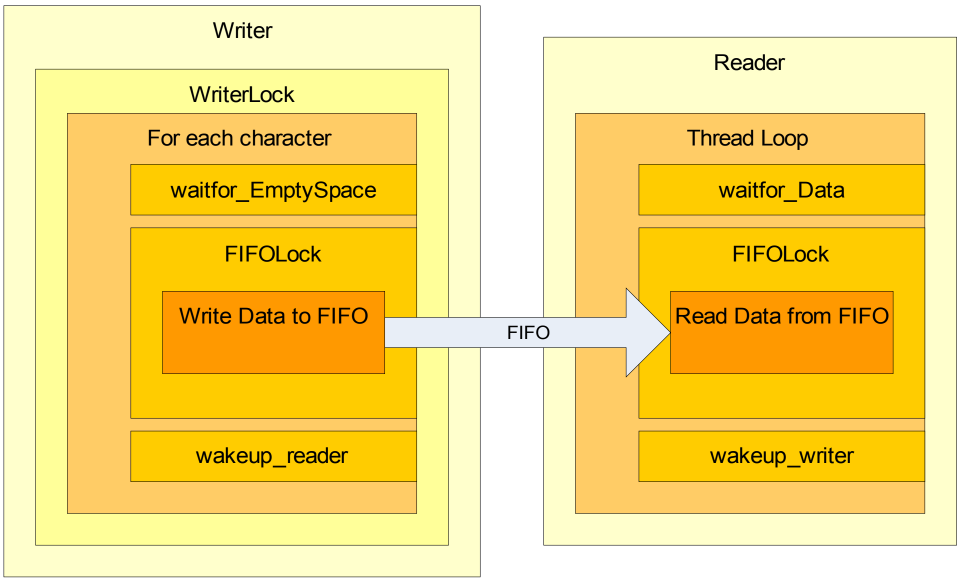
2.1 Lock mechanism

Requirement 2: To meet this requirement a `WriterLock` is used. Only one writer thread can access to the queue at the time. This ensures that no data gets mixed up.

Requirement 5: The reader turns in an endless loop. If no data is available in the FIFO queue the reader is put asleep and waked up if a writer puts some data into the queue. Once the reader has reawaken it acquires the FIFO-Lock and reads a character out of the FIFO. Now since some space in the FIFO is gained, the possibly waiting writer can be reawaken.

Requirement 6: The writer is trying to write each character separately into the FIFO (loop). If the FIFO is full the writer thread is put asleep (`waitfor_EmptySpace`) until the reader has processed some data. If there is enough space, the FIFO-Lock is acquired and the data is written to the FIFO. After that the waiting reader is woken up, since the reader is possibly waiting for data.

The sending of the character can be done outside of the FIFO-Lock.



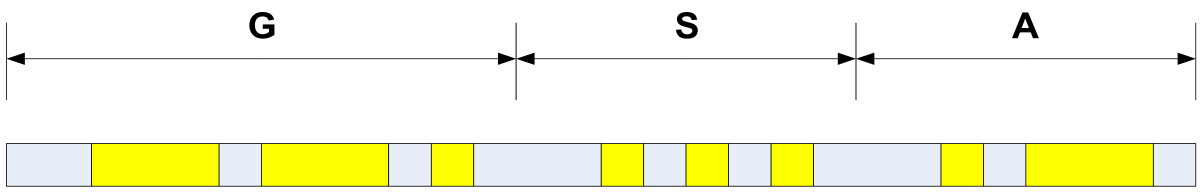
2.2 Morse code

The morse code is provided in a binary table. To get a good performance the character is used as index. The table entry contains a binary string containing the morse code.

To send the morse code only one time parameter is used.

- Each character is separated with 2 time slices.
- Each sign is separated by one time slice
- The dash has a length of 3 time slices.
- The dot has the length of one time slice.

To configure a correct morse code the only one time can be configured within this driver implementation (requirement 4).



3 Implementation

This section talks about the implementation of the `keymorse` kernel module. The techniques and some code fragments were taken from the Linux Driver book [DRV].

First the project structure is discussed. Then the key implementation of the module is explained.

3.1 Project Structure

The project contains 5 files:

- `keymorse.h` Header for definitions and forward declarations of functions and data structures
- `keymorse.cpp` Implementation of `keymorse` driver, including processing thread
- `Make` Make file to compile and build the driver
- `Install.sh` Installs the driver and creates the device `mo0`
- `Uninstall.sh` Uninstalls the driver and removes the device `mo0`

The `keymorse.cpp` is parted into three main sections, functions of the reader thread, functions of the writer thread and registration functions.

3.2 Registration functions

The registration functions are needed to have a valid kernel module.

Initialisation of the module

```
///Initialization of the module
static int initModule(void)
{
  ...
}

///uninitialisation of the module
static void exitModule(void)
{
  ...
}


//Hooks
module_init(initModule);
module_exit(exitModule);
```

Table 1: Initialization of the keymorse kernel module.

Before going into the two main functions `initModule()` and `exitModule()` lets looks at some very important structures and variables needed to register and manage the kernel module. The `keymorse` module is written as a character device. The complete device data is managed with the `TypeMorseDevice` data structure. It contains following elements:

• `cdev` :	character device structure (Requirement 1).
• `pThread`:	Reader thread
• `lockMutex`:	Mutex to protect the FIFO queue
• `lockWriter`:	Mutex to ensure that only one writer can send its data to the FIFO queue (Requirement 2).
• `pcFifoBuffer`:	FIFO Buffer to communicate with the reader thread
• `pTTYDriver`:	The keyboard device driver used to control the keyboard LED's.

TypeMorseDevice data structure

```
struct TypeMorseDevice
{
  struct cdev         cdev;
  struct task_struct* pThread;
  struct kfifo*       pFifo;
  struct semaphore    lockMutex;
  struct semaphore    lockWriter;
  char*               pcFifoBuffer;
  struct tty_driver*  pTTYDriver;
};

///Variable to handle the data
static struct TypeMorseDevice g_MorseDevice;
```

Table 2: Data structure of the keymorse device driver.

A very important structure is used to manage the operations the kernel has to call. The variable containing the file operations is called `g_Main_fops`. Only the functions for opening and releasing the file and for writing data are used.

File operation structures

```
///Variable to store file operations
static struct file_operations g_Main_fops =
  {
    .write   = main_write,
    .open    = main_open,
    .release = main_release,
    .owner   = THIS_MODULE,
  };
```

Table 3: Variable containing all relevant operations.

The device ID containing the major and minor device number is called `g_devID`.

Device ID

```
///ID of the device driver
static dev_t g_devID;
```

Table 4: Device ID containing the major and minor device number.

All these variables must be initialized when loading the module to the kernel. The first step after initializing some global variables is to register the module as a character device. If the major device number is not given, it will be dynamically allocated by the kernel.

Registration as character device

```
///Name of the device
#define KEYMORSE_NAME     "keymorse"

///Initialisation of the module
static int initModule(void)
{
  int iRetval = 0;
  int iIndex  = 0;
  ...
  //Registers the character device
  if (Mo_Major > 0)
  {
    g_devID = MKDEV(Mo_Major, Mo_Minor);
    iRetval = register_chrdev_region(g_devID, Mo_Dev_Numbers, KEYMORSE_NAME);
  }
  else
  {
    iRetval = alloc_chrdev_region(&g_devID, Mo_Minor,
                                  Mo_Dev_Numbers,
                                  KEYMORSE_NAME);
    Mo_Major = MAJOR(g_devID);
  }

  if (iRetval == 0)
  {
    //Registers the device methods and adds them to the kernel
    iRetval = setup_cdev(&g_MorseDevice, iIndex);
    ...
  }
  ...
  return iRetval;
}

///uninitialisation of the module
static void exitModule(void)
{
  if (Mo_Major > 0)
  {
    unregister_chrdev_region(g_devID, Mo_Dev_Numbers);
    release_cdev(&g_MorseDevice);
  }
  ...
}
```

Table 5: Shows the registration of a character device at the initialization and the unregistering at the exit function.

Next step is setting up the module functions for `open()`, `release()` and `write()`. This is done by the `setup_cdev()` function, witch wraps the complete setup procedure. The `release_cdev()` function is used to unregister the module functions.

Registration of the functions

```
/***************************************************************************/
/*! this function registers the main functions of the device
*/
/***************************************************************************/
static int setup_cdev(struct TypeMorseDevice* pData, int iIndex)
{
  int iDeviceNo = MKDEV(Mo_Major, Mo_Minor + iIndex);

  cdev_init(&pData->cdev, &g_Main_fops);
  pData->cdev.owner = THIS_MODULE;
  pData->cdev.ops   = &g_Main_fops;

  return cdev_add(&pData->cdev, iDeviceNo, 1);
}

/***************************************************************************/
/*! this function releases the device data
*/
/***************************************************************************/
static void release_cdev(struct TypeMorseDevice* pData)
{
  cdev_del(&pData->cdev);
}
```

Table 6: functions to register and unregister the functions of the kernel module.

The last step is to prepare the semaphores, FIFO and the thread for communication between the writer and reader thread. The waiting queues are initialized outside of the initialization function.

The FIFO queue must be allocated with size by power of 2. Therefore the buffer size variable is a number between 5 to 10. This gives a minimal size of 32 bytes and a maximal size of 1024 bytes.

The semaphores `lockMutex` and `lockWriter` are both set to 1, meaning that only one instance can acquire the lock.

After the initialization the reader thread can be stared. The thread method will be `run_thread()`.

When uninitializing the module it's important that first the thread is stopped, before freeing the FIFO memory. Otherwise the thread will access to freed memory.

Communication variables

```
///Queue for reader
DECLARE_WAIT_QUEUE_HEAD(queueReader);
///Queue for writer
DECLARE_WAIT_QUEUE_HEAD(queueWriter);
///Number of bytes not yet processed
static atomic_t g_atomUnprocessedData = ATOMIC_INIT(0);


///Initialisation of the module
static int initModule(void)
{
  int iFifoBufferSize = 0;
  ....
  //create fifo and create thread
  if (iRetval == 0)
  {
    iFifoBufferSize = 0x1 << BufferSize;
    atomic_set(&g_atomUnprocessedData, 0);
    g_MorseDevice.pcFifoBuffer = kmalloc(iFifoBufferSize*sizeof(char),
                                         GFP_KERNEL);
    g_MorseDevice.pFifo = kfifo_init(g_MorseDevice.pcFifoBuffer,
                                     iFifoBufferSize, GFP_KERNEL, NULL);
    g_MorseDevice.pTTYDriver = vc_cons[fg_console].d->vc_tty->driver;
    init_waitqueue_head(&queueReader);
    init_waitqueue_head(&queueWriter);
    sema_init(&g_MorseDevice.lockMutex, 1);
    sema_init(&g_MorseDevice.lockWriter, 1);
    //start the morse thread
    g_MorseDevice.pThread = kthread_run(run_morse, &g_MorseDevice,
                                        "morse_thread");
    if (IS_ERR(g_MorseDevice.pThread))
    {
      printk(KERN_WARNING "mo 0-2: Error creating the morse_thread_xx\n");
      iRetval = -EAGAIN;
    }
  }
}

///uninitialisation of the module
static void exitModule(void)
{
  ...
  if (g_MorseDevice.pThread != NULL)
  {
    kthread_stop(g_MorseDevice.pThread);
    g_MorseDevice.pThread = NULL;
  }

  if (g_MorseDevice.pcFifoBuffer != NULL)
  {
    kfree(g_MorseDevice.pcFifoBuffer);
    g_MorseDevice.pcFifoBuffer= NULL;
  }

}
```

Table 7: Initialization of the communication variables.

At the end lets look at the `/proc` interface, witch is needed to receive some status information about the data processing. To access the information of the queue the mutex of the FIFO queue must be acquired, because we need several instructions to calculate the available size of the queue.

For unprocessed data an atomic integer is suitable, since only reading is applicated.

/proc interface

```
///Name of the device
#define KEYMORSE_NAME     "keymorse"

///Initialisation of the module
static int initModule(void)
{
  ...
  //initialize the proc file
  create_proc_read_entry(KEYMORSE_NAME, 0, NULL, read_morsemem, NULL);
  ...
}

///uninitialisation of the module
static void exitModule(void)
{
  //Remove the proc entry
  remove_proc_entry(KEYMORSE_NAME, NULL);
  ...
}

/***************************************************************************/
/*! Reads the data for proc mem file
*/
/***************************************************************************/
/*static*/ int read_morsemem(char* pcBuffer, char** ppcStart, off_t iOffset,
int iCount, int* piEOF, void* pData)
{
  int iLen = 0;
  if (down_interruptible(&g_MorseDevice.lockMutex) == 0)
  {
    iLen += sprintf(pcBuffer + iLen, "Morse Code Driver [%d.%d]\n",
                    Mo_Major, Mo_Minor);
    iLen += sprintf(pcBuffer + iLen, "Blinktime: %d [ms]\n", BlinkTime);
    iLen += sprintf(pcBuffer + iLen, "TotalMem : %d [Bytes]\n",
                    g_MorseDevice.pFifo->size);
    iLen += sprintf(pcBuffer + iLen, "FreeMem  : %d [Bytes]\n",
                    g_MorseDevice.pFifo->size - g_MorseDevice.pFifo->in +
                    g_MorseDevice.pFifo->out);
    iLen += sprintf(pcBuffer + iLen, "Data to process: %d [Bytes]\n",
                    atomic_read(&g_atomUnprocessedData));
    up(&g_MorseDevice.lockMutex);
  }
  *piEOF = 1;
  return iLen;
}
```

Table 8: Implementation of the /proc interface

3.3 Writer thread

Before any data is written to the kernel module, the connection (file) must be opened. Therefore the function `main_open()` is called. What is done here is basically setting the device data pointer to the private data of the file pointer.

This is not very important for the general implementation, as long as not different device drivers are supported.

Whenever an `main_open()` is called there must be a `main_release()` at the end of the usage.

open and release

```
/**********************************************************************/
/*! Opens the device
*/
/**********************************************************************/
/*static*/ int main_open(struct inode* pNode, struct file* pFile)
{
  struct TypeMorseDevice* pDevice;
  pDevice = container_of(pNode->i_cdev, struct TypeMorseDevice, cdev);
  pFile->private_data = pDevice;
  return 0;
}

/**********************************************************************/
/*! Closes the device
*/
/**********************************************************************/
/*static*/ int main_release(struct inode* pNode, struct file* pFile)
{
  return 0;
}
```

Table 9: Implementation of the open and release methods used to access the driver

Writing the data to the FIFO is more involved. First the communication data structure of the module has to be setup. Then the data must be copied from user level memory into the kernel level memory, otherwise it will be not accessible.

The data buffer of kernel module is one character larger than the data buffer of the user level memory. The last item of the kernel level buffer is set to NULL in order to receive a regular NULL terminated string. This is useful for printing and debugging.

Copy data from user level memory

```
/***************************************************************************/
/*! Used to write data to the device
*/
/***************************************************************************/
/*static*/ ssize_t main_write(struct file* pFile, const char __user*
pcUserData, size_t iSize, loff_t* piPos)
{
  int iRetval = -EFAULT;
  struct TypeMorseDevice* pDevice =
                              (struct TypeMorseDevice*)pFile->private_data;

  if (pDevice != NULL && iSize > 0)
  {
    char* pcBuffer = kmalloc((iSize+1)*sizeof(char), GFP_KERNEL);
    if (pcBuffer != NULL)
    {
      memset(pcBuffer, 0, iSize*sizeof(char));
      if (copy_from_user(pcBuffer, pcUserData, iSize) == 0)
      {
        pcBuffer[iSize] = 0x00;
        ...
      }
      kfree(pcBuffer);
    }
  }
}
```

Table 10: Getting communication structure and coping the data from user level to kernel level memory

Before entering the writing section the size of unprocessed data is increased. Then the writer mutex is tried to acquire, witch is used to meet the **requirement 2** (only one writer at the time can access the FIFO queue). If the acquirement failed the counter of unprocessed data must be reduced by the amount just added before.

If the writer lock is acquired it's able to enter the `writeData()` function, where the writer thread can write all its data to the FIFO queue.

Acquire the writer lock

```
...
        atomic_add(iSize, &g_atomUnprocessedData);
        if (down_interruptible(&pDevice->lockWriter) == 0)
        {
          writeData(pDevice, pcBuffer, iSize);
          up(&pDevice->lockWriter);
          iRetval = iSize;
        } //down_interruptible(&pDevice->lockWriter) == 0
        else
        {
          //Could not aquire the write lock (interrupted)
          // -> decrement the unprocessed data size
          atomic_sub(iSize, &g_atomUnprocessedData);
        }
...
```

Table 11: Code to meet the requirement 2 that only one writer can access the FIFO queue

The `writeData()` function writes the data to the FIFO queue in 4 steps.

First step is to check if the FIFO queue is full or if there is still space available to send data. If not the writer thread is getting suspended and put in the waiting queue `queueWriter`.

The second step can be done if there is enough space available to send data. Therefore the writer tries to acquire the FIFO lock `lockMutex`. This ensures that only one thread (either writer thread, reader thread or status information reader) can access the FIFO queue.

The third step is to send the data to the queue and to release the FIFO lock afterwards. This step is done as long as there is free space and not all data is written to the FIFO.

The last step is to wake up the reader thread, witch might wait for data arriving from the FIFO queue.

Step 1 to 4 are repeated n times if the string contains n characters.

In case of aborting (interrupting) a writer, the number of unprocessed data must be set back by the amount left to send to the FIFO queue.

Acquire the writer lock

```
/***************************************************************************/
/*! writes the data to the fifo
*/
/***************************************************************************/
static void writeData(struct TypeMorseDevice* pDevice, char* pcBuffer,
                      int iSize)
{
  char acChar[2] = { 0, 0};
  int i          = 0;
  int iRetval    = 0;

  printk(KERN_NOTICE "mo 0-2: writeData() - ");
  while(i < iSize)
  {
    //1. Check if there is space available to write
    iRetval = wait_event_interruptible(queueWriter,
      (pDevice->pFifo->size - pDevice->pFifo->in + pDevice->pFifo->out > 0));

    //2. get fifo lock
    if (iRetval == 0 && down_interruptible(&pDevice->lockMutex) == 0)
    {
      //3. Send data and release fifo lock
      while(i < iSize &&
        pDevice->pFifo->size - pDevice->pFifo->in + pDevice->pFifo->out > 0)
      {
        acChar[0] = pcBuffer[i];
        __kfifo_put(pDevice->pFifo, acChar, 1);
        i++;
      }
      up(&pDevice->lockMutex);

      //4. wake the reader
      wake_up_interruptible(&queueReader);
    }

    //Error
    else
    {
      //Just to be sure the unprocessed data equals the
      // number of bytes when we entered the function
      atomic_sub(iSize -i, &g_atomUnprocessedData);

      i = iSize;
      printk(KERN_NOTICE "Proceeding interrupted (2)\n");
    }
  }
}
```

Table 12: Implementation of the writing function using FIFO queue

3.4 Reader thread

The reader thread runs in the function `run_morse()`. As parameter the module data structure is passed containing all mutexes and communication data objects. The thread turns inside a while loop. It's very important to check whether the thread must stop of not. Otherwise it is impossible to uninstall the kernel module and stopping the thread at all. The function `kthread_should_stop()` takes care of this.

Thread function of the reader

```
/***************************************************************************/
/*! This runs the morse thread, witch will convert and send all data
    to the keyboard
*/
/***************************************************************************/
static int run_morse(void* pData)
{
  struct TypeMorseDevice* pDev  = (struct TypeMorseDevice*)pData;
  int iRetval = 0;

  if (pDev != NULL)
  {
    printk(KERN_WARNING "mo 0-2: Thread started...\n");

    set_current_state(TASK_INTERRUPTIBLE);
    while(!kthread_should_stop())
    {
      //Reading data from the FIFO queue
      ...
    }
    __set_current_state(TASK_RUNNING);
    printk(KERN_WARNING "mo 0-2: Thread ended.\n");
  } //(pDev != NULL)

  return 0;
}
```

Table 13: Code of the thread function.

The reading is done in 5 steps.

The first step is checking whether there is data available to process or not. If no data is available the reader has to be suspended and put in the waiting queue `queueReader`. There is also the condition `kthread_should_stop()` witch must be checked, in case the thread must stop.

The second step is done when ever the reader wakes up (not interrupted and not stopped). Here it has to acquire the FIFO lock `lockMutex`.

The third step is reading data from the FIFO queue and releasing the `lockMutex` afterwards.

The fourth step is waking up the possibly waiting writer.

The last step is processing the data read before. This can be done outside of any critical section, since the data is copied into a local memory (variable). At the end the number of unprocessed data can be reduced.

Note that only one character is read at the time since sending it as morse code takes very long, it is not necessary to read in blocks to speedup the process.

5 steps reading data from the FIFO queue

```
//1. check if there is data to write
set_current_state(TASK_INTERRUPTIBLE);
iRetval = wait_event_interruptible(queueReader,
    (pDev->pFifo->in - pDev->pFifo->out > 0 || kthread_should_stop()));

//2. check error and try to get the fifo lock
set_current_state(TASK_INTERRUPTIBLE);
if (iRetval == 0 &&
    !kthread_should_stop() &&
    down_interruptible(&pDev->lockMutex) == 0)
{
  char acBuffer[1];
  __kfifo_get(pDev->pFifo, acBuffer, 1);

  //3. Release the lock
  up(&pDev->lockMutex);

  //4. wake up all writers
  wake_up_interruptible(&queueWriter);

  //5. send character as morse code to keyboard
  sendCode(pDev, acBuffer[0]);
  atomic_dec(&g_atomUnprocessedData);
}
```

Table 14: Reading and processing the data.

Finally the processing code witch is implemented within the `sendData()` function is explained.

3.4.1 Processing morse code

The morse code is implemented in a table of `TypeMorseItem` items. A morse code item contains one value for the morse code length (`cLength`) and one for the morse code itself (`cCode`).

The morse code is coded in bit format. Each bit set to 0 gives a short sign. Each bit set to

1 gives a long sign. The length is used to know how many signs must be send. For instance the number 0 (Zero coded as 0x00) can represent different morse codes depending on the given length. Length 1 is used to send an “e”, where as length 4 represents “h”. The whole morse code table is added in the appendix.

Morse item structure

```
struct TypeMorseItem
{
  char cLength;
  char cCode;
};

static struct TypeMorseItem g_acDataMap[59] = { ..}
```

Table 15: The morse code item.

The table contains 59 items in total, including characters A to Z, a to z and digits 0 to 9 as well as some additional characters for punctuation. In fact the lower case must be mapped to the upper case morse code (see below).

algorithm to get the morse code item

```
static void sendCode(struct TypeMorseDevice* pDev, char cASCII)
{
  struct TypeMorseItem Item;

  printk(KERN_NOTICE "mo 0-2:   Sending char code: %d\n", cASCII);

  //Make sure the border are met.
  if (cASCII < 32) { cASCII = 32; }
  if (cASCII > 90)
  {
    // Convert lower case to upper case
    if (cASCII >= 97 && cASCII <= 122)
    {
      cASCII -= 32;
    }
    else
    {
      cASCII = 32;
    }
  }
  Item = g_acDataMap[cASCII-32];
...
}
```

Table 16: The simple algorithm to evaluate the correct morse code item.

The processing algorithm must start with the LSB (least significant bit). The time intervals are set by the variable `BinkTime` (see parameters).

Before sending a letter 2x the interval is waited to separate the different letters. Long signs have a duration of 3x the interval as the short signs have only one time interval.

Between two signs one time interval is used to turn off the LED's.

sending the data

```
//Break from one character to the other one (+BlinkTime from the last send
// character)
schedule_timeout(2*BlinkTime);
for (i = Item.cLength-1; i >= 0; i--)
{
  iLong = (Item.cCode & (1 << i));
  (pDev->pTTYDriver->ioctl)(vc_cons[fg_console].d->vc_tty,
                             NULL,
                             KDSETLED,
                             LED_SCR | LED_NUM | LED_CAP);  //Capslock

  // Long term blinking
  if (iLong)
  {
    set_current_state(TASK_INTERRUPTIBLE);
    schedule_timeout(3*BlinkTime);
  }

  //Short term blinking
  else
  {
    set_current_state(TASK_INTERRUPTIBLE);
    schedule_timeout(BlinkTime);
  }

  //Turn the light off
  (pDev->pTTYDriver->ioctl)(vc_cons[fg_console].d->vc_tty,
                             NULL,
                             KDSETLED,
                             0xFF); //restore
  set_current_state(TASK_INTERRUPTIBLE);
  schedule_timeout(BlinkTime);
}
```

Table 17: Sending the morse code and shows the different time intervals.

4 Manual

This section describes how to build, install and use the `keymorse` kernel module. First a brief introduction how to build the module is given, including installing and uninstalling. Then some information about implemented interfaces is given such as module information and status information over the `/proc` interface.

4.1 Building the project

Commands used to build and install the `keymorse` kernel module:

- To build the project just call **`> make`**
- To clean the project call **`> make clean`**

Requirement 9: Installing and uninstalling script files are provided within the project.

- To install the the keymorse module (Make sure the root mode is used) **`# ./Install.sh`**
- To uninstall the keymorse module (Make sure the root mode is used) **`# ./Uninstall.sh`**

4.1.1 Driver Parameters

Requirement 3 and 4: Following parameters are provided:

Name	Derscription	Min	Max
BufferSize	**Requirement 3**: Size of the internal processing FIFO buffer. The buffer size must is a value of power to 2. Therefore the value 5 gives a buffer of 2^5 = 32 Bytes	5	10
BlinkTime	**Requirement 4**: Time in milliseconds of the short blinking morse code	10	--
Mo_Major	Major number of the device driver	0	255

Changing the parameters can be done only when the install script runs. For instance to change the buffer size and the blinking time following command can be used:

```
# ./Install.sh BufferSize=10 BlinkTime=75
```

The module uses now a FIFO of 1024 Bytes and the short time interval is set at 75 ms.

4.2 Information Interface

To receive general informations about the driver, such as author, version, parameters etc., call following command:

```
# modinfo ./keymorse.ko
```

Output of the modinfo

```
filename:       keymorse.ko
license:        GPL
author:         Benjamin Hadorn
description:    Morse code blink module for key board
vermagic:       2.6.18.8-0.10-default SMP mod_unload 586 REGPARM gcc-4.1
depends:
srcversion:     87CEACA82233F3B33498006
parm:           Mo_Major:Device major number (int)
parm:           BufferSize:Size of the buffer. If BufferSize=5 -> 2^5 = 32
Bytes (int)
parm:           BlinkTime:Time of blinking in [ms] (int)
```

Table 18: This listing shows the output of the modinfo querying the keymorse driver.

Requirement 7: To receive live time informations, call following command:

```
# cd /proc
# more keymorse
```

The output will show the status about the module:

- Blink time in [ms]
- Total memory of the FIFO queue
- Available memory (free memory) of the FIFO queue
- Data waiting to be processed. This includes all unprocessed data of the current writer and all data of writer waiting to send their data.

Status information

```
MorseCodeDriver [254.0]
Blinktime 10 [ms]
TotalMem  1024 [Bytes]
FreeMem   975 [Bytes]
DataMem   50 [Bytes]
```

Table 19: Using the /proc file system, the keymorse driver provides some status informations.

4.3 Sending data to the driver

To send data to the driver over command line use following command:

```
# echo “My data” > /dev/mo0
```

This sends the string “`My data`” to the device `mo0`.

5 Test Application

Before running the test the driver must be installed in the following manner before starting the test application:

```
# ./Install.sh BlinkTime=10 BufferSize=5
```

The test application is located in the `./test` directory. The application can be built with Make. The test uses some source files from Zeus-Framework [ZEUS] witch are located in the subdirectory `./test/zeusbase`.

There is a second `/proc` file for testing called `keymorse_test`. This file returns the processed data stream. Each time the data stream is read, the content is cleared (like a FIFO). Since the data stream size is fixed to 100 Bytes a test should not write more than 100 Bytes to the driver before reading the test stream.

5.1 Tests

The test is done within 3 steps. The first step shows and tests the requirement 7. It reads the current driver state and prints it to the screen.

Reading status information

```
TAutoPtr<TTextInputStream> ptrStream =
  new TTextInputStream(TString(L"/proc/keymorse"),
                       TTextInputStream::etISO_8859_1);
TString strText;
while (ptrStream->readLine(strText) == RET_NOERROR)
{
  ...
}
```

Table 20: Reading the status information using the /proc/keymorse file.

The 2nd step tests the requirements 1, 5 and 6. It sends a simple string to the device and waits for its completion. After the completion the processed data is read and compared with the data sent. The test includes the correct order of Bytes and the correct number of processed data.

Writing data to morse code driver

```
///Sending data
void sendDataToDriver(const TString& rData)
{
  TAutoPtr<TFileOutputStream> ptrStream =
    new TFileOutputStream(TString(L"/dev/mo0"), false);
  ptrStream->write(rData.c_str(NULL), rData.getSize());
}

//Reading processed data
void readProcessedData(TString& rData)
{
  rData = L"";
  TAutoPtr<TTextInputStream> ptrStream =
    new TTextInputStream(TString(L"/proc/keymorse_test"),
                         TTextInputStream::etISO_8859_1);
  TString strText;
  while (ptrStream->readLine(strText) == RET_NOERROR)
  {
    rData += strText;
  }
}
```

Table 21: Writing data to the driver is done using the file /dev/mo0.

The 3rd step tests “the requirement 2” that only one writer can write to the driver. Therefore the data from each writer must be processed separately. The test starts 3 threads each writing 32 Bytes to the device. The first thread is not blocked. The 2nd and 3rd thread are blocked. At the end the processed data must has a format `<Data1><Data2><Data3>`. This test ensures that the data from each thread is processed separately, each after the other.

6 Conclusion

The project gives a very good idea how Linux drivers are working (in general). It shows how flexible modern operation systems are. As a future part writing a similar driver for Windows would help to state the compare Linux with Windows.

This document might be used as reference even to implement other character devices. It gives a good overview of what is needed to implement a simple kernel module. For more sophisticated drivers the book [DRV] is strongly recommended as reference.

A Morse code table

The morse code source was taken from [MORSE].

ASCII code	Character	Morse code	Morse code length	Binary morse code
32	space		0	
33	!	00110	5	0x06
34	“	010010	6	
35	#		0	
36	$		0	
37	%		0	
38	&		0	
39	'	011110	6	0x1E
40	(		0	
41	)			
42	*			
43	+			
44	,	110011	6	0x33
45	-			
46	.	010101	6	0x16
47	/			
48	0	11111	5	0x1F
49	1	01111	5	0x0F
50	2	00111	5	0x07
51	3	00011	5	0x03
52	4	00001	5	0x01
53	5	00000	5	0x00
54	6	10000	5	0x10
55	7	11000	5	0x18
56	8	11100	5	0x1C
57	9	11110	5	0x1E
58	:	111000	5	0x38
59	;			
60	<			
61	“=”	10001	5	0x11
62	>			
63	?	001100	6	0x0C
64	@			
65	A	01	2	0x01

ASCII code	Character	Morse code	Morse code length	Binary morse code
66	B	1000	4	0x08
67	C	1010	4	0x0A
68	D	100	3	0x04
69	E	0	1	0x00
70	F	0010	4	0x02
71	G	110	3	0x06
72	H	0000	4	0x00
73	I	00	2	0x00
74	J	0111	4	0x07
75	K	101	3	0x05
76	L	0100	4	0x04
77	M	11	2	0x03
78	N	10	2	0x02
79	O	111	3	0x07
80	P	0110	4	0x06
81	Q	1101	4	0x0D
82	R	010	3	0x02
83	S	000	3	0x00
84	T	1	1	0x01
85	U	001	3	0x01
86	V	0001	4	0x01
87	W	011	3	0x03
88	X	1001	4	0x09
89	Y	1011	4	0x0B
90	Z	1100	4	0x0C

B Version History

Version	*Description*	*Datum*
0.1	Document created and first release of more code driver implemented	12.11.2008
0.2	Tests implemented	29.11.2008
0.3	Corrections and document redesign	03.12.2008

C Literature index

[OS]: Prof. Béat Hirsbrunner, Amos Brocco, Project Description, 2008, http://diuf.unifr.ch/pai/os/index.php?page=labs/index.php

[DRV]: Jonathan Corbet, Linux Device Drivers, 2005, http://lwn.net/Kernel/LDD3/

[ZEUS]: Benjamin Hadorn, Zeus-Framework, 2008, http://www.xatlantis.ch/zeusbase.html

[MORSE]: unknown, Morse Code, unknown, http://www.vidiot.com/Jericho/images/morse-tab1.jpg