# A Linux Device Driver for Direct Access to the Parallel Port

Kernel versions 2.0.x–2.4.x

Dirk Bächle
dl9obn@darc.de

August 18, 2009

# Contents

# 1 Introduction

## 1.1 The story behind PPProg

While trying to find decent tools for programming a PIC 16F84A microcontroller (or PICs in general) in-circuit under Linux I learned that there are a lot of solutions on the Internet. However, all programs—ranging from PROG84[1] to PICPRG[2]—need root access to work. This is because they read/write from/to the parallel port directly by means of `inb` and `outb` calls.
Basically, this works fine but I don't like to fiddle around as root longer than necessary. From my point of view this solution is slightly against a basic idea of Linux/Unix: Usually all hardware accesses are handled by appropriate device drivers and the user is supposed to trigger these accesses by system calls instead of directly accessing IO ports.
I finally stumbled upon PICPROG, a Linux device driver by Raffael Stocker. It enables reading, writing, bulk erase and much more. But having been developed back in 1999 it is pretty much outdated by now and does not support the PARPORT module. Additionally, it was tailored for a single PIC/adapter combination only.
This raised the question of how to provide access to the parallel port without needing root access, but being flexible enough to support a large variety of PIC/adapter combinations or parallel devices in general. After some thought, I came up with the idea of a driver for general-purpose parallel port programming. It should

- be able to set the data and control lines of the parallel port and to read back the data and status port,
- provide a way to generate command sequences like a *strobe* with—more or less—exact timings,
- be compliant with the PARPORT module and
- be general enough to support almost any special parallel device.

Additionally, a lot of `define` statements are used in the following that adapt PPPROG to the different kernel versions. The source you are currently looking at was designed to run from 2.0.x up to 2.4.x.
For the newer 2.6.x kernels, a separate driver was built that is available at `http://ppprog.sf.net` too.

## 1.2 Basic operation

PPPROG is compliant to the PARPORT module and lets you access the parallel port of your computer without needing root access (except for loading the driver itself at system startup).
At the start of the driver (via `modprobe` or `insmod`) you have to tell which parallel port to use for all following read/write accesses. Then you can use the functions `write` and `read` on an open handle to the appropriate device file (see 6.1 on p. 23) to read/write a byte from/to the data lines `D0`–`D7` of the port:

[1] Available at `www.picprg.com`
[2] Available at `http://www.bclane.com`

```
#include <fcntl.h>
#include <unistd.h>
#include "ppprog.h"

int main(void)
{
  char data;
  int file_desc = open("/dev/ppprog", O_RDONLY);

  if (file_desc < 0)
  return -1;

  /* Read data port lines */
  read(file_desc, &data, 1);
  printf("Data port: %X\n", data);

  close(file_desc);

  return 0;
}
```

However, the main target of this little driver are the so-called *command sequences*. A *command sequence* is a sequence of single writes to the data and control lines of the parallel port. Between these writes a delay can be specified, based on the `udelay` function. Since the *command sequence* function is executed by the driver, i.e. in kernel mode, the resulting timings should be quite exact. Two `ioctl` calls are available for this:

| | |
|---|---|
| `IOCTL_PPPROG_SEND` | Send a command sequence to the parallel port |
| `IOCTL_PPPROG_CHECK` | Check the current status of the parallel port |

As an example we want to generate a single *strobe* signal on data line D2, that goes HIGH for $50\mu s$ and then changes to LOW again for at least $10\mu s$.
The same program wrapper as above can be used, only the issued command changes (see also 6.2 on page 24):

```
#include <fcntl.h>
#include <unistd.h>
#include "ppprog.h"

int main(void)
{
  int data[5];
  int file_desc = open("/dev/ppprog", O_RDONLY);

  if (file_desc < 0)
  return -1;

  data[0] = 2;   /* Two state changes */
  data[1] = 0x4; /* Set D2 HIGH */
  data[2] = 50;  /* Wait 50us */
  data[3] = 0x0; /* Set D2 LOW */
  data[4] = 10;  /* Wait 10us */
```

```
  ioctl(file_desc, IOCTL_PPPROG_SEND, data);

  close(file_desc);

  return 0;
}
```

After initialization of the device file, the array of command sequences is set up. The first value in this array is the number $n$ of transitions that follow. In this example we have two state changes, so $n = 2$. Each single *state command* consists of two integers:

1. The data that should be applied to the parallel port. It is packed into a single integer, where

   | | |
   |---|---|
   | Bits 0–7 | Data port |
   | Bits 8–15 | Status port |
   | Bits 16–23 | Control port |

2. The number of $\mu s$ to wait, after the data has been applied.

Finally, the command sequence is sent to the attached device via the `ioctl` call. For a more realistic example of how to use the PPPROG driver, take a look at the library PICPROG[3]. It can program a 12F629 PIC in-circuit via a special parallel port adapter.

## 1.3 Disclaimer

This device driver is provided as is, without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the driver is with you.
Please, also regard the Gnu GPL disclaimer on page 26 and the full text in the file `COPYING`.
I am neither an expert kernel programmer, nor an experienced writer of device drivers. This work is a by-product of the first device driver that I ever created. So, despite its more or less promising appearance it still has to be regarded a quick hack.

## 1.4 What is Noweb?

This documentation was generated using NOWEB. NOWEB is maintained by Norman Ramsey and provides a tool for *Literate Programming*, an approach where the program and its documentation are written simultaneously. In doing so, the emphasis should be put on describing how the program works.
Derived from similar tools like WEB and CWEB, NOWEB uses the two programs `notangle` and `noweave` to extract the program and the documentation, respectively, from one source file.

[3] Also available at `http://ppprog.sf.net`

Source files consist of so called *chunks*. A chunk can contain a piece of text, or program code, or both. One can think of chunks as little pieces of code, that will be combined to the complete program by `notangle` no matter what language it is (C, C++, Pascal, Basic, Fortran, Lisp, Scheme, HTML, TeX, LaTeX, awk, perl, ...).
These code fragments are *woven* together, logically by the surrounding text, and physically by labels that get defined or referred to in a chunk. With this, one does not have to jump around in the source code for inserting a new variable, define or function. They are added wherever needed and this is what Noweb—and *Literate Programming* in general—is all about: Developing and presenting the idea behind the program instead of the mere code itself.
Documentation can be output in LaTeX, TeX and HTML. The chunks are numbered and referenced automatically and at the end of each block you find a list of the defined and used variables.
For further informations about Noweb, have a look at its homepage

`http://www.eecs.harvard.edu/~nr/noweb/`

or visit

`http://www.literateprogramming.com`

which discusses *Literate Programming* in general.

# 2 The device driver `ppprog.c`

While developing the device driver the recommendations in [1, chap. 5] and [2, chap. 4] are followed, respectively.
The basic structure of the device driver module looks like this:

4a ⟨*ppprog.c* 4a⟩≡ 9b ▷

```
  ⟨Header 5b⟩
  ⟨Include files 5d⟩
  ⟨Defines 6c⟩
  ⟨Global variables 7a⟩
  ⟨Device declarations 7b⟩
  ⟨Module declarations 18c⟩
```

## 2.1 The header file `ppprog.h`

The according header file `ppprog.h` has the following structure:

4b ⟨*ppprog.h* 4b⟩≡

```
  ⟨HF:Header 5a⟩

  #ifndef _PPPROG_H
  #define _PPPROG_H

  ⟨HF:Include files 15b⟩
  ⟨HF:Defines 15a⟩
```

```
#endif
```

Defines:
_PPPROG_H, never used.

5a ⟨*HF:Header* 5a⟩≡ (4b)

```
⟨GPL disclaimer 26⟩

/** \file ppprog.h
* Header file for the Linux Device Driver ppprog.c and all programs using the module.
⟨Common disclaimer 5c⟩
```

## 2.2 Header, includes and defines

Let us begin with the header of our device driver module. Although NOWEB is already used to document the source, a lot of DOXYGEN commands are added, such that a short documentation for quick reference can be generated.

5b ⟨*Header* 5b⟩≡ (4a) 8c ▷

```
⟨GPL disclaimer 26⟩

/** \file ppprog.c
* Linux Device Driver for direct access to the parallel port.
⟨Common disclaimer 5c⟩
```

The common disclaimer as included to all source and header files:

5c ⟨*Common disclaimer* 5c⟩≡ (5)

```
* \author Dirk Baechle, dl9obn@darc.de
* \version 1.0
* \date 2005-05-12
*/

/* This file was created automatically from the file ‘ppprog.nw’ by NOWEB.
* If you want to make changes, please edit the source ‘ppprog.nw’.
* A precompiled documentation can be found in ‘ppprog.pdf’ and ‘ppprog.ps’,
* respectively.
* Read it to understand why things are as they are. Thank you!
*/
```

Next are the include files, split into system header files and `ppprog.h`.

5d ⟨*Include files* 5d⟩≡ (4a)

```
⟨Linux includes 6a⟩

#include "ppprog.h"
```

A whole bunch of Linux headers is added to support the module with functions like `inb`, `outb` and `udelay`.

6a ⟨*Linux includes* 6a⟩≡ (5d) 9a ▷

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/ioport.h>
#include <asm/io.h>

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */
#include <linux/fs.h>
#include <linux/wrapper.h>

#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif

#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#include <linux/delay.h>
#else
#include <asm/delay.h>
#endif

⟨Module information 6b⟩
```

Defines:
`KERNEL_VERSION`, used in chunks 6b, 12–14, and 17–21.
`MODVERSIONS`, never used.

In newer kernels the macro `MODULE_LICENSE` needs to be set to `GPL`, this avoids a warning message while inserting the module.

6b ⟨*Module information* 6b⟩≡ (6a) 20b ▷

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dirk Baechle");
MODULE_DESCRIPTION("Driver for direct access to the parallel port");
MODULE_SUPPORTED_DEVICE("ppprog");
#endif
```

Uses `KERNEL_VERSION` 6a.

Now, the name of the device is defined as it appears in `/proc`. But most important is the definition of **success**...

6c ⟨*Defines* 6c⟩≡ (4a)

```
/** The value for success. */
#define SUCCESS                         0

/** The name of the device as it appears in /proc/devices. */
static char DEVICE_NAME[10] = "ppprog";
```

Defines:
`DEVICE_NAME`, used in chunks 10b, 11c, and 21.
`SUCCESS`, used in chunks 8a, 16a, and 19.

A flag is added to the global variables. It tells whether the device has already been opened or not.

7a ⟨*Global variables* 7a⟩≡ (4a) 7c ▷

```
/** Is the device open? 1 equals yes, 0 equals no. */
static int device_is_open = 0;
```

Defines:
`device_is_open`, used in chunks 8 and 12.

## 2.3 Device functions

Starting with the declarations for the device, the following functions should be supported:

7b ⟨*Device declarations* 7b⟩≡ (4a)
⟨*Device Open* 8a⟩
⟨*Device Release* 12⟩
⟨*Device Read* 13c⟩
⟨*Device Write* 14⟩
⟨*Device IOCtl* 16a⟩

### 2.3.1 Opening and closing the device file

The function `ppprog_open` is called whenever a process attempts to open the device file. First, it has to be ensured that the region of IO ports is still accessible and then they have to be reserved for use by the driver. But which ports are needed?
For a start, the IO address `0x378` is defined as the default base address that is about to be used.

7c ⟨*Global variables* 7a⟩+≡ (4a) ◁ 7a 10b ▷

```
/** Base address of the used parallel port (Default: 0x378) */
static int LptBase = 0x378;
```

Defines:
`LptBase`, used in chunks 10a, 11c, 13, 14, 17, 18b, and 20a.

ppprog_open checks whether the needed region of ports is still accessible and the device has not been opened yet. Then, all required ports are claimed until the device is released again (see ppprog_release).
The parallel port adapter needs to register itself with the Linux parport driver, which *guards* the parallel port in newer Linux versions. So, following the programming outlines in [3] and the source of the paride module in the kernel sources, ppprog_open tries to claim the parallel port from parport.

8a ⟨*Device Open* 8a⟩≡ (7b)

```
/** Attempts to open the device file.
* @param inode Pointer to the inode
* @param file Pointer to the device file
* @return 0 for success, else device is busy
*/
static int ppprog_open(struct inode *inode, struct file *file)
{
#if DEBUG
  printk(KERN_DEBUG "ppprog_open(%p, %p)\n", inode, file);
#endif

  ⟨check if device has not been opened yet 8b⟩
  ⟨register driver with parport 10a⟩
  ⟨claim parallel port regions 11c⟩

  device_is_open++;

  MOD_INC_USE_COUNT;

  return(SUCCESS);

}
```

Defines:
ppprog_open, used in chunk 18d.
Uses device_is_open 7a and SUCCESS 6c.

8b ⟨*check if device has not been opened yet* 8b⟩≡ (8a)

```
  /* Is the device open already? */
  if (device_is_open)
  {
#if DEBUG
    printk(KERN_DEBUG "Device PPPROG is already opened!\n");
#endif
    return(-EBUSY);
  }
```

Uses device_is_open 7a.

In order to register the parallel driver, some definitions from the parport header are needed.

8c ⟨*Header* 5b⟩+≡ (4a) ◁5b

```
/* Comment following define if ‘‘parport’’ */
/* driver should not be used. */
#define USE_PARPORT 1
```

Defines:
USE_PARPORT, used in chunks 9–11, 13, 17, and 18b.

9a ⟨*Linux includes* 6a⟩+≡ (5d) ◁6a

```
#ifdef USE_PARPORT
#include <linux/parport.h>
#endif
```

Uses USE_PARPORT 8c.

A new set of functions is added for interacting with the parport module:

9b ⟨*ppprog.c* 4a⟩+≡ ◁4a

```
#ifdef USE_PARPORT
⟨Parport support functions 9c⟩
#endif
```

Uses USE_PARPORT 8c.

9c ⟨*Parport support functions* 9c⟩≡ (9b)

```
⟨Attach port 9d⟩
⟨Detach port 9e⟩
```

ppprog_attach is called whenever the function parport_register_driver detects a new parallel port. Since the needed port is directly allocated in ppprog_open, there is nothing to do...

9d ⟨*Attach port* 9d⟩≡ (9c)

```
/** Attaches the found port to the device.
* @param port Pointer to struct for the found parallel port
*/
void ppprog_attach(struct parport *port)
{
  ;
}
```

Defines:
ppprog_attach, used in chunk 10b.

ppprog_detach is called whenever the function parport_register_driver detects that a parallel port vanished and therefore should be detached. Like for ppprog_attach, we do not really care...

9e ⟨*Detach port* 9e⟩≡ (9c)

```
/** Is called if a parallel port should be detached.
* @param port Pointer to struct for the parallel port
*/
void ppprog_detach(struct parport *port)
{
  ;
}
```

Defines:
ppprog_detach, used in chunk 10b.

10a ⟨*register driver with parport* 10a⟩≡ (8a)

```
#ifdef USE_PARPORT
⟨register parallel device 11a⟩
#else
  /* Is the region for the parallel port adapter still accessible? */
  if (check_region(LptBase, 3) != 0)
  {
#if DEBUG
    printk(KERN_DEBUG "IO ports for parallel port adapter are not accessible!\n");
#endif
    return(-EBUSY);
  }
#endif
```

Uses LptBase 7c and USE_PARPORT 8c.

A special struct is needed, storing the pointers to the functions ppprog_attach and ppprog_detach.

10b ⟨*Global variables* 7a⟩+≡ (4a) ◁7c 11b▷

```
#ifdef USE_PARPORT
/* Function prototypes */
void ppprog_attach(struct parport *);
void ppprog_detach(struct parport *);
/** Stores the pointers to the functions for attaching and detaching
* detected parallel ports. */
static struct parport_driver ppprog_driver = {
              DEVICE_NAME,
              ppprog_attach,
              ppprog_detach,
              NULL
};
#endif
```

Defines:
ppprog_driver, used in chunk 11a.
Uses DEVICE_NAME 6c, ppprog_attach 9d, ppprog_detach 9e, and USE_PARPORT 8c.

11a ⟨*register parallel device* 11a⟩≡ (10a)

```
if (parport_register_driver(&ppprog_driver) != 0)
{
#if DEBUG
    printk(KERN_DEBUG "PPPROG driver could not be registered with parport module!\n");
#endif
    return(-EBUSY);
}
```

Uses ppprog_driver 10b.

The pointer ppprog_port stores the allocated parallel port, which is dereferenced again in ppprog_release. ppprog_device keeps the pointer to the registered device. It is needed for claiming the ports and unregistering.

11b ⟨*Global variables* 7a⟩+≡ (4a) ◁10b 20c▷

```
#ifdef USE_PARPORT
/** Pointer to the struct of the allocated parallel port. */
struct parport *ppprog_port;
/** Pointer to the struct of the registered device, is needed
* for unregistering. */
struct pardevice *ppprog_device = 0;
#endif
```

Defines:
ppprog_device, used in chunks 11c, 13, 17, and 18b.
ppprog_port, used in chunk 11c.
Uses USE_PARPORT 8c.

If the claiming of ports via the parport module fails, the device is unregistered immediately.

11c ⟨*claim parallel port regions* 11c⟩≡ (8a)

```
#ifdef USE_PARPORT

  /* Get port with correct base number */
  ppprog_port = parport_find_base(LptBase);
  if (ppprog_port == NULL)
  {
#if DEBUG
    printk(KERN_DEBUG "Parallel IO port %X could not be found!\n", LptBase);
#endif
    return(-EBUSY);
  }

  ppprog_device = parport_register_device(ppprog_port,
                                          DEVICE_NAME,
                                          NULL,
                                          NULL,
                                          NULL,
```

```
                                              0,
                                              NULL);
  if (ppprog_device > 0)
  {
    if (parport_claim(ppprog_device) != 0)
    {
      parport_unregister_device(ppprog_device);
#if DEBUG
      printk(KERN_DEBUG "Parallel IO port %X is not accessible via parport driver!\n", L
#endif
      return(-EBUSY);
    }
  }
#else
  /* Claim port regions */
  request_region(LptBase, 3, DEVICE_NAME);
#endif
```

Uses DEVICE_NAME 6c, LptBase 7c, ppprog_device 11b, ppprog_port 11b, and USE_PARPORT 8c.

ppprog_release is called if a process closes the device file. It does not have a return value because it can not fail. It releases the region of ports needed for IO and unregisters the driver from the parport module. Afterwards, the device_is_open counter is decreased.

12 ⟨*Device Release* 12⟩≡ (7b)

```
/** Closes the device file.
* @param inode Pointer to the inode
* @param file Pointer to the device file
*/
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int ppprog_release(struct inode *inode, struct file *file)
#else
static void ppprog_release(struct inode *inode, struct file *file)
#endif
{

#if DEBUG
  printk(KERN_DEBUG "ppprog_release(%p, %p)\n", inode, file);
#endif

  ⟨release parallel port regions 13a⟩
  ⟨unregister parallel device driver 13b⟩

  /* Release device counter */
  device_is_open--;

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
  return(0);
#endif

}
```

Defines:
ppprog_release, used in chunk 18d.
Uses device_is_open 7a and KERNEL_VERSION 6a.

13a ⟨*release parallel port regions* 13a⟩≡ (12)

```
#ifdef USE_PARPORT
  parport_release(ppprog_device);
#else
  /* Release port regions */
  release_region(LptBase, 3);
#endif
```

Uses LptBase 7c, ppprog_device 11b, and USE_PARPORT 8c.

13b ⟨*unregister parallel device driver* 13b⟩≡ (12)

```
#ifdef USE_PARPORT
  parport_unregister_device(ppprog_device);
#endif
```

Uses ppprog_device 11b and USE_PARPORT 8c.

### 2.3.2 Reading and writing the data port

ppprog_read is called whenever a process, that has already opened the device file, attempts to read from it. Returning the status and control bits will be handled by means of ioctl functions. A read returns a single char in the given buffer, representing the data lines D0–D7.

13c ⟨*Device Read* 13c⟩≡ (7b)

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
/** Reads from the already opened device.
* @param file Pointer to the device file
* @param buffer Pointer to the buffer
* @param length Length of the buffer
* @param offset Offset to the file
* @return Number of bytes read
*/
static ssize_t ppprog_read(struct file *file, char *buffer, size_t length,
                           loff_t *offset)
#else
/** Reads from the already opened device.
* @param inode Pointer to inode
* @param file Pointer to the device file
```

```
* @param buffer Pointer to the buffer
* @param length Length of the buffer
* @return Number of bytes read
*/
static int ppprog_read(struct inode *inode, struct file *file, char *buffer,
                     int length)
#endif
{

#if DEBUG
  printk(KERN_DEBUG "ppprog_read(%p, %p, %p)\n", file, buffer, &length);
#endif

  put_user(inb(LptBase), buffer);

  return 1;
}
```

Defines:
 `ppprog_read`, used in chunk 18d.
 `ssize_t`, never used.
Uses `KERNEL_VERSION` 6a and `LptBase` 7c.

`ppprog_write` is called if somebody tries to write to the device file.
Again—just like in `ppprog_read`—a single `char` is processed and written to the data port of the parallel interface.

14 ⟨*Device Write* 14⟩≡ (7b)

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
/** Writes to the already opened device.
* @param file Pointer to the device file
* @param buffer Pointer to the buffer
* @param length Length of the buffer
* @param offset Offset to the file
* @return Number of bytes written
*/
static ssize_t ppprog_write(struct file *file, const char *buffer, size_t length,
                          loff_t *offset)
#else
/** Writes to the already opened device.
* @param inode Pointer to inode
* @param file Pointer to the device file
* @param buffer Pointer to the buffer
* @param length Length of the buffer
* @return Number of bytes written
*/
static int ppprog_write(struct inode *inode, struct file *file,
                      const char *buffer, int length)
#endif
{
```

```
#if DEBUG
  printk(KERN_DEBUG "ppprog_write (%p, %s, %d)", file, buffer, length);
#endif

  outb((char) (*buffer & 0xFF), LptBase);

  return 1;
}
```

Defines:
ppprog_write, used in chunk 18d.
ssize_t, never used.
Uses KERNEL_VERSION 6a and LptBase 7c.

### 2.3.3 Extended reading and writing via ioctl

The `ioctl` function is the very core of this little device driver. While developing it in the following chunks, the *logical actions* are used as defined in the header file `ppprog.h`:

| | |
|---|---|
| IOCTL_PPPROG_SEND | Send a command sequence to the parallel port |
| IOCTL_PPPROG_CHECK | Check the current status of the parallel port |

They have to be declared in a separate header file because they need to be known to both, the kernel module and the functions calling `ioctl` in the user program.
Our `ioctl` calls do not return a value, due to the `_IOR` keyword. As parameter all the functions get a pointer to `int`.
Additionally, the major device number and the name of the device file are defined. Please, note that `DEVICE_FILE_NAME` and `DEVICE_NAME` are something different although they have the same content.

15a ⟨*HF:Defines* 15a⟩≡ (4b) 16c ▷

```
/** The major device number */
#define DEVICE_MAJOR           219

/** The provided ioctl functions */
#define IOCTL_PPPROG_SEND      _IOR(DEVICE_MAJOR, 0, int *)
#define IOCTL_PPPROG_CHECK     _IOR(DEVICE_MAJOR, 1, int *)

/** The name of the device file */
#define DEVICE_FILE_NAME       "ppprog"
```

Defines:
DEVICE_FILE_NAME, never used.
DEVICE_MAJOR, used in chunk 21.
IOCTL_PPPROG_CHECK, used in chunks 18a and 25a.
IOCTL_PPPROG_SEND, used in chunks 16b, 24c, and 25a.

Since the `ioctl` call is used, `ioctl.h` needs to be included.

15b ⟨*HF:Include files* 15b⟩≡ (4b)

```
#include <linux/ioctl.h>
```

`ppprog_ioctl` is called whenever a process tries to do an `ioctl` on our device file. It has two extra parameters: the number of the called `ioctl` and the parameter given to the `ioctl` function.

16a ⟨*Device IOCtl* 16a⟩≡ (7b)

```
/** Handles the ioctl calls of the device driver.
* @param inode Pointer to the inode
* @param file Pointer to the file
* @param ioctl_num Number of the ioctl
* @param ioctl_param Parameter, i.e. pointer to int
* @return 0
*/
int ppprog_ioctl(struct inode *inode, struct file *file,
               unsigned int ioctl_num, unsigned long ioctl_param)
{
  int *temp, cnt, commands;
  int data = 0;

  switch (ioctl_num)
  {
    ⟨Case Send 16b⟩
    ⟨Case Check 18a⟩
  }

  return(SUCCESS);

}
```

Defines:
`ppprog_ioctl`, used in chunk 18d.
Uses `SUCCESS` 6c.

The first *Case* statement sends a sequence of commands.

16b ⟨*Case Send* 16b⟩≡ (16a)

```
case IOCTL_PPPROG_SEND:  temp = (int *) ioctl_param;
                         ⟨send command sequence 17⟩
                         break;
```

Uses `IOCTL_PPPROG_SEND` 15a.

Passing the number of signals to be asserted as a simple integer is quite risky. In order to prevent our system from freezing for some time—or forever if things go really, really bad—an upper bound is defined.

16c ⟨*HF:Defines* 15a⟩+≡ (4b) ◁15a

```
/** Maximum number of parallel port assertions within a
* single command sequence */
#define MAX_ASSERTIONS          64
```

Defines:
MAX_ASSERTIONS, used in chunk 17.

For every single command of the sequence, the *data* and *control* bits are extracted and asserted to the ports. Then, the specified delay is added.

17 ⟨*send command sequence* 17⟩≡ (16b)

```

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
get_user(commands, temp++);
#else
commands = get_user(temp++);
#endif

for (cnt = 0;
     ((cnt < MAX_ASSERTIONS) && (cnt < commands));
     cnt++)
{
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  get_user(data, temp++);
#else
  data = get_user(temp++);
#endif

#ifdef USE_PARPORT
  parport_write_data(ppprog_device->port, (data & 0xFF));
  parport_write_control(ppprog_device->port, ((data >> 16) & 0xFF));
#else
  outb((char) (data & 0xFF), LptBase);
  outb((char) ((data >> 16) & 0xFF), LptBase+2);
#endif

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  get_user(data, temp++);
#else
  data = get_user(temp++);
#endif

  if (data > 0)
    udelay(data);

}

```

Uses KERNEL_VERSION 6a, LptBase 7c, MAX_ASSERTIONS 16c, ppprog_device 11b, and USE_PARPORT 8c.

Reading data words is similar to writing. The pointer `ioctl_param` and the kernel function `put_user` are used to fill the first buffer location with data.

18a ⟨*Case Check* 18a⟩≡ (16a)

```
case IOCTL_PPPROG_CHECK: temp = (int *) ioctl_param;
                         ⟨check parallel port 18b⟩
                         break;
```

Uses `IOCTL_PPPROG_CHECK` 15a.

18b ⟨*check parallel port* 18b⟩≡ (18a)

```
#ifdef USE_PARPORT
data = (parport_read_control(ppprog_device->port) << 16);
data |= (parport_read_status(ppprog_device->port) << 8);
data |= parport_read_data(ppprog_device->port);
#else
data = (inb(LptBase+2) & 0xFF) << 16;
data |= (inb(LptBase+1) & 0xFF) << 8;
data |= inb(LptBase) & 0xFF;
#endif
put_user(data, temp);
```

Uses `LptBase` 7c, `ppprog_device` 11b, and `USE_PARPORT` 8c.

## 2.4 Module declarations

So much for the device driver. Now, only the module declarations are left:

18c ⟨*Module declarations* 18c⟩≡ (4a)
⟨*VFS Struct* 18d⟩
⟨*Init Module* 19⟩
⟨*Cleanup Module* 21b⟩

The struct `Fops` holds the functions to be called by the VFS (Virtual Filesystem Switch) if a process interacts with the created device.

18d ⟨*VFS Struct* 18d⟩≡ (18c)

```
/** Struct that holds the VFS functions for the device. */
static struct file_operations Fops =
{
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
  owner: THIS_MODULE,
  read: ppprog_read,        /* read */
  write: ppprog_write,      /* write */
  ioctl: ppprog_ioctl,      /* ioctl */
  open: ppprog_open,        /* open */
  release: ppprog_release   /* release */
#else
  NULL,                     /* seek */
```

```
  ppprog_read,              /* read */
  ppprog_write,             /* write */
  NULL,                     /* readdir */
  NULL,                     /* select */
  ppprog_ioctl,             /* ioctl */
  NULL,                     /* mmap */
  ppprog_open,              /* open */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,                     /* flush */
#endif
  ppprog_release            /* release */
#endif
};
```

Uses `KERNEL_VERSION` 6a, `ppprog_ioctl` 16a, `ppprog_open` 8a, `ppprog_read` 13c, `ppprog_release` 12 12, and `ppprog_write` 14.

While initializing the module, the main task is to register the device driver. The claiming of IO ports is done in `ppprog_open`. This enables other applications—i.e. the parport driver—to use the printer port for different tasks as long as the device `ppprog` is not opened, although the module may be loaded.

19 ⟨*Init Module* 19⟩≡ (18c)

```
/** Initializes the module by registering the device driver.
* @return 0 for success, < 0 for an error
*/
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,13)
static int ppprog_init(void)
#else
int init_module(void)
#endif
{
  int ret;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
  ⟨process module arguments 20a⟩
#endif
  ⟨try to register the device driver 21a⟩

  return(SUCCESS);
}
```

Defines:
`init_module`, used in chunk 21a.
`ppprog_init`, used in chunk 21b.
Uses `KERNEL_VERSION` 6a and `SUCCESS` 6c.

So far, the fixed LPT port base address `0x378` was used. It would be nice to be able to switch to different port addresses without having to recompile the driver.

Unlike in DOS or Windows there is no direct mapping from the port number to an appropriate IO base address for the parallel port under Linux. The user may specify the port number to be used via the module parameter `lp` (0-3). Then a default IO base address is used which is `0x278` for the `lp` number '1' and `0x378` else. If necessary, this address can be overwritten by the `iobase` parameter...

20a ⟨*process module arguments* 20a⟩≡ (19)

```
switch (lp)
{
  case 0: LptBase = 0x378; break;
  case 1: LptBase = 0x278; break;
  case 2: LptBase = 0x378; break;
  case 3: LptBase = 0x378; break;
}

if (iobase > 0)
  LptBase = iobase;
```

Uses `iobase` 20c, `lp` 20c, and `LptBase` 7c.

The module parameter descriptions are added to the info section at the start of the file...

20b ⟨*Module information* 6b⟩+≡ (6a) ◁6b

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
MODULE_PARM (lp, "i");
MODULE_PARM (iobase, "i");
MODULE_PARM_DESC( lp, "Parallel port that should be used (0-3)" );
MODULE_PARM_DESC( iobase, "Parallel port i/o base address. Overrides ‘lp’. (default: 0x3
#endif
```

Uses `iobase` 20c, `KERNEL_VERSION` 6a, and `lp` 20c.

The variables themselves are declared to be global.

20c ⟨*Global variables* 7a⟩+≡ (4a) ◁11b

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,1,0)
/** Number of the parallel port that should be used (0-3),
* where ‘‘0’’ refers to \c LPT1. */
static int lp = -1;
/** Base IO port address of the parallel port. Overrides \a lp,
* if necessary. */
static int iobase = -1;
#endif
```

Defines:
- `iobase`, used in chunk 20.
- `lp`, used in chunk 20.

Uses `KERNEL_VERSION` 6a.

For earlier kernels (< 2.4.0) the function `register_chrdev` is replaced by `module_register_chrdev` (see [1]).

21a ⟨*try to register the device driver* 21a⟩≡ (19)

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
  ret = register_chrdev(DEVICE_MAJOR, DEVICE_NAME, &Fops);
#else
  ret = module_register_chrdev(0, DEVICE_NAME, &Fops);
#endif

  /* Negative return values signify an error */
  if (ret < 0)
  {
    printk(KERN_ERR "PPPROG: <init_module> : Registering device failed with %d!", ret);
    return(ret);
  }

  printk(KERN_INFO "PPPROG: Device registered with major device number %d\n", DEVICE_MAJ
```

Uses `DEVICE_MAJOR` 15a, `DEVICE_NAME` 6c, `init_module` 19, and `KERNEL_VERSION` 6a.

The last thing to do is the cleanup. The device driver has to be unregistered for removing the kernel module.

21b ⟨*Cleanup Module* 21b⟩≡ (18c)

```
/** Cleanup by unregistering the appropriate file from /proc
*/
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,13)
static void ppprog_exit(void)
#else
void cleanup_module(void)
#endif
{
  int ret;

  ⟨unregister the device 21c⟩

}

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,13)
module_init(ppprog_init);
module_exit(ppprog_exit);
#endif
```

Defines:
`cleanup_module`, used in chunk 21c.
`ppprog_exit`, never used.
Uses `KERNEL_VERSION` 6a and `ppprog_init` 19.

For earlier kernels (< 2.4.0) the function `unregister_chrdev` is replaced by `module_unregister_chrdev` (see [1]).

21c ⟨*unregister the device* 21c⟩≡ (21b)

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
  ret = unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);
#else
  ret = module_unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);
#endif

  if (ret < 0)
  {
    printk(KERN_ERR "PPPROG: <cleanup_module> : Error %d while unregistering\n", ret);
  }
```

Uses `cleanup_module` 21b, `DEVICE_MAJOR` 15a, `DEVICE_NAME` 6c, and `KERNEL_VERSION` 6a.

That is it. The device driver module is now ready for use. But, how does this usage look like?

# 3 Additional defines

Depending on the flags the Linux kernel was compiled with, there are two other symbols that might have to be included to the device driver module.

- `__SMP__` — Symmetrical MultiProcessing. This has to be defined if the kernel was compiled to support symmetrical multiprocessing, even if just one CPU is used.
- `CONFIG_MODVERSIONS` — If `CONFIG_MODVERSIONS` was enabled in the kernel the symbol has to be defined when compiling the module and also `/usr/include/linux/modversions.h` has to be included.

One possible place to check how the kernel was built is `/usr/include/linux/config.h`.

# 4 The Makefile

Now the module can be compiled by using the prepared `Makefile` with the command

```
make
```

and then—changing to *root* mode—the new module and the created headers should be installed by

```
make install
```

Please, regard that the kernel sources have to be installed for compiling the module.
For older versions of the Linux kernel (< 2.4.0) the following `Makefile` can be used. The variable `USE_PARPORT` probably has to be undefined then.

22 ⟨*Makefile.old* 22⟩≡

```
TARGET=ppprog
CC=gcc
MODCFLAGS= -O2 -Wall -DMODULE -D__KERNEL__ -DLINUX

all: $(TARGET).o


$(TARGET).o: $(TARGET).c $(TARGET).h /usr/include/linux/version.h
	$(CC) $(MODCFLAGS) -c $(TARGET).c

$(TARGET).c: $(TARGET).nw
	notangle -L -R$(TARGET).c $(TARGET).nw > $(TARGET).c

$(TARGET).h: $(TARGET).nw
	notangle -L -R$(TARGET).h $(TARGET).nw > $(TARGET).h
```

# 5 Inserting and removing the module

Get *root* to insert and remove kernel modules. Then, the device driver module can be inserted by the command:

`modprobe ppprog`

If everything went fine and the module was properly inserted, it should appear in `/proc/modules`. This can be checked with either

`cat /proc/modules`

or

`lsmod`

Now, the device file (see 6.1) can communicate with the parallel port.
For removing the module again, one has to type:

`rmmod ppprog`

# 6 Talking to the device

## 6.1 Creating a device file

In order to talk to the device a *device file* has to be created. Being *root* one has to change the current directory to `/dev`. Then, the proper device file can be created by:

`mknod ppprog c 219 0`

## 6.2 Example program

Now, a quick example is given of how to use the `ioctl` functions. The task is to output a short message to a printer at the parallel port. Instead of simply calling `lpr` we do it all on our own, using PPProg only:

24a ⟨*ppprogtest.c* 24a⟩≡

```
#include <fcntl.h>
#include <unistd.h>
#include "ppprog.h"

⟨send a single character 25a⟩

int main(void)
{
  int file_desc, data[10];

  ⟨try to open device file 24b⟩
  ⟨init parallel port 24c⟩
  ⟨write message 25b⟩

  /* close device file */
  close(file_desc);

  return(0);
}
```

Defines:
`main`, never used.

The first step is to open the decive file, such that we can talk to our driver.

24b ⟨*try to open device file* 24b⟩≡ (24a)

```
/* try to open device file */
file_desc = open("/dev/ppprog", O_RDONLY);
if (file_desc < 0)
{
  printf("Can not open device file ppprog!\n");
  return(-1);
}
```

We initialize the parallel port to a predefined state by setting all lines to '0', except the lower three bits of the *command* word. The `STROBE` line (bit #0) and `INIT` (bit #2) are active low while the signal gets negated automatically. So for not activating them we have to set them to a '1' each. The `AUTOFEED` (bit #1) is enabled in order to convert single `CR` (carriage return) characters to a proper `CRLF` combination (carriage return, followed by a line feed).

24c ⟨*init parallel port* 24c⟩≡ (24a)

```
data[0] = 1;
data[1] = 0x070000;
data[2] = 0;
```

```
ioctl(file_desc, IOCTL_PPPROG_SEND, &data);
```

Uses IOCTL_PPPROG_SEND 15a.

Before we try to send the current character, a small loop waits until the printer is ready. Then, the command sequence is set up such that a strobe is generated while the data to be output is applied simultaneously. A single call of the `ioctl` function "fires" the sequence of port writes and prints the single letter.

25a ⟨*send a single character* 25a⟩≡ (24a)

```
send_char(int dfile, const char *sChar)
{
  int sData[5];

  /* Wait for printer */
  ioctl(dfile, IOCTL_PPPROG_CHECK, &sData);
  while ((*sData & 0x980000) == 0x0)
    ioctl(dfile, IOCTL_PPPROG_CHECK, &sData);

  /* Set data for command sequence */
  sData[0] = 2;
  sData[1] = 0x060000 | *sChar;
  sData[2] = 400;
  sData[3] = 0x070000 | *sChar;
  sData[4] = 0;
  /* Send command sequence */
  ioctl(dfile, IOCTL_PPPROG_SEND, &sData);

}
```

Uses IOCTL_PPPROG_CHECK 15a and IOCTL_PPPROG_SEND 15a.

Finally, the output text is constructed by sending the single characters, one after the other.

25b ⟨*write message* 25b⟩≡ (24a)

```
send_char(file_desc, "H");
send_char(file_desc, "e");
send_char(file_desc, "l");
send_char(file_desc, "l");
send_char(file_desc, "o");
send_char(file_desc, " ");
send_char(file_desc, "W");
send_char(file_desc, "o");
send_char(file_desc, "r");
send_char(file_desc, "l");
send_char(file_desc, "d");
send_char(file_desc, "!");
send_char(file_desc, "\n");
```

## GPL disclaimer

The GNU GPL disclaimer, as used by all source files...

26 ⟨*GPL disclaimer* 26⟩≡ (5)

```
/* PPProg - A Linux Device Driver for direct access to the
*           parallel port. (Kernel versions 2.0.x-2.4.x)
* Copyright (C) 2005 by Dirk Baechle (dl9obn@darc.de)
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; either version 2
* of the License, or (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public
* License along with this program; if not, write to the
*
* Free Software Foundation, Inc.
* 675 Mass Ave
* Cambridge
* MA 02139
* USA
*
*/
```

## List of code chunks

This list was generated automatically by NOWEB. The numeral is that of the first definition of the chunk.

# Index

This is a list of identifiers used, and where they appear. Underlined entries indicate the place of definition.

## References

[1] Ori Pomerantz. *Linux Kernel Module Programming Guide*, 1999. Version 1.1.0.

[2] Peter Jay Salzman and Ori Pomerantz. *Linux Kernel Module Programming Guide*, 2003. Version 2.4.0, (This document is available at `http://tldp.org/LDP/lkmpg/lkmpg.pdf`).

[3] Tim Waugh. *The Linux 2.4 Parallel Port Subsystem*, 2000.