Parallel Port Device Drivers: A Study in Driver Creation

Noel Vega
Languages for Embedded Systems Design, Fall, 2002

December 16, 2002

Abstract

If monitors, mice, keyboards, and other computer peripherals could talk to any given computer directly, there would be no compatibility issues between hardware and operating systems. Unfortunately, such a nirvana does not yet exist. As such, computer users are left to deal with device drivers - hard to decipher pieces of code designed to allow peripherals to "communicate" with the operating system and transfer direction to and from the computer. The complexity of device drivers, however, makes it very difficult to begin a process of automating their creation.

This paper looks into parallel port drivers in Linux, Free-BSD, Windows 2000, and Windows NT, deciphering the code to arrive at a generalized driver structure for the four operating systems and, ultimately, a set of "pseudo" device drivers for the operating systems. By clearly understanding the structure of a device driver, it becomes a lot easier to attack the problem of automating its creation, which in turn will make it easier for users to install hardware on any machine.

1 Introduction

Computer science has been brought to the point where computers are everywhere in society. Certainly, the hardware used to perform the various tasks required of computers needs a means of communication with the computer. This becomes a great strain on hardware manufacturers, who are charged with the task of writing drivers for their hardware for each operating system (specifically, for each *version* of each operating system) they wish their hardware to be compatible with. Given the difficulty of writing device drivers, this can often lead to complications with drivers; since a new driver needs to be written for each device *and* for each OS version, there are bound to be complications with drivers. In fact, Wang, Malik, and Bergamaschi cite a Microsoft report regarding Windows XP which "shows that 61% of XP crashes are caused by driver problems." [7, 1] Clearly, a better understanding of device driver creation is called for.

While there are plenty of device drivers available, there are surprisingly few papers regarding the structure of a device driver. This paper aims to fill this gap, describing driver structures in enough detail to lend to the potential automation of their creation in the future. Such automation would clearly reduce problems with driver creation, perhaps even in an exponential fashion. At a bare minimum, it is useful - and important - to make a clear understanding of device drivers and their operation available to potential driver writers, who may in turn aid hardware manufacturers in creating drivers while the automation process is created.

2 Related Work

A field such as device driver creation, which has shown its purpose for a long time, has a rich history of related work. In hopes of furthering the understanding of the structure of device drivers, Viscarola and Mason offer a comprehensive manual on writing device drivers for Windows NT for "software engineers who have never written a device driver... those who have written drivers on other operating systems, and even... engineers who have already written a few drivers on Windows NT." [8, 1] Clearly, this is a problem even for the most seasoned driver programmer.

While a good deal of time is spent on writing device drivers, there is a growing interest in reducing the amount of time and energy spent writ-

ing device drivers. This is best accomplished either 1) through writing so-called "universal" device drivers, or 2) studying the possibility of driver automation. An example of a "universal" device driver is Thesycon's Universal Parallel Port Driver for Windows[4], which is designed to be precisely that: a driver for parallel ports designed to work for Windows NT 4.0, Windows 2000, and Windows XP. More interestingly, however, is the strive to automate driver creation. The process described by Wang, Malik, and Bergamaschi[7] aims to synthesize a platform-independent device driver from specified device behavior.

3 Methodology

As outlined above, the focus of this paper is to examine differences in driver structures across different platforms, namely Windows 2000, Windows NT, FreeBSD, and Linux. Since there are so many possible drivers available, not to mention so many categories for different drivers, I have decided to examine specifically parallel port drivers. These drivers, while not as large as many other device drivers, certainly display the general structure of device drivers for each of the specified platforms. A parallel port driver is far from a trivial example; printers, scanners, and disks are examples of the various pieces of hardware which utilize parallel ports[9, 6]. However, the operation is quite simple: there is a two-way byte channel, sending information to the hardware and receiving information from it. Even where a piece of hardware handles only input (a scanner) or only output (a printer), there is still two-way communication. For example, a scanner may only be reading information and sending it to a program for handling, but information still goes out to it to tell it to scan, or that the program is ready to go. Likewise, a printer may only output information to paper, but it will transmit information to the OS to let it know it is ready for the next job, or that there is a problem, etc.

4 Results

What follows is an examination of a parallel port device driver for each of the four platforms discussed in this paper:

4.1 Linux

As with all of the source code in Linux, a sample device driver was easy to get. Although it seemed quite complex at first, close examination showed that the driver was really quite verbose, but not necessarily complex.

Firstly, device drivers are not like traditional user-space C programs; they operate in kernel space, with kernel-level access modes, and as such do not have access to the traditional user-space `include` libraries, which do not have the same modes. Therefore, there are a number of libraries available solely for kernel-space source code; indeed, most kernel-space code will have numerous lines of code dedicated to `includes`. Most user-space libraries have kernel-space equivalents in Linux; for example, `malloc` can be used by including `linux/malloc.h`. Some standard `include` files for device drivers are `linux/sched.h`, with scheduling routines; `linux/kernel.h`, which contains various kernel-level debugging routines such as `printk()`, the kernel version of `printf()`; `linux/malloc.h`, which includes the kernel versions of `malloc()` and `free()`; and so on. Of particular note are `linux/config.h` and `linux/module.h`. `config.h` contains many configuration definitions, while `module.h` must be included by all drivers. There are, of course, many other header files which can be included by device drivers, depending on each particular driver's design.

Another very important aspect of driver design - indeed, with all kernel code - is the use of preprocessor symbols. These symbols indicate to pieces of source code various characteristics of the system which otherwise would not be available to the driver. For a device driver, two symbols must be defined: `__KERNEL__`, which identifies a piece of code as kernel code, and `MODULE`, which identifies a piece of source code as a module, or driver). These two symbols should be defined at the top of the source file, to signify that all code to follow belongs as kernel code, and is a device driver. Other symbols are typically defined and checked throughout the driver to send information to or receive information from the kernel.

Since drivers can (and frequently are) implemented at the console, and as such, a mechanism must be in place to set parameters at load time. There is, indeed, such a mechanism. The `MODULE_PARM` macro is used to declare a console parameter, and to assign it to a variable in the driver.

The final important preprocessor item to note

is the `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` macros, which - as the names imply - increment and decrement the module's use counts. This is vitally important, as systems will not unload modules that are busy (i.e. have nonzero usage counts).

Of course, there's more to a driver than macros and preprocessor symbols. Drivers need to have established entry and exit points, as well as certain identifiable points of entry to perform certain operations. This is accomplished by defining a `struct file_operations` variable, defined in `linux/fs.h`, which will include a series of pointers to each of the entry points in the driver. Any pointers not listed are `null`, and represent entry points that do not exist in the driver, for one reason or another. The defined `struct file_operations` variable is then passed to the kernel in the `register_chrdev` call, which is a vital call for any character device (such as a parallel port). This call is what actually registers the device with the kernel, and by passing the `struct file_operations` variable, the kernel then knows what functions to call inside of the driver. Likewise, there is also a `unregister_chrdev`, which - as the name implied - will unregister the character device. The two calls are typically found in the driver's `init` and `exit` functions, which - as implied - initialize and exit the driver. Like with the operations, you need to specify which functions refer to each operation; this is done with the calls `module_init()` and `module_exit()`, with the appropriate function as a parameter.

The most common operations are listed below[6]:

- `read`: Exactly as it says - will read data from the device.
- `write`: Will write data to the device
- `open`: Though it is not necessary, implementing this operation will let the driver know it has been opened successfully.
- `release`: Like `open`, this can be omitted. This is implemented to release the driver from the kernel. Note, though, that some instances of the driver may be created through the use of `fork` and other calls; in these instances, `release` is never called, but the driver exits nonetheless.
- `ioctl`: This is a very important operation, though is not always implemented. This operation makes the driver perform device-specific operations above and beyond read and write.
- `poll`: Asks the writer if it is in a state to either read or write. If it is not defined, the device is assumed to be both readable and writable.

Although there are more operations than the ones listed above, these operations provide for all of the basic functionality that may be required in a device driver.

After broadly outlining the parts of a device driver, we can now show a rough draft of what a device driver should look like. Bear in mind that the focus of the paper is specifically on parallel port drivers; as such, the structure of the driver below will be that of a character device, which would be the categorization of a parallel port.

```
// Comment section describing the driver and
// the device
// Written by: Noel Vega (noel.vega@pba.com)

#ifndef __KERNEL__    // Need this symbol
#  define __KERNEL__
#endif
#ifndef MODULE   // and this one!
#  define MODULE
#endif

// Two mandatory includes for device drivers
#include <linux/config.h>
#include <linux/module.h>
#include files pertaining to this device
#define any needed symbols here

// declare any console parameters here
MODULE_PARM(var_name, ``var_type'');
// Possible variable types:
// ``i'', an int; ``h'', a short; ``b'', a byte;
// ``l'', a long, and ``s'', a string.

// declare any other variables here

int open(struct inode *inode,
 struct file *filp)
{
  extern struct file_operations fops;
  MOD_INC_USE_COUNT;    // a must!
  // here's where we pass our
  // info to the kernel
  filp->f_op = &fops;

  // Any other code for opening the device
  return 0;
}

int release(struct inode *inode,
    struct file *filp)
{
```

```
  MOD_DEC_USE_COUNT;
  // any other cleanup code goes here
  return 0;
}

int read(struct inode *inode,
 struct file *filp,
 char *buf, int count)
{
  // implement the read operation here
  return 0;
}

int write(struct inode *inode,
  struct file *filp,
  const char *buf, int count)
{
  // implement write here
  return 0;
}

int poll(struct inode *inode,
 struct file *filp,
 int mode, select_table *table)
{
  return the current poll state;
}

// The all-important file_ops:

struct file_operations fops = {
  read: read,
  write: write,
  poll: poll,
  open: open,
  release: release,
};

// possible other functions
int init(void)
{
  // init code
  int return =
register_chrdev(majornum, name, fops);
  return 0; // unless code does otherwise
}

void exit(void)
{
  unregister_chrdev(majornum, name);
  // cleanup code
}

module_init(init);
module_exit(exit);
```

4.2 FreeBSD

As similar as FreeBSD is at the front end, the inner workings are just as similar. Device drivers for FreeBSD, while following a slightly different convention, have much of the same concepts behind them as they do in Linux, but are implemented differently.

There are a number of subtle differences between Linux and FreeBSD device drivers. For one, the `#include` files are typically in the `sys` directory, as opposed to the `Linux` directory. `printk()` is replaced with `uprintf()` in FreeBSD, though it is important to note that `printf()` is permissible in FreeBSD driver code.

There are, however, bigger implementational differences between drivers of the two OS's. A notable difference is that preprocessor symbols are used much less frequently in FreeBSD. For example, whereas in Linux all modules must define `__KERNEL__` and `MODULE` as well as including `linux/module.h` and `linux/config.h`, in FreeBSD only the `includes` are required (note that `config.h` is actually `conf.h` in FreeBSD). Finally, `malloc()` and `free()` are implemented as macros; hence, `malloc()` becomes `MALLOC()`, etc. There are also two extra related macros, `MALLOC_DECLARE` and `MALLOC_DEFINE`, which alert the kernel that a `MALLOC()` is forthcoming, and informs the kernel as to the nature of the `MALLOC()` call.

Even more important, however, are the differences between the data structures between the two. While in Linux a `struct file_operations` is required to point to the various driver entry points, in FreeBSD a character driver would achieve this through the use of a `struct cdevsw`, which is little more than a list of the various possible driver entry points. If one of the entry points is defined in the driver, the name of the function in the driver associated with the entry point is listed in that place; otherwise, a constant `no<entry_point_name>` is used. For example, if a driver did not implement the `ioctl` entry point, that part of the struct is set to `noioctl`. As a result, a common convention is to list each part of the struct with its own line, to ensure that every part of the struct is declared as well as for easy reading. In addition, the name of the device, the device's major number[1], and other related information is included inside of this struct, whereas in Linux most of this other information was set in the `register_chrdev` call. Each function is also declared as a function prototype with a return type of `d_<function-type>_t` early on. For

example, an open function is declared as a d_open_t function.[2] Finally, to declare the module as a device driver, you must make a call to the macro DEV_MODULE(). This macro informs the driver as to the name of the device, and contains a reference to the driver function which implements loading and unloading, FreeBSD's equivalent of Linux's init() and exit(). The loader function is essentially a switch statement, with a case for loading (which includes code to store the device locally based on the struct cdevsw declared earlier, among other things) and one for unloading (which usually includes a call to destroy_dev(), a function which destroys the locally stored device, and FREE() to return the memory used)[5].

With a basic understanding of the underlying concepts behind device drivers for FreeBSD, we can now contruct a pseudo-driver for FreeBSD, keeping in mind again that this driver is specifically targeted at parallel ports, which are character devices. Also note the differences in the conventions used, i.e. the indentation used in function declarations:

```
// Comments describing the source code
// Written by: Noel Vega (noel.vega@pba.com)

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/kernel.h>
#include <sys/conf.h>

// Function prototypes
// Assuming we only read, write, open, close
d_open_t    open;
d_close_t   close;
d_read_t    read;
d_write_t   write;

// Entry points declared here
// Again, assume only the above operations
static struct cdevsw this_cdevsw = {
   open,
   close,
   read,
   write,
   noioctl,
   nopoll,
   nommap,
   nostrategy,
   <name>,
   <major_num>,
   nodump,
   nopsize,
   D_TTY,
   -1
};

// local vars and other definitions

// The loader:
static int
loader(struct module *m, int cmd, void *arg)
{
  int err = 0;
  // local var declarations
  switch(cmd) {
  case MOD_LOAD:
    // load code here
    break;
  case MOD_UNLOAD:
    // unload code here
    break;
  default:
    err = EINVAL; // invalid command error
    break;
  }
  return(err);
}

int
open(dev_t dev, int flags, int devtype,
     struct proc *p)
{
  // device open code here
  return(0);
}

int
close(dev_t dev, int flags, int devtype,
      struct proc *p)
{
  // device close code here
  return(0);
}

int
read(dev_t dev, struct uio *uio,
     int flags)
{
  // read code goes here
  return(0);
}

int
write(dev_t dev, struct uio *uio,
      int flags)
{
  // write code goes here
```

[2] A major number is used in both Linux and FreeBSD to help the kernel identify devices. They are usually dynamically assigned based on which numbers are available.

```
  return(0);
}

DEV_MODULE(name, loader, NULL);
```

Clearly, there are more developed drivers than this one, and more parts which are not included here, but filling in the missing functionality with device-specific code will accomplish the task.

4.3 Windows 2000

Windows 2000 is about as different from Linux and FreeBSD as the two were similar to each other. While there are some functional similarities between the two, including the kinds of routines a driver may provide, there are a vast number of differences in structure, implementation, and conventions. The design of Windows 2000 had the OS implement many of the same features as had existed under Windows NT. From a developmental standpoint, the two operating systems are virtually the same, even sharing key header files such as `ntddk.h`, explained below.

Firstly, we can examine the differences in structure. The Windows 2000 I/O Manager is the aspect of the operating system in charge of handling device drivers. As needed, the manager will call various routines in the driver, much as is done in Linux and FreeBSD. For example, the Win2K equivalents of Linux's `init()` and `exit()` are `DriverEntry()` and `Unload()`. Note, however, that Windows offers a separate routine for cases of system shutdown and for cases of a system crash. The `Shutdown()` routine provides a driver with an opportunity to close down the hardware without completely cleaning up after itself; the thought process here is that the system is "going away" anyway, thus a perfect cleanup is not required.[2] In addition, in the event of a system crash, there may be an opportunity for a driver to gain some semblance of control and perform vital functions to close down during the crash. These operations are performed in the driver's `Bugcheck()` routine. Finally, rather than a series of `#includes`, there is one master header file for driver writing, namely `ntddk.h`.

In addition, there are a number of I/O routines for Win2K drivers, as opposed to simply one `ioctl()` routine. At a minimum, drivers must implement a `CreateDispatch()` and a `CloseDispatch()` routine. These two are the Win2K equivalents of `open()` and `close()` routines, and are called as such. Other routines include an Interrupt Service Routine (ISR), which handles hardware interrupts, a Start I/O routine, which begins the I/O process, synchronization routines, and much more.

Indeed, without one centralized place to declare all of these routines, it can be quite confusing. Thankfully, much as was done in Linux and FreeBSD, there is such a mechanism. In the driver's `DriverEntry()` routine, a `PDRIVER_OBJECT` is passed to the driver which contains pointers to various routines, much as Linux's `struct file_operations`. It is implemented in a slightly different way, however. The object is a set of pointers, as was implemented in Linux. One pointer goes to the `Driver_Unload` routine. The rest are implemented as an array of pointers called `MajorFunction`. A series of constants is used to index the array, with each constant being mapped to the driver function for which it was named. Simply setting the pointer for each implemented function to the appropriate routine in the driver will accomplish the task. For example, to set the `DispatchCreate()` function, the line of code would be: `pDriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;` Other functions are assigned in a similar fashion.

Finally, the device has to be created, and the OS informed of it. Part of this is done externally, with a new entry in the kernel registry notifying the kernel of a new driver. Once the driver is called, though, the rest is handled internally. A call to `IoCreateDevice()`, containing the `PDRIVER_OBJECT` declared earlier, the size of the data structure used to represent the driver, the name of the device, and a pointer to the device object[3]are used. [4] Lastly, a symbolic link must be created to the device. The call `IoCreateSymbolicLink()` does this trick, taking in two strings representing the symbolic link and the actual name of the device. Before going through a pseudo driver, one important distinction must be made about Win2K drivers. Just about all I/O under Windows 2000 is packet-driven, with references to *I/O Request Packets (IRPs)*. The IRP's contain all the information needed for the function to complete the I/O requested. When the I/O is completed, the IRP is sent back to the I/O Manager

[4]The term "object" is used here because there is a recognized degree of opject-oriented programming in Win2K drivers; indeed, many drivers are actually written in C++.More on this in page 64 of the Windows 2000 Device Driver Book[2].

with a status code, which is then returned back to the user.[2]

With this being cleared up, we can now complete a pseudo device driver. Note that there is usually some information stored in a defined `struct` about the driver, usually defined in a separate header file.[8] As elsewhere in this paper, we assume the driver to only need open, close, read, and write functions, consistent with a parallel port. Also note the conventions used in indentation, which are vastly different than those used under Linux and FreeBSD.

```
// Comments, as always.
// Written by: Noel Vega (noel.vega@pba.com)

#include <ntddk.h>
#any necessary includes

// forward declarations
static NTSTATUS CreateDev (
  IN PDRIVER_OBJECT pDriverObject,
  IN ULONG devNum);

static VOID Unload (
  IN PDRIVER_OBJECT pDriverObject);

static NTSTATIC DispCreate (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp);

static NTSTATUS DispClose (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp);

static NTSTATUS DispWrite (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp);

static NTSTATUS DispRead (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp);

// Since this is sometimes in C++,
// extern ``C'' is used here
extern ``C'' NTSTATUS DriverEntry (
  IN PDRIVER_OBJECT pDriverObject,
  IN PUNICODE_STRING pRegPath) {

  NTSTATUS ret;
  // NTSTATUS is a return value,
  // probably just an int on most systems..
  ULONG devNum = 0;
  PDEVICE_OBJECT pDevObj;

  // other local var definitions
  pDriverObject->DriverUnload = Unload;
  pDriverObject->MajorFunction[IRP_MJ_CREATE]
       = DispCreate;
  pDriverObject->MajorFunction[IRP_MJ_CLOSE]
       = DispClose;
  pDriverObject->MajorFunction[IRP_MJ_WRITE]
       = DispWrite;
  pDriverObject->MajorFunction[IRP_MJ_READ]
       = DispRead;
  ret = IoCreateDevice(
    pDriverObject,
    sizeof(<driver_type>),
    <device name>,
    FILE_DEVICE_UNKNOWN,
    0, TRUE,
    &pDevObj);
  // exit if there's an error
  if (!NT_SUCCESS(ret))
    return ret;
  ret = IoCreateSymbolicLink(
    <symbolic link name>,
    <device name>);
  // exit on error
  if (!NT_SUCCESS(ret)) {
     IoDeleteDevice(pDevObj);
     return ret;
  }
  return STATUS_SUCCESS;
}

VOID Unload (
   IN PDRIVER_OBJECT pDriverObject) {

   <for each device controlled> {
      <find the symbolic link>
      IoDeleteSymbolicLink(<link>);
      <find the device>
      IoDeleteDevice(<device>);
   }
}

NTSTATUS DispCreate (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp) {
  // handle any startup issues here
  pIrp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS DispClose (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp) {
  // handle cleanup issues here
  pIrp->IoStatus.Status = STATUS_SUCCESS;
  IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}
```

```
NTSTATUS DispWrite (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp) {
  // write code goes here
  // Status codes assume success...
  pIrp->IoStatus.Status = STATUS_SUCCESS;
  pIrp->IoStatus.Information =
    <# bytes xferred>;
  IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}

NTSTATUS DispRead (
  IN PDEVICE_OBJECT pDevObj,
  IN PIRP pIrp) {
  // Read code goes here
  // Status codes assume success...
  pIrp->IoStatus.Status = STATUS_SUCCESS;
  pIrp->IoStatus.Information =
    <# bytes xferred>;
  IoCompleteRequest(pIrp, IO_NO_INCREMENT);
  return STATUS_SUCCESS;
}
```

Clearly, Win2K code is quite verbose - and not as easily understood as Linux or FreeBSD code. This would more than certainly explain the Windows crash problem; it takes quite a bit of understanding to write a Win2K driver.

4.4 Windows NT

As mentioned above, Windows 2000 was designed to have much of the same functionality as Windows NT. Many of the hardware features in Windows NT are featured in Windows 2000 as well. Windows 2000 represented a quantum leap in capability and security from Windows 98. As a result, there is nothing to examine with regards to Windows NT device drivers; a driver written for Windows 2000 should, in fact, run fine on Windows NT. For example, the Universal Parallel Port Driver for Windows by Thesycon System Software[4] does, in fact, support both Windows NT and Windows 2000 with one source file and no version checking (i.e. it does not have any conditional compilation, such that it would compile some code for Win2K, and others for WinNT[5]). [6] Indeed, the Thesycon driver also supports Windows XP, which suggests that there may be a reduction in compatibility issues from one version of Windows to another in the future.

[6] Such conditional compilation exists for Linux; certain versions of the kernel will define a preprocessor symbol identifying that version. By checking for the symbol, you can conditionally compile.

5 Conclusions

While Windows systems may be reducing the troublesome compatibility issues which have plagued them in the past, there is still a strong issue of compatibility for hardware from one platform to the next. Even in systems based on the same kernel (as both Linux and FreeBSD are based on the UNIX kernel), there are sufficient compatibility issues to keep drivers designed for one system from working on the other. Indeed, in many cases a series of preprocessor conditionals are needed just to get some drivers to compile for various versions of Linux alone. Nonetheless, there are a number of functionality similarities between various systems. In the future, it should be possible to exploit these similarities to create a system whereby driver creation is automated, perhaps through use of a domain-specific language. For example, each of the operating systems requires one function to open and one to close a device file. It may be possible to specify functionality in a separate language, then have a compiler tweak the functionality to fit the requirements for opening and closing a device file in each OS. Clearly, there is a lot of room for enhancement in this field, and there seem to be exciting possibilities for doing so.

References

[1] Peter Baer Galvin Abraham Silberschatz and Greg Gagne. *Operating System Concepts, 6th Ed.* John Wiley and Sons, 2002.

[2] Art Baker and Jerry Lozano. *The Windows 2000 Device Driver Book.* Prentice Hall PTR, 2001.

[3] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, Snowbird, Utah, 2001. ACM.

[4] G. Hildebrandt. *Universal Parallel Port Driver for Windows Reference Manual.* Thesycon System Software & Consulting, Germany, March 2002.

[5] The FreeBSD Documentation Project. Freebsd developers' handbook. Technical report, FreeBSD.org, 2002.

[6] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, Second Edition.* O'Reilly, 2001.

[7] Sharad Malik Shaojie Wang and Reinaldo A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. Technical report, Electrical Engineering Department, Princeton University and IBM T.J. Watson Research Center, 2002. Submitted to the Date 2003 conference, but not accepted as of yet.

[8] Peter G. Viscarola and W. Anthony Mason. *Windows NT Device Driver Development.* New Riders, 2001.

[9] Angel Yu. Custom windows nt 4.0 parallel port device driver: A component of a network performance measurement tool. Master's thesis, California Polytechnic State University, 1998.