A Linux Device Driver for the Parallel Port and the ISA Card Host Interface of the ER2

Dirk Bächle
TI6 (Distributed Systems)
Technical University Hamburg-Harburg

December 2, 2003

Contents

1 Introduction **1**
- 1.1 What is the ER2? 1
- 1.2 Why a device driver? 1
- 1.3 What is NOWEB? 2

2 The device driver `er2p.c` **2**
- 2.1 The header file `er2p.h` 2
- 2.2 Header, includes and defines 3
- 2.3 Device functions 5
- 2.4 How to access the parallel port interface 15
  - 2.4.1 Introduction 15
  - 2.4.2 Adapter functions 16
  - 2.4.3 Initializing the adapter 17
  - 2.4.4 Generating a STROBE 17
  - 2.4.5 Reset the ER2 18
  - 2.4.6 Writing data 18
  - 2.4.7 Writing a datum to the DATAWRITE register 18
  - 2.4.8 Writing an address to the DATAWRITE register 18
  - 2.4.9 Writing a datum/address to the LATCHWRITE register 18
  - 2.4.10 Generate an ER2 interrupt (IRQ2) 18
  - 2.4.11 Reading data 18
  - 2.4.12 Reading data from the DATAREAD register 19
  - 2.4.13 Reading data from the LATCHREAD register 19
- 2.5 IOCTL continued 19

3 Additional defines **37**

4 The Makefile **37**

5 Inserting and removing the module **38**

6 Talking to the device **38**
- 6.1 Creating a device file 38
- 6.2 Ensure correct settings for parallel port 39
- 6.3 Example program 39

1 Introduction

1.1 What is the ER2?

The ER2 is a parallel computer built in a modular fashion. On each of its 16 backplanes one can find 16 connectors for little plug-in processor boards, which are designed around an ADSP-2181. They further contain a FPGA, a crossbar switch, one connector to the backplane and one to additional modules. If fully assembled and by the aid of the *SHARC modules* the ER2 makes it up to a total of 512 processors (256 ADSP-2181, 256 SHARC-2106x).
By inserting a processor board to the backplane it is directly connected to its four neighbours in the north, west, south and east. Additionally, one can use the crossbar switches to establish links between dedicated nodes. This configuration can be done during runtime.
Thus, a large variety of graphs can be mapped to the network topology of the ER2. The DSPs offer a computing power of up to 12+16 GOPS, especially for digital signal processing applications.
For further information about the ER2 see also:

`http://www.tu-harburg.de/ti6/forschung/erii/`

1.2 Why a device driver?

The connection between the ER2 and a PC is established via an interface—also called host adapter. At the moment, two different host adapters exist. One is an ISA bus card that has to be inserted to the PC directly. The other host interface can simply be attached to the parallel port connector of the computer. Considering the functionality of the two interfaces, no differences exist between them. The parallel port interface is only much slower (150 KByte/s max.) than the ISA card (750 KByte/s max.), resulting from the fact that it can not handle full 16-bit words, but operates with 8 bit instead [1].
It has to be noted that the ISA card interface can only be used together with the 2181 modules. A write to a SHARC cluster via the *Fifth* PRS results in a reset of the ER2.
Both devices are controlled via writing to and reading from IO ports with `inb` and `outb` functions. Hence, programs in Linux that want to access the devices, i.e. get permissions for the appropriate ports, normally need to be run as *root*. This is, of course, somewhat awkward because every interested party should be enabled to develop applications for the ER2 without giving him/her the password of the superuser.
There exist several solutions for this problem, including *workarounds* like setting the *root execution* bit or using a daemon.
The normal way—and definitely the best—is to write a device driver. Being closely related to normal file handling (`open`, `read`, `write` ...) it does not only use a well defined interface to the kernel but also provides a useful and portable interface to applications. By hiding the necessary IO port accesses within the device driver one is able to write programs that will always work, regardless of which interface is used at the moment. If a new host adapter is ever developed, only the device driver has to be changed. All other source code stays exactly the same.

1.3 What is NOWEB?

This documentation was generated using NOWEB. NOWEB[1] is a tool for *literate programming*, an approach where the program and its documentation are written simultaneously. In doing so, the stress should lie on describing how the program works.
Derived from WEB[2] and CWEB[3], NOWEB uses the two programs `notangle` and `noweave` to extract the program and the documentation, respectively, from one source file.
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 into 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 right where the thought comes to the head 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 or HTML. The chunks are numbered automatically and at the end of each chunk you find a list of the newly defined and used variables.
For further informations about NOWEB, have a look at its homepage:

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

2 The device driver `er2p.c`

While developing the device driver the recommendations in [3, chap. 5] and [4, chap. 4] are followed, respectively.
The basic structure of the device driver module looks like this:

2a ⟨*er2p.c* 2a⟩≡ 8c▷
  ⟨*Header* 3c⟩
  ⟨*Include files* 4a⟩
  ⟨*Defines* 5a⟩
  ⟨*Global variables* 5b⟩
  ⟨*Device declarations* 5c⟩
  ⟨*Module declarations* 35a⟩

2.1 The header file `er2p.h`

The according header file `er2p.h` has the following structure:

[1] Written by Norman Ramsey
[2] Written by Donald E. Knuth
[3] Written by Silvio Levy and Donald E. Knuth

2b ⟨*er2p.h* 2b⟩≡
```
⟨HF:Header 3a⟩

#ifndef _ER2P_H
#define _ER2P_H

⟨HF:Include files 29a⟩
⟨HF:Defines 28b⟩

#endif
```

Defines:
`_ER2P_H`, never used.

3a ⟨*HF:Header* 3a⟩≡ (2b)

```
/* Name:                  er2p.h */
/* Author:                Dirk Baechle, TI6, TUHH */
/* Date:                  26.11.2003 */
/* Purpose:               Header file for the Linux Device Driver */
/*                        er2p.c and all programs using the module. */

⟨Disclaimer 3b⟩

/** \file er2p.h
Header file for the Linux Device Driver er2p.c and all programs using the module.
\author Dirk Baechle
\version 3.0
\date 26.11.2003
*/
```

Defines:
`file`, used in chunks 3, 7a, 11b, 13a, 14, 28b, 29b, 36c, 39, and 40a.
Uses `Device` 3c.

3b ⟨*Disclaimer* 3b⟩≡ (3 30a)

```
/* This file was created automatically from the file er2p.nw by NOWEB. */
/* If you want to make changes, please edit the source file er2p.nw. */
/* A full documentation is in er2p.tex, i.e. er2p.dvi and er2p.ps. */
/* Read it to understand why things are as they are. Thank you! */
```

Uses `file` 3a 30a.

2.2 Header, includes and defines

Let's 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 produced.

3c ⟨*Header* 3c⟩≡ (2a) 8a▷

```
/* Name:                  er2p.c */
/* Author:                Dirk Baechle, TI6, TUHH */
/* Date:                  26.11.2003 */
/* Purpose:               Linux Device Driver for the parallel */
/*                        port and the ISA card host interface of the ER2. */

⟨Disclaimer 3b⟩

/** \file er2p.c
Linux Device Driver for the parallel port and the ISA card host interface of the ER2.
\author Dirk Baechle
\version 3.0
\date 26.11.2003
*/
```

Defines:
  `Device`, used in chunks 3a, 7b, and 36b.
Uses `file` 3a 30a.

Next come the include files.

4a ⟨*Include files* 4a⟩≡ (2a)

```
⟨Linux includes 4b⟩

#include "er2gdef.h"
#include "er2p.h"
```

4b ⟨*Linux includes* 4b⟩≡ (4a) 8b▷

```
#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

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,4,0)
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dirk Baechle");
MODULE_DESCRIPTION("Driver for ER2 host adapters (ISA/PP)");
MODULE_SUPPORTED_DEVICE("er2p");
#endif
```

Defines:
`KERNEL_VERSION`, used in chunks 11b, 13a, 14, 32, 33a, and 35b.
`MODVERSIONS`, never used.

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

5a ⟨*Defines* 5a⟩≡ (2a) 6a▷

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

/** The name of the device as it appears in /proc/devices. */
static char DEVICE_NAME[10] = "er2p";
```

Defines:
`DEVICE_NAME`, used in chunks 9–11, 36b, and 37.
`SUCCESS`, used in chunks 7a, 29b, and 36a.

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

5b ⟨*Global variables* 5b⟩≡ (2a) 9c▷

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

Defines:
`device_is_open`, used in chunks 7 and 11b.

2.3 Device functions

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

5c ⟨*Device declarations* 5c⟩≡ (2a)
⟨*Device Open* 7a⟩
⟨*Device Release* 11b⟩
⟨*Device Read* 13a⟩
⟨*Device Write* 14⟩
⟨*Device IOCtl* 15a⟩

The function `er2p_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? Because both interfaces are combined into one device driver all ports have to be requested at once. Unfortunately, `ioctl` calls can only be sent to already opened device files. Otherwise, a special `ioctl` command could have been added in order to tell the device driver which interface should be used.
Some defines are introduced, in order to give the various port addresses a name:

6a ⟨*Defines* 5a⟩+≡ (2a) ◁5a 20a▷
⟨*Defines for the parallel port* 6b⟩
⟨*Defines for the ISA card* 6c⟩

6b ⟨*Defines for the parallel port* 6b⟩≡ (6a) 19▷

```
/* IO addresses of the parallel port */
#define PP_LPT1_DATA              0x378
#define PP_LPT1_STATUS            0x379
#define PP_LPT1_COMMAND           0x37A
```

Defines:
`PP_LPT1_COMMAND`, used in chunks 20–23.
`PP_LPT1_DATA`, used in chunks 9b, 10c, 12b, and 21–23.
`PP_LPT1_STATUS`, never used.

6c ⟨*Defines for the ISA card* 6c⟩≡ (6a) 25b▷

```
/* IO addresses of the ISA card */
#define IC_OUTP_LATCH             0x320
#define IC_STROBE_LOW             0x324
#define IC_STROBE_HIGH            0x326
#define IC_RESET_MODE             0x328
#define IC_ADDRESS_MODE           0x32A
#define IC_READ_MODE              0x32C
#define IC_WRITE_MODE             0x32E
```

Defines:
`IC_ADDRESS_MODE`, used in chunk 26d.
`IC_OUTP_LATCH`, used in chunks 7c, 11a, 12a, 26, and 27a.
`IC_READ_MODE`, used in chunk 27a.
`IC_RESET_MODE`, used in chunk 26b.
`IC_STROBE_HIGH`, used in chunk 25c.
`IC_STROBE_LOW`, used in chunk 25c.
`IC_WRITE_MODE`, used in chunk 26c.

`erp_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 `er2p_release`).
This can be done pretty straightforward for the ISA card interface. The parallel port adapter however, needs to register itself with the Linux `parport` driver, which *guards* the parallel port in newer kernel versions (> 2.4.x). So, following

the programming outlines in [5] and the source of the paride module in the kernel sources, er2p_open tries to claim the parallel port from parport.

7a ⟨*Device Open* 7a⟩≡ (5c)

```
/** 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 er2p_open(struct inode *inode, struct file *file)
{
#if DEBUG
  printk("er2p_open(%p, %p)\n", inode, file);
#endif

  ⟨check if device has not been opened yet 7b⟩
  ⟨check if ISA port regions are accessible 7c⟩
  ⟨register driver with parport 9b⟩
  ⟨claim parallel port regions 10c⟩
  ⟨claim ISA port regions 11a⟩

  device_is_open++;

  MOD_INC_USE_COUNT;

  return(SUCCESS);

}
```

Defines:
er2p_open, used in chunk 35b.
Uses device_is_open 5b, file 3a 30a, and SUCCESS 5a.

7b ⟨*check if device has not been opened yet* 7b⟩≡ (7a)

```
  /* Is the device open already? */
  if (device_is_open)
  {
#if DEBUG
    printk("Device ER2 is already opened!\n");
#endif
    return(-EBUSY);
  }
```

Uses Device 3c and device_is_open 5b.

7c ⟨*check if ISA port regions are accessible* 7c⟩≡ (7a)

```
  /* Is the port region of the ISA card still accessible? */
  if (check_region(IC_OUTP_LATCH, 16) != 0)
```

```
  {
#if DEBUG
    printk("IO ports for ISA card are not accessible!\n");
#endif
    return(-EBUSY);
  }
```

Uses IC_OUTP_LATCH 6c.

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

8a ⟨*Header* 3c⟩+≡ (2a) ◁3c

```
/* Comment following define if ''parport'' */
/* driver shouldn't be used. */
#define USE_PARPORT
```

Defines:
USE_PARPORT, used in chunks 8–10 and 12.

8b ⟨*Linux includes* 4b⟩+≡ (4a) ◁4b

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

Uses USE_PARPORT 8a.

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

8c ⟨*er2p.c* 2a⟩+≡ ◁2a

```
#ifdef USE_PARPORT
⟨Parport support functions 8d⟩
#endif
```

Uses USE_PARPORT 8a.

8d ⟨*Parport support functions* 8d⟩≡ (8c)

```
⟨Attach port 8e⟩
⟨Detach port 9a⟩
```

er2p_attach is called whenever the function parport_register_driver detects a new parallel port. Since the needed port is directly allocated in er2p_open, there is nothing to do...

8e ⟨*Attach port* 8e⟩≡ (8d)

```
/** Attaches the found port to the device.
* @param port Pointer to struct for the found parallel port
*/
void er2p_attach(struct parport *port)
{
```

```
    ;
  }
```

Defines:
er2p_attach, used in chunk 9c.

er2p_detach is called whenever the function parport_register_driver detects that a parallel port vanished and therefore should be detached. Again, a rather uninteresting case...

9a ⟨*Detach port* 9a⟩≡ (8d)

```
  /** Is called if a parallel port should be detached.
  * @param port Pointer to struct for the parallel port
  */
  void er2p_detach(struct parport *port)
  {
    ;
  }
```

Defines:
er2p_detach, used in chunk 9c.

9b ⟨*register driver with parport* 9b⟩≡ (7a)

```
  #ifdef USE_PARPORT
  ⟨register parallel device 10a⟩
  #else
    /* Is the region for the parallel port adapter still accessible? */
    if (check_region(PP_LPT1_DATA, 3) != 0)
    {
  #if DEBUG
      printk("IO ports for parallel port adapter are not accessible!\n");
  #endif
      return(-EBUSY);
    }
  #endif
```

Uses PP_LPT1_DATA 6b and USE_PARPORT 8a.

A special struct is needed, that stores the pointers to the functions er2p_attach and er2p_detach.

9c ⟨*Global variables* 5b⟩+≡ (2a) ◁5b 10b▷

```
  #ifdef USE_PARPORT
  /* Function prototypes */
  void er2p_attach(struct parport *);
  void er2p_detach(struct parport *);
  /** Stores the pointers to the functions for attaching and detaching
  detected parallel ports. */
  static struct parport_driver er2p_driver = {
              DEVICE_NAME,
              er2p_attach,
```

```
                er2p_detach,
                NULL
};
#endif
```

Defines:
er2p_driver, used in chunk 10a.
Uses DEVICE_NAME 5a, er2p_attach 8e, er2p_detach 9a, and USE_PARPORT 8a.

10a ⟨*register parallel device* 10a⟩≡ (9b)

```
if (parport_register_driver(&er2p_driver) != 0)
{
#if DEBUG
    printk("ER2P driver could not be registered with parport module!\n");
#endif
    return(-EBUSY);
}
```

Uses er2p_driver 9c.

The pointer `er2p_port` stores the allocated parallel port, which is dereferenced again in `er2p_release`. `er2p_device` keeps the pointer to the registered device. It is needed for claiming the ports and unregistering.

10b ⟨*Global variables* 5b⟩+≡ (2a) ◁9c 30b▷

```
#ifdef USE_PARPORT
/** Pointer to the struct of the allocated parallel port. */
struct parport *er2p_port;
/** Pointer to the struct of the registered device, is needed
for unregistering. */
struct pardevice *er2p_device = 0;
#endif
```

Defines:
er2p_device, used in chunks 10c and 12.
er2p_port, used in chunks 10c and 12b.
Uses USE_PARPORT 8a.

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

10c ⟨*claim parallel port regions* 10c⟩≡ (7a)

```
#ifdef USE_PARPORT

  /* Get port with correct base number */
  er2p_port = parport_find_base(PP_LPT1_DATA);
  if (er2p_port == NULL)
  {
#if DEBUG
    printk("Parallel IO port %X could not be found!\n", PP_LPT1_DATA);
```

```
#endif
    return(-EBUSY);
  }

  er2p_device = parport_register_device(er2p_port,
                                        DEVICE_NAME,
                                        NULL,
                                        NULL,
                                        NULL,
                                        0,
                                        NULL);
  if (er2p_device > 0)
  {
    if (parport_claim(er2p_device) != 0)
    {
      parport_unregister_device(er2p_device);
#if DEBUG
      printk("IO ports for parallel port adapter are not accessible via parport driver!\
#endif
      return(-EBUSY);
    }
  }
#else
  /* Claim port regions */
  request_region(PP_LPT1_DATA, 3, DEVICE_NAME);
#endif
```

Uses `DEVICE_NAME` 5a, `er2p_device` 10b, `er2p_port` 10b, `PP_LPT1_DATA` 6b, and `USE_PARPORT` 8a.

11a ⟨*claim ISA port regions* 11a⟩≡ (7a)

```
  /* Claim port regions */
  request_region(IC_OUTP_LATCH, 16, DEVICE_NAME);
```

Uses `DEVICE_NAME` 5a and `IC_OUTP_LATCH` 6c.

`er2p_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.

11b ⟨*Device Release* 11b⟩≡ (5c)

```
/** 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 er2p_release(struct inode *inode, struct file *file)
#else
static void er2p_release(struct inode *inode, struct file *file)
```

```
#endif
{

#if DEBUG
  printk("er2p_release(%p, %p)\n", inode, file);
#endif

  ⟨release ISA port regions 12a⟩
  ⟨release parallel port regions 12b⟩
  ⟨unregister parallel device driver 12c⟩

  /* Release device counter */
  device_is_open--;

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return(0);
#endif

}
```

Defines:
`er2p_release`, used in chunk 35b.
Uses `device_is_open` 5b, `file` 3a 30a, and `KERNEL_VERSION` 4b.

12a ⟨*release ISA port regions* 12a⟩≡ (11b)

```
  /* Release port regions */
  release_region(IC_OUTP_LATCH, 16);
```

Uses `IC_OUTP_LATCH` 6c.

12b ⟨*release parallel port regions* 12b⟩≡ (11b)

```
#ifdef USE_PARPORT
  parport_release(er2p_device);
  parport_put_port(er2p_port);
#else
  /* Release port regions */
  release_region(PP_LPT1_DATA, 3);
#endif
```

Uses `er2p_device` 10b, `er2p_port` 10b, `PP_LPT1_DATA` 6b, and `USE_PARPORT` 8a.

12c ⟨*unregister parallel device driver* 12c⟩≡ (11b)

```
#ifdef USE_PARPORT
  parport_unregister_device(er2p_device);
#endif
```

Uses `er2p_device` 10b and `USE_PARPORT` 8a.

`er2p_read` is called whenever a process, that has already opened the device file, attempts to read from it. Since all the communication between the PC and the interface shall be handled by means of `ioctl` functions, the given buffer is simply filled with zeros on every `read`.

13a ⟨*Device Read* 13a⟩≡ (5c)

```
#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 er2p_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 er2p_read(struct inode *inode, struct file *file, char *buffer,
                     int length)
#endif
{
  /* Number of bytes actually written into the buffer */
  int bytes_read = 0;

#if DEBUG
  printk("er2p_read(%p, %p, %p)\n", file, buffer, &length);
#endif

  ⟨fill buffer with zeros 13b⟩

#if DEBUG
  printk("Read %d bytes\n", bytes_read);
#endif

  return(bytes_read);
}
```

Defines:
  `er2p_read`, used in chunk 35b.
  `ssize_t`, never used.
Uses `file` 3a 30a and `KERNEL_VERSION` 4b.

13b ⟨*fill buffer with zeros* 13b⟩≡ (13a)

```
while (length)
{
  put_user(0x0, buffer++);

  bytes_read++;
  length--;

}
```

er2p_write is called if somebody tries to write to the device file.
Again—just like in er2p_read—it basically does nothing but simply returns the number of written bytes, in order to pretend everything is OK.

14 ⟨*Device Write* 14⟩≡ (5c)

```
#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 er2p_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 er2p_write(struct inode *inode, struct file *file,
                      const char *buffer, int length)
#endif
{

#if DEBUG
  printk("er2p_write (%p, %s, %d)", file, buffer, length);
#endif

  return(length);
}
```

Defines:
  er2p_write, used in chunk 35b.
  ssize_t, never used.
Uses file 3a 30a and KERNEL_VERSION 4b.

The `ioctl` function is the very core of this little device driver. It is split up into some auxiliary functions and the `ioctl` subroutine itself.

15a ⟨*Device IOCtl* 15a⟩≡ (5c)
  ⟨*Auxiliary IOCtl functions* 15b⟩
  ⟨*Device IOCtl function* 29b⟩

The auxiliary functions are responsible for talking to the parallel port (PP) or the ISA card (IC), i.e. the interface to the ER2, directly.

15b ⟨*Auxiliary IOCtl functions* 15b⟩≡ (15a)
  ⟨*PP:Auxiliary IOCtl functions* 15c⟩
  ⟨*IC:Auxiliary IOCtl functions* 25a⟩

The supported functions for the parallel port interface are:

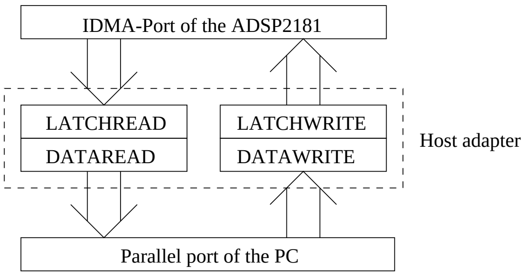
15c ⟨*PP:Auxiliary IOCtl functions* 15c⟩≡ (15b)
  ⟨*PP:Send Strobe Signal* 20b⟩
  ⟨*PP:Reset ER2* 20c⟩
  ⟨*PP:Write Latch* 21a⟩
  ⟨*PP:Write Byte* 21b⟩
  ⟨*PP:Write Address* 21c⟩
  ⟨*PP:Trigger ER2 Interrupt* 22a⟩
  ⟨*PP:Read Latch* 22b⟩
  ⟨*PP:Read Data* 23a⟩
  ⟨*PP:Write Word Data* 23b⟩
  ⟨*PP:Write Word Address* 24a⟩
  ⟨*PP:Read Word Data* 24b⟩

2.4 How to access the parallel port interface

Following, a few words about the parallel port host adapter and how to access it, i.e. how to trigger the provided functions:

2.4.1 Introduction

The interface—also called *host adapter* in the following—ensures the communication between a PC and the parallel computer ER2. It uses the parallel port of the PC and the IDMA port of the so called *root processor* on the ER2. The root processor is the ADSP2181 that is directly attached to the host adapter.
Although the interface needs the PPE mode for bidirectional data transfer to work correctly, it does not use it.
Since the IDMA port has 16 bit but the parallel port only 8 bit, always two cycles are needed for data transfer. The HIGH byte of a datum/address has to be stored in a special latch register of the host adapter.
The both directions *read* and *write* operate independently, i.e. there exist two latch registers (LATCHREAD and LATCHWRITE) for the HIGH byte and two registers (DATAREAD and DATAWRITE) for the LOW byte.



After the power reset both registers are zero and can only be changed by overwriting them with a different value.

2.4.2 Adapter functions

The host adapter provides 8 functions—actually, only seven—that can be specified via the pins $SEL0$–$SEL2$ of the parallel port command register.

No.	Description
0	Reset of the ER2
1	Write a datum to the DATAWRITE register
2	Write an address to the DATAWRITE register
3	Write a datum/address to the LATCHWRITE register
4	Trigger an ER2 interrupt (IRQ2)
5	Read a datum/address from the DATAREAD register
6	Read a datum/address from the LATCHREAD register
7	No function

The functions 1 and 2 transfer the whole 16-bit word to the ER2 immediately. While communicating with the interface the following port addresses of LPT1 are used:

Address	Port
0x378	Data port
0x379	Status port
0x37A	Command port

In order to trigger one of these functions the desired action has to be specified by setting the appropriate bits on the command port. Then a pulse has to be sent to the *Auto-Linefeed* pin to execute the command. This pulse is supposed to be *active low*, but since the *Auto-Linefeed* pin gets inverted, it has to be set to 1 (HIGH) first and then to 0 (LOW). This is called a *STROBE* from now on. The bits $SEL0$–$SEL2$ for specifying a command are not arranged in consecutive order at the parallel port. Hence, the bit patterns look a little bit confusing ... What follows is a short overview of the ports and the meaning of each single bit:

Status port (0x379)

Bit	Status
7	Busy
6	Acknowledge
5	Out of Paper
4	Select In
3	Error
2	IRQ (not)
1	Reserved
0	Reserved

Command port (0x37A)

Bit	Normal meaning	Meaning for interface	Con. Pin No.
7	Unused		
6	Unused		
5	Enable bi-directional port	1=Read, 0=Write	
4	Enable IRQ via ACK line		
3	Select printer	$\overline{SEL0}$	Pin 17
2	Initialize printer (Reset)	$SEL1$	Pin 16
1	Auto Linefeed	$\overline{STROBE}$	Pin 14
0	Printer-Strobe	$\overline{SEL2}$	Pin 1

The overlines for $SEL0$, $SEL2$ and $STROBE$ denote that these signals get inverted in the parallel port.
Now the triggering of the single functions is described step by step:

2.4.3 Initializing the adapter

This initialization sets the interface to *inactive mode*, i.e. the output drivers of the host adapter are in high impedance state.

- Write 0x24 to the command port

2.4.4 Generating a STROBE

The STROBE signal triggers the functions of the host adapter. Since only one bit (Auto-Linefeed) of the command port has to be toggled, while the others specify the command, we have to use logical ANDs and ORs.

- Combine the command with 0x02 by a logical OR and write it to the command port. The *Auto-Linefeed* bit is set to 1 (HIGH), gets inverted internally and at pin 14 of the parallel port connector we get a 0 (LOW).

- Combine the command with 0xFD by a logical AND and write it to the command port. The *Auto-Linefeed* gets 0 (LOW) and pin 14 1 (HIGH) again. The adapter detects the strobe signal and executes the specified function.

This is called *generating a command-strobe* from now on.

2.4.5 Reset the ER2

- Write the command 0x09 to the command port.
- Generate a command-strobe with 0x09.

2.4.6 Writing data

The order of the single commands while writing data may not be exchanged! Writing the datum/address to the data port first and then the command to the command port can give problems with newer motherboards. They seem to be very sensitive to slightly incorrect timings ...

2.4.7 Writing a datum to the DATAWRITE register

- Write the command 0x01 to the command port.
- Write the datum to the data port.
- Generate a command-strobe with 0x01.

2.4.8 Writing an address to the DATAWRITE register

- Write the command 0x0D to the command port.
- Write the address to the data port.
- Generate a command-strobe with 0x0D.

2.4.9 Writing a datum/address to the LATCHWRITE register

- Write the command 0x05 to the command port.
- Write the datum/address to the data port.
- Generate a command-strobe with 0x05.

2.4.10 Generate an ER2 interrupt (IRQ2)

- Write the command 0x08 to the command port.
- Generate a command-strobe with 0x08.

2.4.11 Reading data

While reading data the bits of the data port are only valid as long as the *Auto-Linefeed* bit is held at 1 (HIGH), i.e. pin 14 is 0 (LOW).
Thus, the *generate command-strobe* routine can not be used as before.

2.4.12 Reading data from the DATAREAD register

- Write the command 0x2C to the command port.
- Combine the command 0x2C with 0x02 by a logical OR and write it to the command port. This sets the *Auto-Linefeed* bit to 1 (HIGH), i.e. pin 14 to 0 (LOW). The output drivers of the interface switch to active mode and the valid datum appears at the data port.
- Read the datum from the data port.
- Combine the command 0x2C with 0xFD by a logical AND and write it to the command port. This sets the *Auto-Linefeed* bit to 0 (LOW) and pin 14 to 1 (HIGH) again. The output drivers of the interface switch back to the high impedance state.

2.4.13 Reading data from the LATCHREAD register

- Write the command 0x20 to the command port.
- Combine the command 0x20 with 0x02 by a logical OR and write it to the command port. This sets the *Auto-Linefeed* bit to 1 (HIGH), i.e. pin 14 to 0 (LOW). The output drivers of the interface switch to active mode and the valid datum appears at the data port.
- Read the datum from the data port.
- Combine the command 0x20 with 0xFD by a logical AND and write it to the command port. This sets the *Auto-Linefeed* bit to 0 (LOW) and pin 14 to 1 (HIGH) again. The output drivers of the interface switch back to the high impedance state.

2.5 IOCTL continued

The following defines are used for the implementation of the single adapter functions.

19 ⟨*Defines for the parallel port* 6b⟩+≡ (6a) ◁6b

```
/* Interface commands */
#define PP_IFC_STROBE_LOW       0x02
#define PP_IFC_STROBE_HIGH      0xFD
#define PP_STROBE_DELAY         10

#define PP_IFC_RESET_ER2        0x09
#define PP_IFC_WRITE_LATCH      0x05
#define PP_IFC_WRITE_DATA       0x01
#define PP_IFC_WRITE_ADDRESS    0x0D
#define PP_IFC_IRQ_ER2          0x08
#define PP_IFC_READ_LATCH       0x2C
#define PP_IFC_READ_DATA        0x20
```

Defines:
PP_IFC_IRQ_ER2, used in chunk 22a.
PP_IFC_READ_DATA, used in chunk 23a.
PP_IFC_READ_LATCH, used in chunk 22b.
PP_IFC_RESET_ER2, used in chunk 20c.
PP_IFC_STROBE_HIGH, used in chunks 20b, 22b, and 23a.
PP_IFC_STROBE_LOW, used in chunks 20b, 22b, and 23a.
PP_IFC_WRITE_ADDRESS, used in chunk 21c.
PP_IFC_WRITE_DATA, used in chunk 21b.
PP_IFC_WRITE_LATCH, used in chunk 21a.
PP_STROBE_DELAY, used in chunks 20b, 22b, and 23a.

A new data type called byte is defined.

20a ⟨*Defines* 5a⟩+≡ (2a) ◁6a

```
/* Defining a byte */
typedef unsigned char byte;
```

Defines:
byte, used in chunks 20, 21, 23, and 24.

This function sends a strobe signal, i.e. the interface will execute the function as specified by the signals on the command port.

20b ⟨*PP:Send Strobe Signal* 20b⟩≡ (15c)

```
/** Triggers one of the eight functions of the interface by sending
* a LOW pulse to the pin AUTO-LF (= STROBE) combined with the
* command byte.
* @param data The data byte for the command
*/
static void pp_ifc_send_strobe(byte data)
{

        outb((PP_IFC_STROBE_LOW | data), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);
        outb((PP_IFC_STROBE_HIGH & data), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);
}
```

Defines:
pp_ifc_send_strobe, used in chunks 20–22.
Uses byte 20a 22b 23a, PP_IFC_STROBE_HIGH 19, PP_IFC_STROBE_LOW 19, PP_LPT1_COMMAND 6b, and PP_STROBE_DELAY 19.

This function resets the ER2.

20c ⟨*PP:Reset ER2* 20c⟩≡ (15c)

```
/** Resets the ER2.
*/
static void pp_ifc_reset_er2(void)
{
        outb(PP_IFC_RESET_ER2, PP_LPT1_COMMAND);
        pp_ifc_send_strobe(PP_IFC_RESET_ER2);
```

```
}
```

Defines:
pp_ifc_reset_er2, used in chunk 31a.
Uses PP_IFC_RESET_ER2 19, pp_ifc_send_strobe 20b, and PP_LPT1_COMMAND 6b.

This function writes a byte (the HIGH byte of a datum or address) into the latch register of the interface.

21a ⟨*PP:Write Latch* 21a⟩≡ (15c)

```
/** Writes a byte (the HIGH byte of a datum or address)
* to the latch register of the interface.
* @param data The data byte
*/
static void pp_ifc_write_latch(byte data)
{
        outb(PP_IFC_WRITE_LATCH, PP_LPT1_COMMAND);
        outb(data, PP_LPT1_DATA);

        pp_ifc_send_strobe(PP_IFC_WRITE_LATCH);
}
```

Defines:
pp_ifc_write_latch, used in chunks 23b and 24a.
Uses byte 20a 22b 23a, pp_ifc_send_strobe 20b, PP_IFC_WRITE_LATCH 19, PP_LPT1_COMMAND 6b, and PP_LPT1_DATA 6b.

This function writes a byte (the LOW byte) into the data register of the interface.

21b ⟨*PP:Write Byte* 21b⟩≡ (15c)

```
/** Writes a byte (the LOW byte of a datum)
* to the register of the interface.
* @param data The data byte
*/
static void pp_ifc_write_data(byte data)
{
        outb(PP_IFC_WRITE_DATA, PP_LPT1_COMMAND);
        outb(data, PP_LPT1_DATA);

        pp_ifc_send_strobe(PP_IFC_WRITE_DATA);
}
```

Defines:
pp_ifc_write_data, used in chunk 23b.
Uses byte 20a 22b 23a, pp_ifc_send_strobe 20b, PP_IFC_WRITE_DATA 19, PP_LPT1_COMMAND 6b, and PP_LPT1_DATA 6b.

This function writes an address byte (the LOW byte of an address) into the register of the interface.

21c ⟨*PP:Write Address* 21c⟩≡ (15c)

```
/** Writes an address byte (the LOW byte of an address)
* to the register of the interface.
* @param address The address byte
*/
static void pp_ifc_write_address(byte address)
{
        outb(PP_IFC_WRITE_ADDRESS, PP_LPT1_COMMAND);
        outb(address, PP_LPT1_DATA);

        pp_ifc_send_strobe(PP_IFC_WRITE_ADDRESS);
}
```

Defines:
  `pp_ifc_write_address`, used in chunk 24a.
Uses `byte` 20a 22b 23a, `pp_ifc_send_strobe` 20b, `PP_IFC_WRITE_ADDRESS` 19,
  `PP_LPT1_COMMAND` 6b, and `PP_LPT1_DATA` 6b.

This function is used to send messages between the processors of the ER2 and triggers an interrupt (IRQ2) at the root processor.

22a ⟨*PP:Trigger ER2 Interrupt* 22a⟩≡ (15c)

```
/** Triggers an interrupt on the ER2.
*/
static void pp_ifc_irq_er2(void)
{
        outb(PP_IFC_IRQ_ER2, PP_LPT1_COMMAND);
        pp_ifc_send_strobe(PP_IFC_IRQ_ER2);
}
```

Defines:
  `pp_ifc_irq_er2`, used in chunk 31b.
Uses `PP_IFC_IRQ_ER2` 19, `pp_ifc_send_strobe` 20b, and `PP_LPT1_COMMAND` 6b.

This function reads a byte (the HIGH byte of a datum) from the latch register of the interface.

22b ⟨*PP:Read Latch* 22b⟩≡ (15c)

```
/** Reads a byte (the HIGH byte of a datum or address)
* from the latch register of the interface.
* @return The data byte
*/
static byte pp_ifc_read_latch(void)
{
        byte data;

        outb(PP_IFC_READ_LATCH, PP_LPT1_COMMAND);
        outb((PP_IFC_READ_LATCH | PP_IFC_STROBE_LOW), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);
        data=inb(PP_LPT1_DATA);
```

```
        outb((PP_IFC_READ_LATCH & PP_IFC_STROBE_HIGH), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);

        return(data);
}
```

Defines:
  byte, used in chunks 20, 21, 23, and 24.
Uses PP_IFC_READ_LATCH 19, PP_IFC_STROBE_HIGH 19, PP_IFC_STROBE_LOW 19, PP_LPT1_COMMAND 6b, PP_LPT1_DATA 6b, and PP_STROBE_DELAY 19.

This function reads a byte (the LOW byte of a datum) from the register of the interface.

23a ⟨*PP:Read Data* 23a⟩≡ (15c)

```
/** Reads a byte (the LOW byte of a datum)
* from the register of the interface.
* @return The data byte
*/
static byte pp_ifc_read_data(void)
{
        byte data;

        outb(PP_IFC_READ_DATA, PP_LPT1_COMMAND);
        outb((PP_IFC_READ_DATA | PP_IFC_STROBE_LOW), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);
        data=inb(PP_LPT1_DATA);
        outb((PP_IFC_READ_DATA & PP_IFC_STROBE_HIGH), PP_LPT1_COMMAND);
        udelay(PP_STROBE_DELAY);

        return(data);
}
```

Defines:
  byte, used in chunks 20, 21, 23, and 24.
Uses PP_IFC_READ_DATA 19, PP_IFC_STROBE_HIGH 19, PP_IFC_STROBE_LOW 19, PP_LPT1_COMMAND 6b, PP_LPT1_DATA 6b, and PP_STROBE_DELAY 19.

This function writes a 16-bit data word to the IDMA port of the root processor.

23b ⟨*PP:Write Word Data* 23b⟩≡ (15c)

```
/** Writes a 16-bit data word to the IDMA port of the
* ADSP-2181 processor.
* @param data The 16-bit data word
*/
static void pp_ifc_write_data_word(int data)
{
        byte data_byte;

        /* Write the HIGH byte to the latch register first */
        data_byte = (byte) (data >> 8);
```

```
        pp_ifc_write_latch(data_byte);

        /* Then write the LOW byte */
        data_byte = (byte) (data & 0x00FF);
        pp_ifc_write_data(data_byte);
}
```

Defines:
pp_ifc_write_data_word, used in chunk 32b.
Uses byte 20a 22b 23a, pp_ifc_write_data 21b, and pp_ifc_write_latch 21a.

This function writes a 16-bit address word to the IDMA port of the root processor.

24a ⟨*PP:Write Word Address* 24a⟩≡ (15c)

```
/** Writes a 16-bit address word to the IDMA port of the
* ADSP-2181 processor.
* @param address The 16-bit address word
*/
static void pp_ifc_write_address_word(int address)
{
        byte address_byte;

        /* Write the HIGH byte to the latch register first */
        address_byte = (byte) (address >> 8);
        pp_ifc_write_latch(address_byte);

        /* Then write the LOW byte */
        address_byte = (byte) (address & 0x00FF);
        pp_ifc_write_address(address_byte);
}
```

Defines:
pp_ifc_write_address_word, used in chunk 33b.
Uses byte 20a 22b 23a, pp_ifc_write_address 21c, and pp_ifc_write_latch 21a.

This function reads a 16-bit data word from the IDMA port of the root processor.

24b ⟨*PP:Read Word Data* 24b⟩≡ (15c)

```
/** Reads a 16-bit data word from the IDMA port of the
* ADSP-2181 processor.
* @return The 16-bit data word
*/
static int pp_ifc_read_data_word(void)
{
        int data;
        byte data_byte;

        /* Read the LOW byte first */
        data_byte = pp_ifc_read_data();
        data = (int) (data_byte);
```

```
        /* Then read the HIGH byte from the latch register */
        data_byte = pp_ifc_read_latch();
        data |= (int) (data_byte << 8);

        return(data);
}
```

Defines:
 pp_ifc_read_data_word, used in chunk 34a.
Uses byte 20a 22b 23a.

There are just a few auxiliary functions for the ISA card because it supports full 16-bit words. Accessing this interface is rather easy, using the hints given in [2, p. 3]. The following functions are required:

25a ⟨*IC:Auxiliary IOCtl functions* 25a⟩≡ (15b)
⟨*IC:Generate Strobe* 25c⟩
⟨*IC:Reset ER2* 26b⟩
⟨*IC:Write Word Data* 26c⟩
⟨*IC:Write Word Address* 26d⟩
⟨*IC:Read Word Data* 27a⟩
⟨*IC:Trigger ER2 Interrupt* 28a⟩

This function generates a strobe signal for the ISA card host adapter. A word is written to the STROBE_LOW port and then to the STROBE_HIGH port. The ISA card also needs a little delay between the single IO port accesses.

25b ⟨*Defines for the ISA card* 6c⟩+≡ (6a) ◁6c 26a▷

```
/** ISA card strobe delay */
#define ISA_STROBE_DELAY                10
```

Defines:
 ISA_STROBE_DELAY, used in chunks 25 and 26.

25c ⟨*IC:Generate Strobe* 25c⟩≡ (25a)

```
/** Generates a STROBE for the ISA card host adapter.
*/
static void ic_ifc_generate_strobe(void)
{
        udelay(ISA_STROBE_DELAY);
        outw(0, IC_STROBE_LOW);
        udelay(ISA_STROBE_DELAY);
        outw(0, IC_STROBE_HIGH);
        udelay(ISA_STROBE_DELAY);
}
```

Defines:
 ic_ifc_generate_strobe, used in chunks 26 and 27a.
Uses IC_STROBE_HIGH 6c, IC_STROBE_LOW 6c, and ISA_STROBE_DELAY 25b.

This function resets the ER2. A word is written to the RESET_MODE port, followed by a *strobe*.

26a ⟨*Defines for the ISA card* 6c⟩+≡ (6a) ◁25b 27b▷

```
/** ISA card reset delay */
#define IC_RESET_DELAY          10000
```

Defines:
IC_RESET_DELAY, used in chunk 26b.

26b ⟨*IC:Reset ER2* 26b⟩≡ (25a)

```
/** Resets the ER2.
*/
static void ic_ifc_reset_er2(void)
{
        outw(0, IC_RESET_MODE);
        udelay(IC_RESET_DELAY);
        ic_ifc_generate_strobe();
}
```

Defines:
ic_ifc_reset_er2, used in chunk 31a.
Uses ic_ifc_generate_strobe 25c, IC_RESET_DELAY 26a, and IC_RESET_MODE 6c.

This function writes a 16-bit data word to the IDMA port of the root processor. First, the ISA card is set to WRITE_MODE. Then, the data is put to the port OUTP_LATCH, followed by a *strobe*.

26c ⟨*IC:Write Word Data* 26c⟩≡ (25a)

```
/** Writes a 16-bit data word to the IDMA port of the
* ADSP-2181 processor.
* @param data The 16-bit data word
*/
static void ic_ifc_write_data_word(int data)
{
        outw(0, IC_WRITE_MODE);
        udelay(ISA_STROBE_DELAY);
        outw(data, IC_OUTP_LATCH);
        ic_ifc_generate_strobe();
}
```

Defines:
ic_ifc_write_data_word, used in chunk 32c.
Uses ic_ifc_generate_strobe 25c, IC_OUTP_LATCH 6c, IC_WRITE_MODE 6c, and ISA_STROBE_DELAY 25b.

Similar to pp_ifc_write_data_word a 16-bit address is written. First, the ISA card is set to ADDRESS_MODE. Then, the address is put to the port OUTP_LATCH and a *strobe* is generated.

26d ⟨*IC:Write Word Address* 26d⟩≡ (25a)

```
/** Writes a 16-bit address word to the IDMA port of the
* ADSP-2181 processor.
* @param address The 16-bit address word
*/
static void ic_ifc_write_address_word(int address)
{
        outw(0, IC_ADDRESS_MODE);
        udelay(ISA_STROBE_DELAY);
        outw(address, IC_OUTP_LATCH);
        ic_ifc_generate_strobe();
}
```

Defines:
  `ic_ifc_write_address_word`, used in chunks 28a and 33b.
Uses `IC_ADDRESS_MODE` 6c, `ic_ifc_generate_strobe` 25c, `IC_OUTP_LATCH` 6c, and `ISA_STROBE_DELAY` 25b.

This function reads a 16-bit data word from the IDMA port of the root processor. First, the ISA card is set to `READ_MODE`. Then, after generating a *strobe* the data can be read from the `OUTP_LATCH` port.

27a ⟨*IC:Read Word Data* 27a⟩≡ (25a)

```
/** Reads a 16-bit data word from the IDMA port of the
* ADSP-2181 processor.
* @return The 16-bit data word
*/
static int ic_ifc_read_data_word(void)
{
        int data;

        outw(0, IC_READ_MODE);
        ic_ifc_generate_strobe();
        data = inw(IC_OUTP_LATCH);

        return(data);
}
```

Defines:
  `ic_ifc_read_data_word`, used in chunk 34b.
Uses `ic_ifc_generate_strobe` 25c, `IC_OUTP_LATCH` 6c, and `IC_READ_MODE` 6c.

An ER2 interrupt (IRQ2) can be triggered via the ISA card host adapter by setting the address to `0x8000`. First, a define is added for this value:

27b ⟨*Defines for the ISA card* 6c⟩+≡ (6a) ◁26a

```
/** Interrupt for sending messages */
#define IRQ             0x8000
```

Defines:
  `IRQ`, used in chunk 28a.

28a ⟨*IC:Trigger ER2 Interrupt* 28a⟩≡ (25a)

```
/** Triggers an interrupt on the ER2.
*/
static void ic_ifc_irq_er2(void)
{
        ic_ifc_write_address_word(IRQ);
}
```

Defines:
  `ic_ifc_irq_er2`, used in chunk 31b.
Uses `ic_ifc_write_address_word` 26d and `IRQ` 27b.

While developing the `ioctl` function in the following chunks, the *logical actions* are used as defined in the header file `er2p.h`:

`IOCTL_ER2_RESET`	Reset the ER2
`IOCTL_ER2_IRQ`	Trigger an ER2 interrupt
`IOCTL_ER2_WRITE_WORDS`	Write 16-bit data
`IOCTL_ER2_WRITE_ADDRESS`	Write a 16-bit address word
`IOCTL_ER2_READ_WORDS`	Read 16-bit data
`IOCTL_ER2_SET_LENGTH`	Sets the number of words to read/write
`IOCTL_ER2_SET_INTERFACE`	Sets the used interface

They have to be declarated in a separate header file because they need to be known both to the kernel module and the functions calling `ioctl` in `pc_er2.c`. 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.

28b ⟨*HF:Defines* 28b⟩≡ (2b)

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

/** The provided ioctl functions */
#define IOCTL_ER2_RESET         _IOR(DEVICE_MAJOR, 0, int *)
#define IOCTL_ER2_IRQ           _IOR(DEVICE_MAJOR, 1, int *)
#define IOCTL_ER2_WRITE_WORDS   _IOR(DEVICE_MAJOR, 2, int *)
#define IOCTL_ER2_WRITE_ADDRESS _IOR(DEVICE_MAJOR, 3, int *)
#define IOCTL_ER2_READ_WORDS    _IOR(DEVICE_MAJOR, 4, int *)
#define IOCTL_ER2_SET_LENGTH    _IOR(DEVICE_MAJOR, 5, int *)
#define IOCTL_ER2_SET_INTERFACE _IOR(DEVICE_MAJOR, 6, int *)

/** The name of the device file */
#define DEVICE_FILE_NAME        "er2p"
```

Defines:
  `DEVICE_FILE_NAME`, never used.
  `DEVICE_MAJOR`, used in chunks 36b and 37.
  `IOCTL_ER2_IRQ`, used in chunk 31b.
  `IOCTL_ER2_READ_WORDS`, used in chunks 33c, 40, and 41b.
  `IOCTL_ER2_RESET`, used in chunks 31a and 40b.

IOCTL_ER2_SET_INTERFACE, used in chunk 34d.
IOCTL_ER2_SET_LENGTH, used in chunk 34c.
IOCTL_ER2_WRITE_ADDRESS, used in chunks 33a, 40, and 41.
IOCTL_ER2_WRITE_WORDS, used in chunks 32a, 40c, and 41a.
Uses `file` 3a 30a.

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

29a ⟨*HF:Include files* 29a⟩≡ (2b)

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

`er2p_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.
If the `ioctl` is write or read/write—meaning output is returned to the calling process—, the `ioctl` call returns the output of this function.
Here, no function will return a value. As parameter all the functions get a pointer to `int`.

29b ⟨*Device IOCtl function* 29b⟩≡ (15a)

```
/** 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 er2p_ioctl(struct inode *inode, struct file *file,
               unsigned int ioctl_num, unsigned long ioctl_param)
{
  int *temp, data;

  switch (ioctl_num)
  {
    ⟨Case Statement 29c⟩
  }

  return(SUCCESS);

}
```

Defines:
`er2p_ioctl`, used in chunk 35b.
Uses `file` 3a 30a and `SUCCESS` 5a.

29c ⟨*Case Statement* 29c⟩≡ (29b)

⟨*Case Reset* 31a⟩
⟨*Case Interrupt* 31b⟩
⟨*Case Write Words* 32a⟩
⟨*Case Write Address* 33a⟩
⟨*Case Read Words* 33c⟩

⟨*Case Set Length* 34c⟩
⟨*Case Set Interface* 34d⟩

For the following `case` statements it has to be known whether the device driver should use the parallel port or the ISA card interface. Thus, a new header file named `er2gdef.h` is introduced. It keeps some defines that will be used by all programs somehow relating to this driver.

30a ⟨*er2gdef.h* 30a⟩≡

```
⟨Disclaimer 3b⟩

/** \file er2gdef.h
Header file for some defines, common to all programs and libs that
want to use the ER2 via the Linux device driver ''er2p''.
\author Dirk Baechle
\version 1.0
\date 26.11.2003
*/

#ifndef _ER2GDEF_H
#define _ER2GDEF_H

/* Defines for the ER2 host interface type */
#define PARALLEL_PORT          0
#define ISA_CARD               1

/* Defines for function return values */
#define OK                     0
#define ERROR                  1

#endif
```

Defines:
`_ER2GDEF_H`, never used.
`ERROR`, never used.
`file`, used in chunks 3, 7a, 11b, 13a, 14, 28b, 29b, 36c, 39, and 40a.
`ISA_CARD`, used in chunk 34d.
`OK`, never used.
`PARALLEL_PORT`, used in chunks 30–34.

A new global variable is added for the kind of host adapter that is used. The default is the parallel port interface.

30b ⟨*Global variables* 5b⟩+≡ (2a) ◁10b 31c▷

```
/** Which interface is used? 0 equals parallel port interface, 1
equals ISA card interface. */
static int used_interface = PARALLEL_PORT;
```

Defines:
`used_interface`, used in chunks 31–34.
Uses `PARALLEL_PORT` 30a.

The first *case* is the reset of the ER2. It has to call the appropriate function ic_ifc_reset_er2 or pp_ifc_reset_er2.

31a ⟨*Case Reset* 31a⟩≡ (29c)

```
case IOCTL_ER2_RESET: if (used_interface == PARALLEL_PORT)
                      {
                        pp_ifc_reset_er2();
                      }
                      else
                      {
                        ic_ifc_reset_er2();
                      }
                      break;
```

Uses **ic_ifc_reset_er2** 26b, **IOCTL_ER2_RESET** 28b, **PARALLEL_PORT** 30a, **pp_ifc_reset_er2** 20c, and **used_interface** 30b.

Next is the interrupt IRQ2, used for sending messages from the root processor to others in the network.

31b ⟨*Case Interrupt* 31b⟩≡ (29c)

```
case IOCTL_ER2_IRQ: if (used_interface == PARALLEL_PORT)
                    {
                      pp_ifc_irq_er2();
                    }
                    else
                    {
                      ic_ifc_irq_er2();
                    }
                    break;
```

Uses **ic_ifc_irq_er2** 28a, **IOCTL_ER2_IRQ** 28b, **PARALLEL_PORT** 30a, **pp_ifc_irq_er2** 22a, and **used_interface** 30b.

It is desirable to be able to read/write not only one word, but fill a whole buffer with one access, i.e. call of IOCTL_ER2_READ_WORDS or IOCTL_ER2_WRITE_WORDS. So, another global variable is added for the length of the data buffer that can be set by IOCTL_ER2_SET_LENGTH. Since the default value is 1, IOCTL_ER2_SET_LENGTH does not have to be called each time before reading/writing a single word.

31c ⟨*Global variables* 5b⟩+≡ (2a) ◁30b

```
/** The length of the data buffer. */
static int buffer_length = 1;
```

Defines:
buffer_length, used in chunks 32–34.

It is the task of the application to ensure that the data array is properly initialized and its length is correct.
After each IOCTL_ER2_READ_WORDS, IOCTL_ER2_WRITE_WORDS and IOCTL_ER2_WRITE_ADDRESS the length of the buffer is set back to 1 automatically.

The next *case* writes data words. After deciding which interface is selected, the pointer **ioctl_param** and the kernel function **get_user** are used to write the data words one after the other.

32a ⟨*Case Write Words* 32a⟩≡ (29c)

```
case IOCTL_ER2_WRITE_WORDS: temp = (int *) ioctl_param;
                            if (used_interface == PARALLEL_PORT)
                            {
                              ⟨PP:Case Write Words 32b⟩
                            }
                            else
                            {
                              ⟨IC:Case Write Words 32c⟩
                            }

                            /* Set buffer length to default */
                            buffer_length = 1;
                            break;
```

Uses **buffer_length** 31c, **IOCTL_ER2_WRITE_WORDS** 28b, **PARALLEL_PORT** 30a, and **used_interface** 30b.

32b ⟨*PP:Case Write Words* 32b⟩≡ (32a)

```
for (; buffer_length > 0; buffer_length--)
{

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

        pp_ifc_write_data_word(data);
}
```

Uses **buffer_length** 31c, **KERNEL_VERSION** 4b, and **pp_ifc_write_data_word** 23b.

32c ⟨*IC:Case Write Words* 32c⟩≡ (32a)

```
for (; buffer_length > 0; buffer_length--)
{

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

        ic_ifc_write_data_word(data);
```

```
}
```

Uses buffer_length 31c, ic_ifc_write_data_word 26c, and KERNEL_VERSION 4b.

For writing an address the length of the buffer is disregarded. Just one address is written and buffer_length is set back to 1.

33a ⟨*Case Write Address* 33a⟩≡ (29c)

```
case IOCTL_ER2_WRITE_ADDRESS: temp = (int *) ioctl_param;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
                              get_user(data, temp);
#else
                              data = get_user(temp);
#endif
                              ⟨which interface is used? 33b⟩

                              /* Set buffer length to default */
                              buffer_length = 1;
                              break;
```

Uses buffer_length 31c, IOCTL_ER2_WRITE_ADDRESS 28b, and KERNEL_VERSION 4b.

33b ⟨*which interface is used?* 33b⟩≡ (33a)

```
/* Which interface is used? */
if (used_interface == PARALLEL_PORT)
{
  pp_ifc_write_address_word(data);
}
else
{
  ic_ifc_write_address_word(data);
}
```

Uses ic_ifc_write_address_word 26d, PARALLEL_PORT 30a, pp_ifc_write_address_word 24a, and used_interface 30b.

Reading data words is similar to writing. Depending on the used interface, the pointer ioctl_param and the kernel function put_user are used to fill the buffer with data.

33c ⟨*Case Read Words* 33c⟩≡ (29c)

```
case IOCTL_ER2_READ_WORDS: temp = (int *) ioctl_param;
                           if (used_interface == PARALLEL_PORT)
                           {
                             ⟨PP:Case Read Words 34a⟩
                           }
                           else
```

```
                          {
                            ⟨IC:Case Read Words 34b⟩
                          }

                          /* Set buffer length to default */
                          buffer_length = 1;
                          break;
```

Uses **buffer_length** 31c, **IOCTL_ER2_READ_WORDS** 28b, **PARALLEL_PORT** 30a, and **used_interface** 30b.

34a ⟨*PP:Case Read Words* 34a⟩≡ (33c)

```
for (; buffer_length > 0; buffer_length--)
{
        data = pp_ifc_read_data_word();
        put_user(data, temp++);
}
```

Uses **buffer_length** 31c and **pp_ifc_read_data_word** 24b.

34b ⟨*IC:Case Read Words* 34b⟩≡ (33c)

```
for (; buffer_length > 0; buffer_length--)
{
        data = ic_ifc_read_data_word();
        put_user(data, temp++);
}
```

Uses **buffer_length** 31c and **ic_ifc_read_data_word** 27a.

This function sets the new length of the data buffer. After the next `read` or `write` operation the length will be set back to the default of “1” automatically.

34c ⟨*Case Set Length* 34c⟩≡ (29c)

```
case IOCTL_ER2_SET_LENGTH: temp = (int *) ioctl_param;
                           buffer_length = *temp;
                           break;
```

Uses **buffer_length** 31c and **IOCTL_ER2_SET_LENGTH** 28b.

The interface type should only be changed right after the device was opened, such that things do not end mixed up.

34d ⟨*Case Set Interface* 34d⟩≡ (29c)

```
case IOCTL_ER2_SET_INTERFACE: temp = (int *) ioctl_param;
                              if (*temp == PARALLEL_PORT)
                              {
                                used_interface = PARALLEL_PORT;
                              }
                              else
```

```
{
  used_interface = ISA_CARD;
}
break;
```

Uses IOCTL_ER2_SET_INTERFACE 28b, ISA_CARD 30a, PARALLEL_PORT 30a, and used_interface 30b.

That's all for the device driver. Now, only the module declarations are left:

35a ⟨*Module declarations* 35a⟩≡ (2a)
⟨*VFS Struct* 35b⟩
⟨*Init Module* 36a⟩
⟨*Cleanup Module* 36c⟩

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

35b ⟨*VFS Struct* 35b⟩≡ (35a)

```
/** 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: er2p_read,        /* read */
  write: er2p_write,      /* write */
  ioctl: er2p_ioctl,      /* ioctl */
  open: er2p_open,        /* open */
  release: er2p_release   /* release */
#else
  NULL,                   /* seek */
  er2p_read,              /* read */
  er2p_write,             /* write */
  NULL,                   /* readdir */
  NULL,                   /* select */
  er2p_ioctl,             /* ioctl */
  NULL,                   /* mmap */
  er2p_open,              /* open */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,                   /* flush */
#endif
  er2p_release            /* release */
#endif
};
```

Uses er2p_ioctl 29b, er2p_open 7a, er2p_read 13a, er2p_release 11b 11b, er2p_write 14, and KERNEL_VERSION 4b.

While initializing the module the main—and in fact, only—task is to register the device driver. The claiming of IO ports is done in er2p_open. This enables other applications—e.g. the parport driver—to use the printer port for different tasks as long as the device **er2p** is not opened, although the module may be loaded.

36a ⟨*Init Module* 36a⟩≡ (35a)

```
/** Initializes the module by registering the device driver.
@return 0 for success, < 0 for an error
*/
int init_module()
{
  int ret;

  ⟨try to register the device driver 36b⟩

  return(SUCCESS);
}
```

Defines:
`init_module`, used in chunk 36b.
Uses `SUCCESS` 5a.

For earlier kernels (< 2.4.x) the function `register_chrdev` has to be replaced by `module_register_chrdev` (see [3]).

36b ⟨*try to register the device driver* 36b⟩≡ (36a)

```
  ret = register_chrdev(DEVICE_MAJOR, DEVICE_NAME, &Fops);

  /* Negative return values signify an error */
  if (ret < 0)
  {
    printk("ER2P: <init_module> : Registering device failed with %d!", ret);
    return(ret);
  }

  printk("ER2P: Device registered with major device number %d\n", DEVICE_MAJOR);
```

Uses `Device` 3c, `DEVICE_MAJOR` 28b, `DEVICE_NAME` 5a, and `init_module` 36a.

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

36c ⟨*Cleanup Module* 36c⟩≡ (35a)

```
/** Cleanup by unregistering the appropriate file from /proc
*/
void cleanup_module()
{
  int ret;

  ⟨unregister the device 37⟩

}
```

Defines:
cleanup_module, used in chunk 37.
Uses file 3a 30a.

For earlier kernels (< 2.4.x) the function unregister_chrdev has to be replaced by module_unregister_chrdev (see [3]).

37 ⟨*unregister the device* 37⟩≡ (36c)

```
ret = unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);

if (ret < 0)
{
  printk("ER2P: <cleanup_module> : Error %d while unregistering\n", ret);
}
```

Uses cleanup_module 36c, DEVICE_MAJOR 28b, and DEVICE_NAME 5a.

That's 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.

The right 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 complete kernel sources have to be installed for the compilation.

For older versions of the Linux kernel (< 2.4.x) the following `Makefile` can be used. The variable `USE_PARPORT` probably has to be undefined then. Additionally, the functions `register_chrdev` and `unregister_chrdev` have to be replaced by `module_register_chrdev` and `module_unregister_chrdev`, respectively (see [3]).

38 ⟨*Makefile.old* 38⟩≡

```
CC=gcc
MODCFLAGS=-O2 -Wall -DMODULE -D__KERNEL__ -DLINUX
INCLINUX=/usr/include

all: er2p.o


er2p.o: er2p.c $(INCLINUX)/linux/version.h
        $(CC) er2p.c -c $(MODCFLAGS) -I$(INCLINUX)

er2p.c: er2p.nw
        notangle -Rer2p.c er2p.nw > er2p.c
        notangle -Rer2p.h er2p.nw > er2p.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 er2p
```

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 host adapter.
For removing the module again, one has to type:

```
rmmod er2p
```

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 er2p c 219 0
```

The resulting file normally has read/write access only for its owner, which is *root* in this case. For the other users a new group—named "er2" for example—should be added to the system. After issueing the commands:

```
chgrp er2 /dev/er2p
chmod g+w /dev/er2p
```

all members of this group can use the device file.

6.2 Ensure correct settings for parallel port

Once again the remark: The parallel port interface for the ER2 needs the EPP/SPP mode! Furthermore, the IO addresses are fixed to the parallel port `LPT1` at `0x378`. So make sure that the appropriate settings in the BIOS are correct.

6.3 Example program

Now, a short example is given of how to use the `ioctl` functions:

39 ⟨*er2ptest.c* 39⟩≡

```
#include <fcntl.h>
#include "er2gdef.h"
#include "er2p.h"

int main(void)
{
  int file_desc, data;
  int mem_address = 0x5000;

  ⟨try to open device file 40a⟩
  ⟨reset the ER2 40b⟩
  ⟨detect number of root processor 40d⟩
  ⟨initialize memory at mem-address 40c⟩
  ⟨check data at mem-address 40e⟩
  ⟨write memory at mem-address 41a⟩
  ⟨read data at mem-address 41b⟩

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

  return(0);
}
```

Defines:
  `main`, never used.
Uses `file` 3a 30a.

40a ⟨*try to open device file* 40a⟩≡ (39)

```
/* try to open device file */
file_desc = open("/dev/er2p", O_RDONLY);
if (file_desc < 0)
{
  printf("Can not open device file er2p!\n");
  return(-1);
}
```

Uses `file` 3a 30a.

40b ⟨*reset the ER2* 40b⟩≡ (39)

```
/* reset ER2 */
ioctl(file_desc, IOCTL_ER2_RESET, &data);
```

Uses `IOCTL_ER2_RESET` 28b.

40c ⟨*initialize memory at mem-address* 40c⟩≡ (39)

```
data = mem_address;
/* set address */
ioctl(file_desc, IOCTL_ER2_WRITE_ADDRESS, &data);
/* write 16-bit word */
data = 0x0000;
ioctl(file_desc, IOCTL_ER2_WRITE_WORDS, &data);
```

Uses `IOCTL_ER2_WRITE_ADDRESS` 28b and `IOCTL_ER2_WRITE_WORDS` 28b.

40d ⟨*detect number of root processor* 40d⟩≡ (39)

```
data = 0x4207;
/* set address */
ioctl(file_desc, IOCTL_ER2_WRITE_ADDRESS, &data);
/* read 16-bit word */
ioctl(file_desc, IOCTL_ER2_READ_WORDS, &data);
printf("Root is #%d...\n", data);
```

Uses `IOCTL_ER2_READ_WORDS` 28b and `IOCTL_ER2_WRITE_ADDRESS` 28b.

40e ⟨*check data at mem-address* 40e⟩≡ (39)

```
data = mem_address;
/* set address */
ioctl(file_desc, IOCTL_ER2_WRITE_ADDRESS, &data);
/* read 16-bit word */
ioctl(file_desc, IOCTL_ER2_READ_WORDS, &data);
printf("16 bit word at address %X is : %X\n", mem_address, data);
```

Uses `IOCTL_ER2_READ_WORDS` 28b and `IOCTL_ER2_WRITE_ADDRESS` 28b.

41a ⟨*write memory at mem-address* 41a⟩≡ (39)

```
data = mem_address;
/* set address */
ioctl(file_desc, IOCTL_ER2_WRITE_ADDRESS, &data);
/* write 16-bit word */
data = 0xDBDB;
ioctl(file_desc, IOCTL_ER2_WRITE_WORDS, &data);
```

Uses IOCTL_ER2_WRITE_ADDRESS 28b and IOCTL_ER2_WRITE_WORDS 28b.

41b ⟨*read data at mem-address* 41b⟩≡ (39)

```
data = mem_address;
/* set address */
ioctl(file_desc, IOCTL_ER2_WRITE_ADDRESS, &data);
/* read 16-bit word */
ioctl(file_desc, IOCTL_ER2_READ_WORDS, &data);
printf("16-bit word at address %X is : %X\n", mem_address, data);
```

Uses IOCTL_ER2_READ_WORDS 28b and IOCTL_ER2_WRITE_ADDRESS 28b.

List of code chunks

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

⟨*Attach port* 8e⟩
⟨*Auxiliary IOCtl functions* 15b⟩
⟨*Case Interrupt* 31b⟩
⟨*Case Read Words* 33c⟩
⟨*Case Reset* 31a⟩
⟨*Case Set Interface* 34d⟩
⟨*Case Set Length* 34c⟩
⟨*Case Statement* 29c⟩
⟨*Case Write Address* 33a⟩
⟨*Case Write Words* 32a⟩
⟨*check data at mem-address* 40e⟩
⟨*check if device has not been opened yet* 7b⟩
⟨*check if ISA port regions are accessible* 7c⟩
⟨*claim ISA port regions* 11a⟩
⟨*claim parallel port regions* 10c⟩
⟨*Cleanup Module* 36c⟩
⟨*Defines* 5a⟩
⟨*Defines for the ISA card* 6c⟩
⟨*Defines for the parallel port* 6b⟩
⟨*Detach port* 9a⟩
⟨*detect number of root processor* 40d⟩
⟨*Device declarations* 5c⟩
⟨*Device IOCtl* 15a⟩

⟨*Device IOCtl function* 29b⟩
⟨*Device Open* 7a⟩
⟨*Device Read* 13a⟩
⟨*Device Release* 11b⟩
⟨*Device Write* 14⟩
⟨*Disclaimer* 3b⟩
⟨*er2gdef.h* 30a⟩
⟨*er2p.c* 2a⟩
⟨*er2p.h* 2b⟩
⟨*er2ptest.c* 39⟩
⟨*fill buffer with zeros* 13b⟩
⟨*Global variables* 5b⟩
⟨*Header* 3c⟩
⟨*HF:Defines* 28b⟩
⟨*HF:Header* 3a⟩
⟨*HF:Include files* 29a⟩
⟨*IC:Auxiliary IOCtl functions* 25a⟩
⟨*IC:Case Read Words* 34b⟩
⟨*IC:Case Write Words* 32c⟩
⟨*IC:Generate Strobe* 25c⟩
⟨*IC:Read Word Data* 27a⟩
⟨*IC:Reset ER2* 26b⟩
⟨*IC:Trigger ER2 Interrupt* 28a⟩
⟨*IC:Write Word Address* 26d⟩
⟨*IC:Write Word Data* 26c⟩
⟨*Include files* 4a⟩
⟨*Init Module* 36a⟩
⟨*initialize memory at mem-address* 40c⟩
⟨*Linux includes* 4b⟩
⟨*Makefile.old* 38⟩
⟨*Module declarations* 35a⟩
⟨*Parport support functions* 8d⟩
⟨*PP:Auxiliary IOCtl functions* 15c⟩
⟨*PP:Case Read Words* 34a⟩
⟨*PP:Case Write Words* 32b⟩
⟨*PP:Read Data* 23a⟩
⟨*PP:Read Latch* 22b⟩
⟨*PP:Read Word Data* 24b⟩
⟨*PP:Reset ER2* 20c⟩
⟨*PP:Send Strobe Signal* 20b⟩
⟨*PP:Trigger ER2 Interrupt* 22a⟩
⟨*PP:Write Address* 21c⟩
⟨*PP:Write Byte* 21b⟩
⟨*PP:Write Latch* 21a⟩
⟨*PP:Write Word Address* 24a⟩
⟨*PP:Write Word Data* 23b⟩
⟨*read data at mem-address* 41b⟩
⟨*register driver with parport* 9b⟩
⟨*register parallel device* 10a⟩
⟨*release ISA port regions* 12a⟩

⟨*release parallel port regions* 12b⟩
⟨*reset the ER2* 40b⟩
⟨*try to open device file* 40a⟩
⟨*try to register the device driver* 36b⟩
⟨*unregister parallel device driver* 12c⟩
⟨*unregister the device* 37⟩
⟨*VFS Struct* 35b⟩
⟨*which interface is used?* 33b⟩
⟨*write memory at mem-address* 41a⟩

Index

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

`_ER2GDEF_H`: 30a
`_ER2P_H`: 2b
`buffer_length`: 31c, 32a, 32b, 32c, 33a, 33c, 34a, 34b, 34c
`byte`: 20a, 20b, 21a, 21b, 21c, 22b, 23a, 23b, 24a, 24b
`cleanup_module`: 36c, 37
`Device`: 3a, 3c, 7b, 36b
`DEVICE_FILE_NAME`: 28b
`device_is_open`: 5b, 7a, 7b, 11b
`DEVICE_MAJOR`: 28b, 36b, 37
`DEVICE_NAME`: 5a, 9c, 10c, 11a, 36b, 37
`er2p_attach`: 8e, 9c
`er2p_detach`: 9a, 9c
`er2p_device`: 10b, 10c, 12b, 12c
`er2p_driver`: 9c, 10a
`er2p_ioctl`: 29b, 35b
`er2p_open`: 7a, 35b
`er2p_port`: 10b, 10c, 12b
`er2p_read`: 13a, 35b
`er2p_release`: 11b, 11b, 35b
`er2p_write`: 14, 35b
`ERROR`: 30a
`file`: 3a, 3b, 3c, 7a, 11b, 13a, 14, 28b, 29b, 30a, 36c, 39, 40a
`IC_ADDRESS_MODE`: 6c, 26d
`ic_ifc_generate_strobe`: 25c, 26b, 26c, 26d, 27a
`ic_ifc_irq_er2`: 28a, 31b
`ic_ifc_read_data_word`: 27a, 34b
`ic_ifc_reset_er2`: 26b, 31a
`ic_ifc_write_address_word`: 26d, 28a, 33b
`ic_ifc_write_data_word`: 26c, 32c
`IC_OUTP_LATCH`: 6c, 7c, 11a, 12a, 26c, 26d, 27a
`IC_READ_MODE`: 6c, 27a
`IC_RESET_DELAY`: 26a, 26b
`IC_RESET_MODE`: 6c, 26b
`IC_STROBE_HIGH`: 6c, 25c
`IC_STROBE_LOW`: 6c, 25c

IC_WRITE_MODE: 6c, 26c
init_module: 36a, 36b
IOCTL_ER2_IRQ: 28b, 31b
IOCTL_ER2_READ_WORDS: 28b, 33c, 40d, 40e, 41b
IOCTL_ER2_RESET: 28b, 31a, 40b
IOCTL_ER2_SET_INTERFACE: 28b, 34d
IOCTL_ER2_SET_LENGTH: 28b, 34c
IOCTL_ER2_WRITE_ADDRESS: 28b, 33a, 40c, 40d, 40e, 41a, 41b
IOCTL_ER2_WRITE_WORDS: 28b, 32a, 40c, 41a
IRQ: 27b, 28a
ISA_CARD: 30a, 34d
ISA_STROBE_DELAY: 25b, 25c, 26c, 26d
KERNEL_VERSION: 4b, 11b, 13a, 14, 32b, 32c, 33a, 35b
main: 39
MODVERSIONS: 4b
OK: 30a
PARALLEL_PORT: 30a, 30b, 31a, 31b, 32a, 33b, 33c, 34d
PP_IFC_IRQ_ER2: 19, 22a
pp_ifc_irq_er2: 22a, 31b
PP_IFC_READ_DATA: 19, 23a
pp_ifc_read_data_word: 24b, 34a
PP_IFC_READ_LATCH: 19, 22b
PP_IFC_RESET_ER2: 19, 20c
pp_ifc_reset_er2: 20c, 31a
pp_ifc_send_strobe: 20b, 20c, 21a, 21b, 21c, 22a
PP_IFC_STROBE_HIGH: 19, 20b, 22b, 23a
PP_IFC_STROBE_LOW: 19, 20b, 22b, 23a
PP_IFC_WRITE_ADDRESS: 19, 21c
pp_ifc_write_address: 21c, 24a
pp_ifc_write_address_word: 24a, 33b
PP_IFC_WRITE_DATA: 19, 21b
pp_ifc_write_data: 21b, 23b
pp_ifc_write_data_word: 23b, 32b
PP_IFC_WRITE_LATCH: 19, 21a
pp_ifc_write_latch: 21a, 23b, 24a
PP_LPT1_COMMAND: 6b, 20b, 20c, 21a, 21b, 21c, 22a, 22b, 23a
PP_LPT1_DATA: 6b, 9b, 10c, 12b, 21a, 21b, 21c, 22b, 23a
PP_LPT1_STATUS: 6b
PP_STROBE_DELAY: 19, 20b, 22b, 23a
ssize_t: 13a, 14
SUCCESS: 5a, 7a, 29b, 36a
USE_PARPORT: 8a, 8b, 8c, 9b, 9c, 10b, 10c, 12b, 12c
used_interface: 30b, 31a, 31b, 32a, 33b, 33c, 34d

References

[1] Dirk Bächle. *Ermittlung der Datentransferraten für den alten und neuen Host-Adapter des ER2*, 2000. Internal report.

[2] Georg-Friedrich Mayer-Lindenberg. *Message Passing im ER2 und Funktionen der Laufzeitkerne*, 1997. Internal report.

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

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

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