Writing Drivers for NetBSD

Jochen Kunz

Version 1.0.1e
August 25th, 2003

An introduction into NetBSD's `autoconfig(0)` system and the basics of device drivers under NetBSD.

Contents

List of Figures

1 Preface

This document is intended to teach the basics of Unix-Kernelprogramming to a beginning Programmer with basic C knowledge. As an example, a device driver for a floppy drive under NetBSD was chosen, as the hardware and necessary documentation was available but the driver itself missing. NetBSD was chosen as the target operating system, as it lends itself perfectly as a teaching example due to its clearly structured source code and well defined interfaces.

Unfortunately, there is hardly any specific documentation on Unix-Kernelprogramming under NetBSD apart from references to the functions in section 9 of the NetBSD manual pages. These manual pages are however missing an introduction, a document that clarifies the connection between the various functions. This document attempts to provide such an introduction, to act as the necessary glue between the individual parts. Therefore, I will reference external documents, particularly section 9 of the NetBSD manual pages in many places. This document is mainly based on the experiences I made when I wrote the driver `rf(4)`[1] for the UnixBus / QBus RX211 8" floppy controller.

8" floppy? Those things existed? Yes. Those were the first floppies, built at the end of the 60s / beginning of the 70s. *UniBus / QBus? Whatsdat?* That's the most common bus found in VAXen[2]. The VAX was *the* machine of the late 70s up until the beginning of the 90s. Then it was obsoleted by the Alpha architecture. BSD Unix has a long and glorious history on the VAX. [McK 99] But why am I writing a driver today for such antiquated technology? In the end, it doesn't really matter if I explain the necessary steps using the latest 1GBit/s Ethernetcard for a PCIX Bus or anything else. The underlying principles are the same. Besides, the hardware used in this example is relatively simplistic, so that we can see the essential aspects instead of being hindered by PeeCee idiocrasy.

The following chapter gives a short overview of the `autoconf(9)` concept in NetBSD. Some details have been omitted, and I refer to the according manual pages to avoid duplication of information.

The third chapter documents the implementation of the `autoconf(9)` interface of `rf(4)`.

The fourth and last chapter covers the actual driver, i.e. the functionality of the driver carrying the data from and to the physical device.

In the appendix, you will find the complete source code of the driver as well

[1] *R*X01/02 *F*loppy

[2] plural for VAX

as a copy of the referenced manual pages.

Future prospects: In its current form, this document represents only a beginning. A description of a network device driver, the internal functionality of `bus_space(9)` and `bus_dma(9)` or what is required to port NetBSD to a new architecture would be possible extensions. Similarly, a discussion of the UVM / UBC internals or a file system interface would be of interest to implement a new file system for example. But at least the last example goes a bit too far away from the initial intent of giving an overview or an introduction to device driver programming and would be more suitable for a more extensive document on NetBSD Kernel internals, which one day may evolve out of this text.

Thanks to Hubert Feyrer and Marc Balmer, who took the time to proof-read my mental outpourings and provided incitement for some diagrams.

2 The autoconf(9) system

The kernel configuration file is based on three pillars: ioconf.c / cfdata and sys/kern/subr_autoconf.c. This concept has become known as *autoconf* . But what exactly is going on behind the curtain?

2.1 config(8)

There is one central file which declares the kernel configuration for a BSD Unix Kernel. Under NetBSD, this file is located in sys/arch/<arch>/conf. <arch> represents the appropriate machine- / processor architecture. In our example, this is vax, i.e. d.h. sys/arch/vax/conf. In this folder, you can find the kernel configuration file GENERIC, which contains all the drivers and options supported by this architecture. You can create a user-defined configuration file by copying the file to a new name in this directory and editing it. Usually, this means commenting out all the drivers for devices not available in the particular machine. This process can be automated by using the tool pkgsrc/sysutils/adjustkernel.

After calling config(8), it reads the kernel configuration file to determine which drivers / functionality should be included in the kernel. Some "‘files.*’" files assign the .c- and .h-files to the various drivers and functionality. Using these dependencies, config(8) creates a compilation directory containing a Makefile as well as a range of .c- and .h-files. The .h-files usually contain defines with parameters such as the max. number of driver instances (for example PseudoTTYs, BPF, ...), kernel options such as KTRACE, ... The file param.c also falls into this category.

The compilation directory is named after the kernel configuration file and is located in sys/arch/vax/compile. After changing into said directory, the actual compilation is started by the command make depend netbsd. See config(8) and http://www.netbsd.org/Documentation/kernel/ for details.

2.2 ioconf.c and cfdata

The file ioconf.c in the compilation directory contains the data structure, marking the central point of access of the entire *autoconf* process. This *configuration data* table shows all the devices supported by the kernel. Let’s start with an excerpt of the kernels configuration file:

```
mainbus0        at root
```

```
ibus0           at mainbus0               # All MicroVAX
sbi0            at mainbus0               # SBI, master bus on 11/780.
vsbus0          at mainbus0               # All VAXstations

uba0            at ibus0                  # Qbus adapter
ze0             at ibus0                  # SGEC on-board ethernet
le0             at ibus0                  # LANCE ethernet (MV3400)

le0             at vsbus0 csr 0x200e0000 # LANCE ethernet
ze0             at vsbus0 csr 0x20008000 # SGEC ethernet
si0             at vsbus0 csr 0x200c0080 # VS2000/3100 SCSI-ctlr
asc0            at vsbus0 csr 0x200c0080 # VS4000/60 (or VLC) SCSI-ctlr
asc0            at vsbus0 csr 0x26000080 # VS4000/90 SCSI-ctlr

uda0            at uba? csr 0172150       # UDA50/RQDX?
qe0             at uba? csr 0174440       # DEQNA/DELQA
rlc0            at uba? csr 0174400       # RL11/RLV11 controller
rl*             at rlc? drive?            # RL01/RL02 disk drive

scsibus*        at asc?
scsibus*        at si?

sd*             at scsibus? target? lun?
st*             at scsibus? target? lun?
cd*             at scsibus? target? lun?
```

We quickly realize that the organization of the device drivers can be represented in a *treelike structure* as in figure 1. Attached to the imaginary `root` (which appears "out of nowhere") we find the first child, the abstract `mainbus`. This `mainbus` is parent to the children `ibus`, `sbi` and `vsbus`. These children in turn are parent to `uba`, `le`, `asc`, ... These relationships represent the above mentioned `cfdata` table found in the file `ioconf.c`. The programmer does not need to know or care about this table, as it is automagically created by `config(8)` .

It is important to realize that each device (node) has a parent (except for `root`, due to the old chicken-or-the-egg problem). A node that has children, is a Bus or a controller. The actual devices are the leaves of the tree. Each leaf and each node represent a device driver. That of course means, that there must be drivers for the

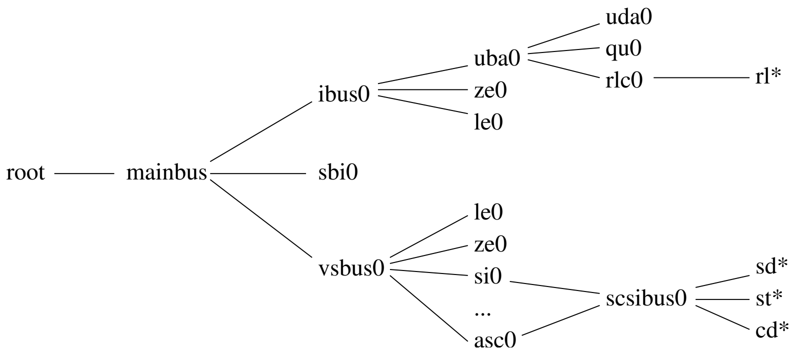

Figure 1: device tree

bus systems as well. The bus drivers are, especially in this context, responsible for locating the devices attached to the bus (i.e. the "busscan").

Another important realization lies in the fact that there are several different ways of arriving at the same driver! For example `le0`: `root => mainbus0 => ibus0 => le0` or `root => mainbus0 => vsbus0 => le0`. `le` is the actual driver for the LANCE Ethernet Chip. Since the core of the driver accesses the hardware only through *abstract,bus-independent* functions, the special details of the hardware are hidden from it. Instead of manipulating the hardware directly, the driver utilizes abstract handles. These handles and the according functions are provided by the parent (i.e. the driver of the bussystem). Of course, all possible parents (`vsbus` and `ibus` in this case) for a given child (`le`, in this case) need to provide the same interfaces.[3] These interfaces and the dependencies among the drivers and the other kernel subsystems are described in more detail via so-called *attributes* further down.

2.3 `sys/kern/subr_autoconf.c`

The functions found in `sys/kern/subr_autoconf.c` walk down the `cfdata` table in `ioconf.c` on boot and descend down the entire device tree. For this to work properly, each driver needs to implement a special interface for these functions.

The reader is advised to read the following manual pages, preferably in this order: `driver(9)`, `config(9)`, `autoconf(9)`.

[3]Things get even more entertaining when we account for other architectures. `le` may also be attached to `tc`, `pci`, `zbus`, `vme`, `dio`, `mainbus`, `sbus`, ...

2.4 Attributes and Locators

Something that unfortunately does not become quite clear are the differences and interrelations among the interface- and plain attributes and locators. A *plain attribute* simply signifies that a driver has a certain property, such as, for example, that the driver enables an ethernet or a serial interface. This allows several similar drivers to associate themselves with the attribute and thusly utilize the same source code. When a driver is included in a kernel that requires a certain attribute, the source code snippets that provide the attribute are then included in the kernel as well. `ifnet, ether, tty, isadma, ...` are examples of such plain attributes.

An *interface attribute* describes a logical software interface between some devices, typically a bus driver and the attached drivers. Usually, it contains one or more so-called "locators". A locator contains the "position" on the bus / controller at which the child-device can be found. In the above mentioned kernel configuration file, for example, there exists the `qe` device, which attaches to the `uba`[4], meaning the QBus driver implements the software interface labeled with the attribute `uba`, to which the device `qe` refers. `csr` is the (only) locator of the interface attribute `uba`.

```
device  uba { csr }
file    dev/qbus/uba.c                          uba

# DEQNA/DELQA Ethernet controller
device  qe: ifnet, ether, arp
attach  qe at uba
file    dev/qbus/if_qe.c                        qe
```

The above is an excerpt of `sys/dev/qbus/files.uba`. The first line introduces the interface attribute `uba` with the locator `csr`. The following line instructs `config(8)` to include the file `dev/qbus/uba.c` in the kernel compilation if a device is associated with the `uba` attribute. The last three lines define the `qe` device. It is associated with the three plain attributes `ifnet, ether, arp`, attaches to the interface attribute `uba` and the source code is found in `dev/qbus/if_qe.c`.

An example of an interface attribute with multiple locators is `isa`, which supports the locators `port, size, iomem, iosiz, irq, drq, drq2`. See the

[4] Before the QBus, there was the very similar UniBus; *U*ni*B*us *A*adapters then became `ubas`. Since both busses are very similar, a single bus driver is sufficient for both.

declaration in sys/dev/isa/files.isa. The locators given in the kernel configuration file directly lead to the according values in the void *aux parameters of the foo_match and foo_attach functions. (Well, read driver(9)? ;-))

Locators do not have to contain absolute values. Depending on the capabilities of the driver, wildcards may be possible. A typical candidate for wildcards is a bus- or controller driver supporting direct configuration. The "files.*" file defining the interface attribute has to provide standard values for such a locator in this case. Typical standard values are 0 for bus addresses or -1 for common indices. The chapter 3.1.2 shows an example of such a case. If no standard values are assigned, then the kernel configuration file needs to provide a value and wildcards are not allowed. A locator declared in [] is optional.

2.5 Where are my children?

There's one question the reader should have by now: Well, sure, the driver / device is found. But how and where does my driver look for its children? (If the driver does support a bus or a controller.) What's up with these config_search() and config_found_sm() functions?

There are two cases when integrating a device on a bus:

direct configuration The bus adapter hardware provides a complete list of all currently available physically available child-devices. By reading the "PCI configuration space", a bus driver can find out which PCI devices are currently available and thus only pull in the drivers for those devices.

indirect configuration With the QBus or ISA, the second case applies. With these busses, the driver can not simply loop through all the bus addresses to determine which devices do (and do not) exist.

In the case of indirect configuration, the bus driver has to utilize the config_search() function. config_search() walks down the potential child device drivers in cfdata, i.e. it will call the foo_match() function of all potential child device drivers. So the bus- / controller driver calls config_search() only once to find all the child devices. The config_attach() function of the driver of the located child device will therefore *not* be called. So the bus- / controller driver would have to search the cfdata table for the just found children and call config_attach() for those. But I have not found any driver that does this. Es described under autoconf(9) in the section on config_search(), you can achieve the same by using the func function parameter of config_search(). This func

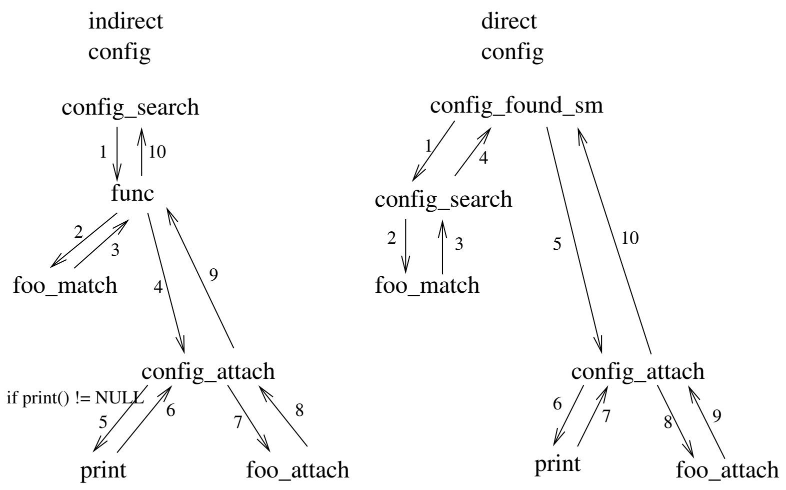

Figure 2: calling chain of `autoconf(9)` functions

function is provided by the bus- / controller driver. `config_search()` calls this function for all the child device drivers found in the `cfdata` table. That function, on the other hand, calls the `foo_match()` function of the child device driver and, should the child have located the device, the `config_attach()` function of the child device driver. If a bus / controller supports direct configuration, as for example PCI or PNP-ISA, the bus driver calls `config_found_sm()` once for each child device. This function first calls `config_search()` and then, if a child has been found, `config_attach()`. The `config_attach()` function reserves memory for the `softc` structure of the child device drivers and calls its `foo_attach()` function. In this case, the `submatch()` is often `NULL`, or the `config_found()` function is called immediately. The parent provides the `print` function , and passes a pointer to this function to `config_found_sm()`. The `print` function is called from within `config_attach()`, after `config_attach()` has printed a message like “`foo at bar`”. The `name` parameter of the `print` function is then `NULL`. The `print` function should be used to output more detailed information on the console about the child device, such as the exact device type, data transfer rate, ... If this is not desired, `NULL` may be passed in place of the function pointer.

But why do we look for the children using `config_search()`, if the bus- / controller driver already noticed the presence of any child device? Well, while the child device may without any doubt be present, the question remains whether or not a driver for it is included in the current kernel. If there is no driver for this child device, then config_search() fails and print (provided by the parent) is called directly instead of config_attach(). During this call, the name parameter of the print function is a pointer to the name of the parents device. The print function should then print a message like "foo at bar". Usually, foo_attach would print this message, but since no driver for foo is included in the kernel, there is no foo_attach function. The print then returns either UNCONF or UNSUPP . Either the message "not configured" or "unsupported" will be appended to the message printed by print accordingly. If the driver does exist (at least in principle), but was not compiled into the current kernel, UNCONF is returned; UNSUPP, if the parent detects a child and knows that there isn't a driver available for it. Another reason for the use of config_search() and foo_match() under direct configuration is explained in section 3.3.3. But that detail shall confuse use only lateron. ;-)

Any questions? Hell yeah! Where exactly does the driver call config_found() or config_search() from? Well, that's easy: in its own foo_attach() function. foo_attach() initializes the driver, and looking for your children is part of the initialization.

All these pieces fall into place if, armed with this knowledge, you venture into the kernel source code and take a close look at the autoconf(9) interface as well as the involved "files.*" files. Also of interest is sys/kern/subr_autoconf.c, as well as the config_search() (+mapply()), config_found_sm() and config_attach() functions (you don't need to completely understand config_attach() just yet). These few lines of code in sys/kern/subr_autoconf.c are what it's all about.

2.6 bus_space(9) and bus_dma(9)

As mentioned above, one and the same driver core can be attached to different busses. In this special case, bus refers to a systembus, the addressspace of which can be mapped into the address space of the CPU, such as the QBus or PCI, for example. That means, the CPU is able to transfer data from and to the bus device using load and store operations. Not part of this category are mass memory-, desktop- and other busses like SCSI, HP-IB, ADB, USB, ... with which can only be accessed indirectly through a hostadapter, possibly using a packet oriented protocol.

The core of the driver uses only abstract functions to access the hardware using tags and handles. This makes it independent of the bus system. Part of this core of the driver may be several *bus-attachments*. These bus-attachments implement the `cfattach` and `softc` data structures described in `driver(9)` as well as the `foo_match()` and `foo_attach()` functions for each bus system separately. The `foo_attach()` function "frobs" the tags and handles for the driver core. `bus_space(9)` and `bus_dma(9)` represent the system of NetBSD that accesses the hardware independent of the utilized bus system through the use of abstract functions. If you write a driver, you get `bus_space(9)` and / or `bus_dma(9)` tags and handles back which are passed to the `foo_match()` and `foo_attach()` functions. But you don't get `bus_space(9)` / `bus_dma(9)` tags and handles directly, but rather an attach-`struct`, specific to the according bus, which contains the `bus_space(9)` / `bus_dma(9)` tags and handles as well as other bus-specific parameters.

The `bus_space(9)` / `bus_dma(9)` tag is the abstract representation of the bus-hierarchy of a machine, while the handle represents an abstract address of a bus within this hierarchy. The structure of these tags and handles largely depends on the architecture of the specific hardware. Bus drivers for QBus, ISA, ... convert the locator values from the kernel configuration file into the appropriate handles. The actual driver does not need to take care of the structure or the content of these tags and handles, but can simply make use of them.

Well, now it's about time to read `bus_space(9)` and `bus_dma(9)`. ;-)

3 The `autoconf(9)` part of the `rf(4)` driver

And here we go. We want to write a driver for the VAX. The `rf(4)` driver consists, from the point of view of `autoconf(9)`, of two drivers. One driver for the controller and one driver for the attached drives.

First a comment about the basic structure of the drivers source code: The code should begin with a copyright notice. The copyright of the source code has to be compatible with the BSD license. It therefore makes sense to use the BSD license or a BSD-like license. After that comes a comment with common notes, such as "This is the driver for blah, ...", TODO lists, known BUGS etc. Then the include statements, starting with the common, kernel include files up to the special include files that are only used by this driver. Next are the preprocessor directions such as macro definitions and symbolic constants, followed by the functions prototypes and the declaration of data structures and -types. See `/usr/share/misc/style` (also found in the appendix). This file explains the indentations rules, which *must* be followed for NetBSD source code. (If you ever want to see the code in the NetBSD CVS repository.)

3.1 Configuration files

3.1.1 The kernel configuration file

```
rfc0            at uba? csr 0177170     # RX01/02 controller
rf*             at rfc? drive?          # RX01/RX02 floppy disk drive
```

That's all we need to add to the kernels configuration file to activate the driver. The `csr` is the locator of the UniBus / QBus. It is given in octal and represents the address of the device under which it identifies itself on the QBus.

3.1.2 `sys/dev/qbus/files.uba`

Then we need to enter our driver and the files that implement it in `sys/dev/qbus/files.uba`[5]:

```
# RX01/02 floppy disk controller
device  rfc { drive=-1 }
attach  rfc at uba
```

[5] `files.uba` instead of `files.qbus`, as the UniBus existed before the QBus.

```
device  rf: disk
attach  rf at rfc
file    dev/qbus/rf.c                 rf | rfc needs-flag
```

The first line announces to `config(8)`, that there is a driver called `rfc`. Since other devices may attach to `rfc`, and it thus represents an interface attribute, we tell `config(8)` to use the locator `drive`. Since the `rfc` driver supports direct configuration, it makes sense to allow wildcards, we pass a standard value of -1 to the locator. (See 2.4 as well as -1.)

The second line informs `config(8)`, that `rfc` attaches to the `uba` bus (=interface attribute). Accordingly, the fourth line describes the connection between `rf` and `rfc`.

The third line is a little bit more interesting. This line defines the `rf` driver and associates it with the `disk` attribute[6].

The last line finally makes sure that the file `sys/dev/qbus/rf.c` is compiled into the kernel if one of the `rfc` and / or `rf` attributes is found in the current kernel configuration. `needs-flag` assures that the file `rf.h` is created by `config(8)` in the compilation directory. This file contains `#define NRF 1` and `#define NRFC 1` if the `rf(4)` driver is compiled into the kernel and `#define NRF 0` and `#define NRFC 0` if it isn't. These preprocessor constants can be used by other parts of the source code if they depend on the (non-) existence of the driver.[7]

3.1.3 The device numbers

As all of you certainly know, device nodes, which appear as files in the file system, are a way for the userland processes to access the hardware, the drivers in the kernel. The kernel tells the device nodes apart by their major- and minor device numbers. Each driver, that supports device nodes has an individual major device number, by which it is identified.

The kernel needs to have a table telling it which driver maps to which major device number. (The minor device number is handled by the driver itself.) There are separate tables for character- and block devices, so that the major device number of a character device can be different from one of a block device.

[6]Actually, `disk` is a "*device class*", whatever the precise difference between an attribute and a "device class" may be.

[7]Up until NetBSD 1.6-release these preprocessor constants are also necessary for handling the Majordevice numbers of the character- and block device nodes. Therefore, `needs-flag` is absolutely necessary in those versions.

Since NetBSD 2.0-current, these tables are created automatically by `config(8)` in the file `devsw.c` inside the kernels compilation directory.[8] The list of major device numbers in `sys/arch/<arch>/conf/majors.<arch>` is used as a template, where `<arch>` represents the machine- / processor architecture, `vax` in our case. Therefore, the following line in `sys/arch/vax/conf/majors.vax` is necessary to get the major device numbers:

```
device-major   rf              char 78  block 27       rf
```

The numbers 78 and 27 are used, since the last available entry used the numbers 77 and 26 respectively.

The file `devsw.c` contains the two tables `bdevsw` and `cdevsw`[9]. The index of this table is the major device number. Each line in these tables conforms to a device and contains a pointer to the `struct bdevsw` data structure (or the `struct cdevsw` data structure) declared in the driver, which contains several function pointers depending on the device type:

```
const struct bdevsw rf_bdevsw = {
        rfopen,
        rfclose,
        rfstrategy,
        rfioctl,
        rfdump,
        rfsize,
        D_DISK
};

const struct cdevsw rf_cdevsw = {
        rfopen,
        rfclose,
        rfread,
        rfwrite,
        rfioctl,
        nostop,
        notty,
        nopoll,
```

[8] Up until NetBSD 1.6-release, these tables are located in `sys/arch/<arch>/<arch>/conf.c` and need to be maintained by hand.

[9] BlockDEViceSWitch and CharacterDEViceSWitch respectively. Both these tables have been available under the same name in UNIX V6 since 1976. See [Li 77]

```
        nommap,
        nokqfilter,
        D_DISK
};
```

These functions implement the different operations possible with the device in question, such as `open, close, write, ioctl...`. If a driver does not implement one of these functions, it writes `no<functionname>` in its place, for example `nommap`. The last field in the `cdevsw` or `bdevsw` data structure is the device type. Currently, the following types are available: `D_DISK, D_TAPE, D_TTY`. Block devices are understandably always of type `D_DISK`. The device type determines which functions a driver has to implement:

D_DISK: `open, close, read, write, ioctl`

D_TAPE: `open, close, read, write, ioctl`

D_TTY: `open, close, read, write, ioctl, stop, tty, poll`

`D_DISK` and `D_TAPE` therefore do not need to provide `stop, tty, poll`. (Question to all gurus: What about drivers, which also implement mmap(2)?) As we will see below, there are preprocessor macros to simplify the declaration of these functions. All these macros and the prepared data structures can be found in `sys/sys/conf.h`.

```
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);
```

3.2 Data structures for `autoconf(9)`

As mentioned in `driver(9)`, the kernel uses a static `struct`, through which the functions necessary for `autoconf(9)` are included. Since NetBSD 2.0-current, this declaration is done through a macro `CFATTACH_DECL`. Since the `rf(4)` driver doesn't need the "detach" and "activate" functions, you simply pass `NULL`-pointers in their place.

```
CFATTACH_DECL(
        rfc,
        sizeof(struct rfc_softc),
        rfc_match,
        rfc_attach,
        NULL,
        NULL
);

CFATTACH_DECL(
        rf,
        sizeof(struct rf_softc),
        rf_match,
        rf_attach,
        NULL,
        NULL
);
```

And finally the `struct`, through which the `rfc` parent tells its `rf` child at `autoconfig(9)` time its actual found "bus address". More on this under section 3.3.3, when we talk about `rf_match`.

```
struct rfc_attach_args {
        u_int8_t type;          /* controller type, 1 or 2 */
        u_int8_t dnum;          /* drive number, 0 or 1 */
};
```

3.3 Functions for `autoconf(9)`

3.3.1 `rfc_match()`

A parent device hands down a bus- or controller specific `attach_args` data structure to its child devices in the `void *aux` parameter. These data inform the child device driver "where" on the bus it should look for a device. In extensible busses such as the QBus, this data structure also contains the `bus_space(9)` handles. The driver can access the hardware only by means of these handles through the `bus_space(9)` functions / macros. That is why one of the first actions in one of the "match" or "attache" routines is a typecast of the `void *aux` parameter to the appropriate `attach_args` data structure.

In order to communicate with the controller, it maps a two byte wide "registers" (the so-called command and status registers) into the address space of the bus.[10] As the name suggests, it is possible to send a specific command to the controller by sending a certain combination of bits via `bus_space_write_2(9)` or to query its status via `bus_space_read_2(9)`. Depending on the command, multiple writes may be necessary to provide all parameters such as sector number, etc.

```
int
rfc_match(struct device *parent, struct cfdata *match, void *aux)
{
        struct uba_attach_args *ua = aux;
        int i;

        /* Issue reset command. */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS, RX2CS_INIT);
        /* Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set. */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & RX2CS_DONE) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & (RX2ES_RDY | RX2ES_ID)) != 0)
                        break;
                DELAY(100000);  /* wait 100ms */
        }
        /*
         * Give up if the timeout has elapsed
         * and the controller is not ready.
         */
        if (i >= 20)
                return(0);
        /*
         * Issue a Read Status command with interrupt enabled.
         * The uba(4) driver wants to catch the interrupt to get the
         * interrupt vector and level of the device
         */
```

[10] Remember the name of the locator for the UniBus / QBus from the kernels configuration file? It's "csr".

```
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS,
            RX2CS_RSTAT | RX2CS_IE);
        /*
         * Wait for command to finish, ignore errors and
         * abort if the controller does not respond within the timeout
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & (RX2CS_DONE | RX2CS_IE)) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & RX2ES_RDY) != 0 )
                        return(1);
                DELAY(100000);  /* wait 100ms */
        }
        return(0);
}
```

First we send a reset to the controller and wait up to two seconds if it acknowledges the command. Should the controller end the reset properly, a second command with an interrupt enable bit is sent. Why are we using an interrupt, if `driver(9)` states that the entire `autoconfig(9)` procedure takes place when no interrupts have been enabled yet? Well, this only means that a driver can not yet use any interrupts, since the interrupt handling of the kernel has not yet been initialized. A device can, however, cause an interrupt nonetheless, it will just remain in the depth of the hard-/software. The interrupt gets lost, the interrupt handler is not called. We'll cover interrupts in more details lateron.

In this case, this is actually necessary. A driver for a QBus device *has to* cause an interrupt in its `foo_match()` function. This interrupt is caught by the QBus bus driver and is the only possibility for it to determine the interrupt level and -vector of the device. Therefore, the QBus bus driver does give an error message at boot time, if a `foo_match()` function indicates the presence of a device but no interrupt has taken place. (All of this is handled in the function `sys/dev/qbus/uba.c:ubasearch()`, which is the `func` parameter of `config_search` in the QBus bus driver. See 2.5.)

3.3.2 rfc_attach()

The attach_args data structure is valid only temporarily at autoconfig(9) time. Therefore, the driver lateron copies the required information from attach_args data structure into its softc data structure (more on this in the next chapter) and initializes the variables only lateron. Part of this is also the reservation of the appropriate resources such as the “DMA map” and to detach the interrupt handler. What and how exactly all this is done, does of course depend very much on the supported device and the bus-/controller to which it attaches.

```
void
rfc_attach(struct device *parent, struct device *self, void *aux)
{
        struct rfc_softc *rfc_sc = (struct rfc_softc *)self;
        struct uba_attach_args *ua = aux;
        struct rfc_attach_args rfc_aa;
        int i;

        rfc_sc->sc_iot = ua->ua_iot;
        rfc_sc->sc_ioh = ua->ua_ioh;
        rfc_sc->sc_dmat = ua->ua_dmat;
        rfc_sc->sc_curbuf = NULL;
        /* Tell the QBus busdriver about our interrupt handler. */
        uba_intr_establish(ua->ua_icookie, ua->ua_cvec, rfc_intr, rfc_sc,
            &rfc_sc->sc_intr_count);
        /* Attach to the interrupt counter, see evcnt(9) */
        evcnt_attach_dynamic(&rfc_sc->sc_intr_count, EVCNT_TYPE_INTR,
            ua->ua_evcnt, rfc_sc->sc_dev.dv_xname, "intr");
        /* get a bus_dma(9) handle */
        i = bus_dmamap_create(rfc_sc->sc_dmat, RX2_BYTE_DD, 1, RX2_BYTE_DD, 0,
            BUS_DMA_ALLOCNOW, &rfc_sc->sc_dmam);
        if (i != 0) {
                printf("rfc_attach: Error creating bus dma map: %d\n", i);
                return;
        }
```

Passing another reset to initialize the device at this point is a “Good Idea” (C) (R) (TM), since an “attach” routine must not rely on any “pre-requisites” of the “match” routine.

```
/* Issue reset command. */
bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, RX2CS_INIT);
/*
 * Wait for the controller to become ready, that is when
 * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set.
 */
for (i = 0 ; i < 20 ; i++) {
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
            & RX2CS_DONE) != 0
            && (bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2ES)
            & (RX2ES_RDY | RX2ES_ID)) != 0)
                break;
        DELAY(100000);  /* wait 100ms */
}
/*
 * Give up if the timeout has elapsed
 * and the controller is not ready.
 */
if (i >= 20) {
        printf(": did not respond to INIT CMD\n");
        return;
}
```

Ok, the controller has been found. After our `rfc_match()` function as announced the presence of a suitable device, the QBus driver will print a message like
`rfc0 at uba0 csr 177170 vec 264 ipl 17`
without a trailing `\n`. That way, our `rfc_attach()` is able to print more detailed information regarding the device. And that's exactly what we'll do first: determine if the device in question is a RX01 or a RX02, save that piece of information in the `softc` structure and print an appropriate message.

```
/* Is ths a RX01 or a RX02? */
if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
    & RX2CS_RX02) != 0) {
        rfc_sc->type = 2;
        rfc_aa.type = 2;
} else {
        rfc_sc->type = 1;
```

```
                rfc_aa.type = 1;
        }
        printf(": RX0%d\n", rfc_sc->type);
```

The last task remaining is to look for the children, i.e. to determine if, where and how any floppy drives are attached. Those are then integrated into the device tree by means of `config_found()`.

```
#ifndef RX02_PROBE
        /*
         * Both disk drives and the controller are one physical unit.
         * If we found the controller, there will be both disk drives.
         * So attach them.
         */
        rfc_aa.dnum = 0;
        rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
        rfc_aa.dnum = 1;
        rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
#else /* RX02_PROBE */
        /*
         * There are clones of the DEC RX system with standard shugart
         * interface. In this case we can not be sure that there are
         * both disk drives. So we want to do a detection of attached
         * drives. This is done by reading a sector from disk. This means
         * that there must be a formated disk in the drive at boot time.
         * This is bad, but I did not find an other way to detect the
         * (non)existence of a floppy drive.
         */
        if (rfcprobedens(rfc_sc, 0) >= 0) {
                rfc_aa.dnum = 0;
                rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,
                    rf_print);
        } else
                rfc_sc->sc_childs[0] = NULL;
        if (rfcprobedens(rfc_sc, 1) >= 0) {
                rfc_aa.dnum = 1;
                rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,
                    rf_print);
        } else
```

```
                rfc_sc->sc_childs[1] = NULL;
#endif /* RX02_PROBE */
        return;
}
```

3.3.3 rf_match()

Since the rfc driver supports direct configuration, is calls config_found() (and through it rf_match()) only if the device is without a doubt present. You may think to yourself: "No problem, the device is present. We can reduce the function rf_match to a measly return(1);", but you would be wrong.

```
int
rf_match(struct device *parent, struct cfdata *match, void *aux) {
    struct rfc_attach_args *rfc_aa = aux;

    if ( match->cf_loc[RFCCF_DRIVE] == RFCCF_DRIVE_DEFAULT ||
        match->cf_loc[RFCCF_DRIVE] == rfc_aa->dnum ) {
        return( 1);
    }
    return( 0);
}
```

Why this check? Or rather, *what* is being checked? The match data structure of type cfdata describes the autoconfig(9) parameter for this driver from the kernels configuration file. The cf_loc array of the cfdata data structure contains the value of the locators. The position of the locators in this array is given by config(8). In order to be able to access the value of a certain locator in this array, there are preprocessor constants which follow this naming convention: <ATTR>CF_<LOC>, with <ATTR> representing the name of the interface attribute to which the locator belongs, and <LOC> representing the desired locator. The rf driver attaches to the rfc interface attribute with the drive locator, i.e. RFCCF_DRIVE. This way, the driver can directory access the values of the drive locator given in the kernel configuration file. For example, if it contains:
rf0 at rfc0 drive 1
then the value of match->cf_loc[RFCCF_DRIVE] is 1. If the kernel configuration file does not assign a value to the locator, a wildcard is used, so that the value os set to the standard value given in the file files.uba. This standard value is

available as a preprocessor constant RFCCF_DRIVE_DEFAULT and in is our example -1. (See 3.1.2 and 2.4 for the reason.)

Thus, the driver gets the value of the locators from two sides of the interface attribute to which it attaches, i.e. the position, adress, ID, ... under which it is known to its parent. The one side from which the driver receives the value of the locator is the cf_loc array of the cfdata data structure. These values are the same as the ones give in the kernels configuration file; they are statically given at compilation time. The other side is the attach_args data structure passed to the rfmatch function as void *aux. That structure is passed from the parent to the child when the kernel boots. It therefore contains the locator values, that are actually found in the hardware.

In the above if conditional, the child is checking two conditions: first, whether or not a wildcard was given as the locator. If so, then the position, address, ID, ... under which it is known to the parent matches, it returns 1 and the child driver is attached. If the kernel configuration file did not contain a precise position, address, ID, ... for the child, it has to use the match function to check if the value of the configuration file is identical to the one actually found. If so, then all's peachy and the child driver is attached. If not, then the found hardware configuration does not match the kernel configuration an the child driver must not be attached. Therefore, the match function returns 0. This check is necessary to "nail down" certain locators in the kernels configuration file. For example:
sd2 at scsibus1 target 6 lun 0

Using this mechanism, it is possible to on the one hand use wildcards and at the same time to "nail down" certain instances of a driver to a specific position. That is another reason why config_found() calls config_search(). config_search() iterates over all children. That's the only way to find the "right" child, if no wildcards are used.

3.3.4 rf_attach()

```
void
rf_attach(struct device *parent, struct device *self, void *aux)
{
        struct rf_softc *rf_sc = (struct rf_softc *)self;
        struct rfc_attach_args *rfc_aa = (struct rfc_attach_args *)aux;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;
```

```
rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
rf_sc->sc_dnum = rfc_aa->dnum;
rf_sc->sc_state = 0;
rf_sc->sc_open = 0;
rf_sc->sc_disk.dk_name = rf_sc->sc_dev.dv_xname;
rf_sc->sc_disk.dk_driver = &rfdkdriver;
disk_attach(&rf_sc->sc_disk);
dl = rf_sc->sc_disk.dk_label;
```

From an `autoconf(9)` view, this is nothing special aside from the initialization of the `softc` and other data structures. The first assignment shows how a child device can access the `softc` data structure of its parent. `disk_attach(9)` and the following assignments initialize the disklabel.

4 The core of the driver

In this chapter we'll cover the actual driver core, i.e. the functions and data structures that the driver has to provide to the kernel in order for data to be able to be transferred from and to the hardware.

4.1 Data structures of the driver

4.1.1 Data structures per controller

```
struct rfc_softc {
        struct device sc_dev; /* common device data */
        struct device *sc_childs[2]; /* child devices */
        struct evcnt sc_intr_count; /* Interrupt counter for statistics */
        struct buf *sc_curbuf; /* buf that is currently in work */
        bus_space_tag_t sc_iot; /* bus_space IO tag */
        bus_space_handle_t sc_ioh; /* bus_space IO handle */
        bus_dma_tag_t sc_dmat; /* bus_dma DMA tag */
        bus_dmamap_t sc_dmam; /* bus_dma DMA map */
        caddr_t sc_bufidx; /* current position in buffer data */
        int sc_curchild; /* child whose bufq is in work */
        int sc_bytesleft; /* bytes left to transfer */
        u_int8_t type; /* controller type, 1 or 2 */
};
```

sc_dev always has to be the first field of the `softc` data structure. This is required by `autoconf(9)`.

sc_childs contains pointers to the two possible child devices. (Each controller can manage two drives at most.)

sc_intr_count The interrupt caller. Mainly used for statistical purposes and is not strictly necessary for the driver to function but should always be used.

sc_iot, sc_ioh, sc_dmat, sc_dmah , , , are the `bus_space(9)` and `bus_dma(9)` tags and handles used by the driver to access the hardware.

sc_bufidx A buffer may be larger than a single sector, so a driver needs to remember at which point in the buffer it currently is.

`sc_curchild` Contains the number of the child device, currently being worked on by the controller.

`sc_bytesleft` How many bytes are left in the buffer and have to be transferred from / to the floppy.

`type` 1 or 2, depending on if it's a RX01 or a RX02.

4.1.2 Data structure per drive

```
#define RFS_DENS        0x0001    /* single or double density */
#define RFS_AD          0x0002    /* density auto detect */
#define RFS_NOTINIT     0x0000    /* not initialized */
#define RFS_PROBING     0x0010    /* density detect / verify started */
#define RFS_FBUF        0x0020    /* Fill Buffer */
#define RFS_EBUF        0x0030    /* Empty Buffer */
#define RFS_WSEC        0x0040    /* Write Sector */
#define RFS_RSEC        0x0050    /* Read Sector */
#define RFS_SMD         0x0060    /* Set Media Density */
#define RFS_RSTAT       0x0070    /* Read Status */
#define RFS_WDDS        0x0080    /* Write Deleted Data Sector */
#define RFS_REC         0x0090    /* Read Error Code */
#define RFS_IDLE        0x00a0    /* controller is idle */
#define RFS_CMDS        0x00f0    /* command mask */
#define RFS_OPEN_A      0x0100    /* partition a open */
#define RFS_OPEN_B      0x0200    /* partition b open */
#define RFS_OPEN_C      0x0400    /* partition c open */
#define RFS_OPEN_MASK   0x0f00    /* mask for open partitions */
#define RFS_OPEN_SHIFT  8         /* to shift 1 to get RFS_OPEN_A */
#define RFS_SETCMD(rf, state)   ((rf) = ((rf) & ~RFS_CMDS) | (state))


struct rf_softc {
        struct device sc_dev; /* common device data */
        struct disk sc_disk; /* common disk device data */
        struct bufq_state sc_bufq; /* queue of pending transfers */
        int sc_state; /* state of drive */
        u_int8_t sc_dnum; /* drive number, 0 or 1 */
};
```

sc_dev see above.

sc_disk Each device of type `disk` needs to have this data structure in their data structure. The kernel needs this to manage the disk drive.

sc_bufq The buffer queue of the drive. More on this later.

sc_state The driver uses this variable to remember the state of the drive in oder to in guarantee the proper sequence of initialized, initializing, idle, data transfer, writing sector, idle, ... Symbolic definitions accordingly.

sc_dnum Is this the first or the second drive on this controller?

4.2 The necessary functions

```
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);
```

The macros above make sure that the kernel requires the following functions when being linked:

```
int rfopen(dev_t dev, int oflags, int devtype, struct proc *p);
int rfclose(dev_t dev, int fflag, int devtype, struct proc *p);
int rfread(dev_t dev, struct uio *uio, int ioflag);
int rfwrite(dev_t dev, struct uio *uio, int ioflag);
int rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p);
void rfstrategy(struct buf *bp);
int rfdump(dev_t dev, daddr_t blkno, caddr_t va, size_t size);
int rfsize(dev_t dev);
```

The `open, close, read, write, ioctl` functions directly relate to the functions from user-space of the same name. That is, if you call for example `open( "'/dev/rrf0c"', O_RDONLY, 0);` from a program, this will lread to a call of `rfopen()`. The `rfstrategy, rfdump, rfsize` functions are required by some kernel internal functions.

```
int rfc_sendcmd(struct rfc_softc *, int, int, int);
struct rf_softc* get_new_buf( struct rfc_softc *);
static void rfc_intr(void *);
```

These are some helper functions of the driver. The first is used to send a command to the controller, as the name suggests. The last one is the interrupt handler and, together with `rfstrategy()` provides the main functionality. The second is a helper function of the interrupt handler.

```
int rfcprobedens(struct rfc_softc *, int);
```

This is just a helper function for debugging purposes. Also see the comments in `rfc_attach`.

Now, before we get into in the annoying little details of each function, a short overview of the basic process flow within the driver: In order to be able to do anything with the device, we need to first call `rfopen()`. This always happens and is essential to the driver, since it's the only way for it to initialize itself. Note that the `open()` function of a driver may appear several times after another, for example in order to open different partitions of a disk or several ports on a serial multi-port card (with different minor device numbers) as well as to transfer data over a file handle while receiving instructions from another via `ioctl` (with the same minor device numbers). Either a process from user-space that opens the device node or the kernel itself (for example through `mount(2)`) initiates the `open()` function call.

Noteworthy: Each time a user or a kernel process opens the device node (for example through a `mount(2)` oder in the context of `RAIDframe(9)`) the `open()` function is called. But the `close()` function is not called until the last user of the device executes a `close()`. For example: Three processes, A, B, and C open one after another the same device node and keep it open, leading to three calls of the `open()` function. Process A closes the device node - the driver doesn't care. Process C closes the device node – still, the driver doesn't care. Only after process B closed the device node, the `close()` function call is made, since this process was the last one to have an open handle on the device node.

Data transfer from and to the hardware are handled by the `rfstrategy()` function. This is the drivers point of access for data transfer, provided by the driver to the kernel. On each call, `rfstrategy()` receives a pointer to a single *buffer*. These infamous buffers are used by the buffer cache and describe a block oriented data transfer, i.e. which data should be read from RAM at which position / address on which device, or what should be read from where in RAM.

The `strategy()` function itself usually does not perform any I/O operations, but performs a few checks and simply organizes the I/O operations. Basically, this boils down to sorting the buffers in the bufferqueue (thus *strategy*). Other routines then walk down the bufferqueues buffer by buffer. The number of these routines, what exactly they do and how they do it is very specific to each driver. The kernel does not tell the driver how to do this.

Typically, there exists at least one helper functions aside from `strategy()`: the interrupt handler. Compared to the CPU, I/O devices are rather slow. Therefore, you can't just simply wait until an I/O operation is finished. Instead, the driver initiates an operation and kernel does other things, such as dispatching the processing cycles etc. When the hardware has finished the I/O operation, it causes an interrupt. The "normal" program flow is thus interrupted by an electronic signal by the hardware and a special routine inside the kernel is called. This routine determines which piece of hardware has caused the interrupt and calls the interrupt handler responsible for the piece of hardware that caused the interrupt. The interrupt handler in turn finished the I/O operation by checking the error code of the hardware (for example a read error of the floppy), moving indices within the buffer, taking the next buffer out of the queue, etc.

4.2.1 `rfdump()`

... is used by the kernel to barf a core dump onto the disk or into swapspace, if it needs to abort. I think it's improbable that a floppy with 0.5MB capacity will be sufficient to cause this. Therefore, this function simply consists of `return( ENXIO);`. Also see `errno(2)`.

4.2.2 `rfsize()`

Gives back the (swap) partitions size in chunks of `DEV_BSIZE` size.

4.2.3 `rfopen()`

```
rfopen(dev_t dev, int oflags, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;
        int unit;
```

```
        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
```

First, rfopen() needs to determine if the device does actually exist, and if so, it needs to fetch the softc data structure for the device. The path from the device number to the softc data structure is, as you can see, relatively easy, once you know it. The DISKUNIT() macro uses the minor device number to determine which instance number of the device is addressed by the device number. See sys/disklabel.h. This is where the cfdriver data structure comes into play. For each device mentioned in the kernels configuration file, config(8) creates an instance of the data structur in ioconf.[ch] inside the kernel compilation directory, using a naming convention of <DEV>_cd. Here, <DEV> represents the name given in the kernel configuration file, rf in this case. cd stands for *C*onfiguration*D*river. The type definitions of this data structure can be found in sys/device.h. The cd_defs field of this data structure is an array of pointers, pointing to a softc data structure of an instance of the device. Since device may be added or removed during kernel runtime, this array is dynamic. The field cd_ndevs contains the total number of instances for the device (i.e. number of fields in cd_devs). Furthermore, there's cd_name, a pointer to a string with the name of the device and an enum describing the device class, cd_class.

```
        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        dl = rf_sc->sc_disk.dk_label;
        switch (DISKPART(dev)) {
                case 0:                 /* Part. a is single density. */
                        /* opening in single and double density is sensless */
                        if ((rf_sc->sc_state & RFS_OPEN_B) != 0 )
                                return(ENXIO);
                        rf_sc->sc_state &= ~RFS_DENS;
                        rf_sc->sc_state &= ~RFS_AD;
                        rf_sc->sc_state |= RFS_OPEN_A;
                break;
                case 1:                 /* Part. b is double density. */
                        /*
                         * Opening a singe density only drive in double
```

```
                 * density or simultaneous opening in single and
                 * double density is sensless.
                 */
                if (rfc_sc->type == 1
                    || (rf_sc->sc_state & RFS_OPEN_A) != 0 )
                        return(ENXIO);
                rf_sc->sc_state |= RFS_DENS;
                rf_sc->sc_state &= ~RFS_AD;
                rf_sc->sc_state |= RFS_OPEN_B;
        break;
        case 2:                 /* Part. c is auto density. */
                rf_sc->sc_state |= RFS_AD;
                rf_sc->sc_state |= RFS_OPEN_C;
        break;
        default:
                return(ENXIO);
        break;
}
```

Typically, you mark the open state in the `softc` data structure and block the manual ejection. This way, you can prevent multiple simultaneous accesses of the same device and remember if the device has been opened in a special mode. The `rf(4)` driver takes advantage of this by assigning different minor device numbers (i.e. partitions for a disk driver) to single or double write speeds or determining the write speed automatically from the format of the floppy disk.

```
if ((rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
        rfc_sc->sc_curchild = rf_sc->sc_dnum;
        /*
         * Controller is idle and density is not detected.
         * Start a density probe by issuing a read sector command
         * and sleep until the density probe finished.
         * Due to this it is impossible to open unformated media.
         * As the RX02/02 is not able to format its own media,
         * media must be purchased preformated. fsck DEC marketing!
         */
        RFS_SETCMD(rf_sc->sc_state, RFS_PROBING);
        disk_busy(&rf_sc->sc_disk);
```

```
                if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
                    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD),
                    1, 1) < 0) {
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
                /* wait max. 2 sec for density probe to finish */
                if (tsleep(rf_sc, PRIBIO | PCATCH, "density probe", 2 * hz)
                    != 0 || (rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
                        /* timeout elapsed and / or somthing went wrong */
                        rf_sc->sc_state = 0;
                        return(ENXIO);
                }
        }
```

The drive has not been initialized at the first open call, that is, a sector needs to be read from the floppy in order to determine if the inserted medium has the appropriate density (or in order to determine the actual density when using "auto density"). Therefore, the driver enters a `RFS_PROBING` state and reports the floppy disk as being "busy"[11]. Afterwards, the controller receives a command to read the sector and the driver waits for the (non) successful return of this command.

The problem with this: You can just loop around using `DELAY()` in `rfc_match()` (busy wait), which would mean that the entire machine is completely *frozen*. We are inside the kernel![12]

The proper solution to the problem are the functions `tsleep(9)` and `wakeup(9)`. `rfopen()` was caused by some process, and that process needs to wait until the operation has completed. Other processes can happily continue. `tsleep(9)` marks the process that caused the `rfopen()` operation as "sleeping" and tells the scheduler to allow the other processes to use the CPU until the operation has completed.

This completion then happens inside the interrupt handler. The controller, as we know, received a read-command via the interrupt enable (`RX2CS_IE`), meaning the controller causes an interrupt as soon as the command has completed, which in turn causes the call of the interrupt handler. The interrupt handler checks the

[11] Using `disk_busy` is not necessary strictly speaking, but certainly a nice thing to do and can be used to determine statistical data; see `iostat(8)`.

[12] In `rfc_match()`, this is feasible, as it only happens during boot.

result of the command, manipulates the state variable of the driver accordingly and tells the scheduler via `wakeup(9)`, that the operation has completed and the sleeping processes can be awakened. The next time the process gets any cycles on the CPU, it continues inside `rfopen()` at the same point at which `tsleep(9)` was called. Therefore, this is the spot at which we need to check whether or not the interrupt handler has determined a successful completion or if there was a timeout.

```
        /* disklabel. We use different fake geometries for SD and DD. */
        if ((rf_sc->sc_state & RFS_DENS) == 0) {
                dl->d_nsectors = 10;            /* sectors per track */
                dl->d_secpercyl = 10;           /* sectors per cylinder */
                dl->d_ncylinders = 50;          /* cylinders per unit */
                dl->d_secperunit = 501; /* sectors per unit */
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = 500;
        } else {
                dl->d_nsectors = RX2_SECTORS / 2;  /* sectors per track */
                dl->d_secpercyl = RX2_SECTORS / 2; /* sectors per cylinder */
                dl->d_ncylinders = RX2_TRACKS;     /* cylinders per unit */
                /* sectors per unit */
                dl->d_secperunit = RX2_SECTORS * RX2_TRACKS / 2;
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        }
        return(0);
}
```

A driver for a medium supporting partitions should make sure to read the `disklabel(5,9)`s at this point. Since we do not support `disklabel(5,9)` on RX01/02 floppies, we just create pseudo `disklabel(5,9)`.

4.2.4 `rfclose()`

```
int
rfclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;
```

```
        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfclose: can not close on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        else
                rf_sc->sc_state &= ~(1 << (DISKPART(dev) + RFS_OPEN_SHIFT));
        if ((rf_sc->sc_state & RFS_OPEN_MASK) == 0)
                rf_sc->sc_state = 0;
        return(0);
}
```

Closing a device that hasn't been opened before, is a serious problem, and causes a kernel panic. In the other case, we simply reset the bit that marks the partition as opened. The last if statement checks, if all partitions have been closed and if so, all the blocks created by `rfopen` need to be removed and the `softc` data structur needs to be reinitialized.

4.2.5 `rfread()` and `rfwrite()`

Consisting of:

```
return( physio( rfstrategy, NULL, dev, B_READ, minphys, uio));
bzw.
return( physio( rfstrategy, NULL, dev, B_WRITE, minphys, uio));
```

If a process wants to `read(2)` or `write(2)` data to the character device node, these system calls will resolve the mapping of the file descriptor to the device nide. The buffer given to `read(2)/write(2)` is separated by the system call until only an array of pointers to the pages in physical RAM are left (plus offset and length within the page). These pointers then end up in the `uio` structure. The `physio()` function changes these pointers to pages into file system buffers within the `uio` structure, as they are used by the buffer cache (`struct buf`) and calls `strategy()` to pass the file system buffers one by one to the driver. Also see `physio(9)` and `sys/kern/kern_physio.c`. Access of the character device do `not` use the buffer cache, but instead `physio(9)` will transfer the data immediately from and to the memory of the process and the driver.

4.2.6 rfstrategy()

As mentioned above, this function is the main part of the driver with respect to data transfer. It takes the buffers and fills or empties them, but doesn't perform any IO operations itself. Instead, `rfstrategy()` writes these IO requests into one or more queues. It's the responsibility of the driver to sort the queues in a way that reduces seeks of the read/write-head. The queues are emptied by the interrupt handler. Each time, the hardware has completed an IO operation, it causes an interrupt. The interrupt handler then registers it as completed and removes it from the queue. Then, it initiates the next IO operation waiting in the queue, ...

```
rfstrategy(struct buf *buf)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
        int i;

        i = DISKUNIT(buf->b_dev);
        if (i >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[i]) == NULL) {
                buf->b_flags |= B_ERROR;
                buf->b_error = ENXIO;
                biodone(buf);
                return;
        }
```

In order to get to the `rf_sc softc` data structure, we do the same dance as before, with the exception of the error handling. A buffer may be treated as a work order; different buffers are independent of one another. Remember, `frstrategy()` is the only point of entrance for all `rf(4)` instances. A buffer may receive a work order for the first drive and complete it successfully. Another buffer may be related to a drive that isn't physically present, so that it must fail. Or maybe one of the buffers may fail due to a broken sector but the rest of the buffers are not affected by this.

The point in time at which the buffer will be "done" is unknown. If the drive is not physically present, we can check for it (using the above if statement) when calling `rfstrategy()`. But it's possible that the buffer was sorted way back in the queue, in which case it will be worked on much later, after `rfstrategy()` has long since "eaten" the buffer and `return`ed. In other words, finishing a buffer is asynchronous to the `rfstrategy()` function, which is why the kernel needs to be signaled when a buffer is ready using biodone(9). If an error occurs, you set a flag and pass an appropriate error code accordingly (as we can see above).

```
rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
/* We are going to operate on a non open dev? PANIC! */
if ((rf_sc->sc_state & 1 << (DISKPART(buf->b_dev) + RFS_OPEN_SHIFT))
    == 0)
        panic("rfstrategy: can not operate on non-open drive %s "
            "partition %d", rf_sc->sc_dev.dv_xname,
            DISKPART(buf->b_dev));
```

The comment says all.

```
if (buf->b_bcount == 0) {
        biodone(buf);
        return;
}
```

A small optimization. If buf->b_bcount == 0, there's nothing to be done and we are done with the buffer immediately.

```
/*
 * BUFQ_PUT() operates on b_rawblkno. rfstrategy() gets
 * only b_blkno that is partition relative. As a floppy does not
 * have partitions b_rawblkno == b_blkno.
 */
buf->b_rawblkno = buf->b_blkno;
/*
 * from sys/kern/subr_disk.c:
 * Seek sort for disks.  We depend on the driver which calls us using
 * b_resid as the current cylinder number.
 */
i = splbio();
if (rfc_sc->sc_curbuf == NULL) {
        rfc_sc->sc_curchild = rf_sc->sc_dnum;
        rfc_sc->sc_curbuf = buf;
        rfc_sc->sc_bufidx = buf->b_un.b_addr;
        rfc_sc->sc_bytesleft = buf->b_bcount;
        rfc_intr(rfc_sc);
```

```
        } else {
                buf->b_resid = buf->b_blkno / RX2_SECTORS;
                BUFQ_PUT(&rf_sc->sc_bufq, buf);
                buf->b_resid = 0;
        }
        splx(i);
        return;
}
```

Let's assume, `rfstrategy()` receives three buffers shortly after one another. The first reads a sector from track 1, the seconds reads a sector from the last track and the thirds reads a sector from somewhere in the middle. If `rfstrategy()` blindly followed the order in which it received the buffers, it would need to move the read/write head first to track 1, then all the way across the disk to the last track and then back to the middle. But moving the read/write head is *slow*, i.e. "expensive". So we try to achieve minimal movement, which is why we want to sort the buffer queue such that we work the first buffer first, then the third and finally the second buffer. This way, reading the sector in the middle happens kind of "on the way". Fortunately, the driver doesn't need to care too much about sorting the buffers and leaves that to `BUFQ_PUT()`. `BUFQ_PUT()` expects the track- or cylinder number (depending on the disk geometry) in the `b_resid` field in order to optimize it according to the cylinder numbers. `BUFQ_PUT()` then enters the buffer into the buffer queue and sorts is appropriately.

The entire sorting process is, however, not necessary if the controller is currently idle. In that case, `rfc_sc->sc_curbuf == NULL`. As the name suggests, `rfc_sc->sc_curbuf` points to the buffer currently being worked on. If the controller is idle, then the buffer is pointed to the current one and all related variables within `softc` are initialized with it. If a new buffer arrives while the first is being worked on in `rfc_sc->sc_curbuf`, they are put in the buffer queue. Calling `rfc_intr()` initiates the actual data transfer. The rest is done in `rfc_intr()` in the context of an interrupt.

Interruptcontext, what's that exactly anyway, and are there other contexts? Simplified, there are three contexts under Unix:

Usercontext: A normal user process utilizes the CPU. The time is the same as "'xx% user"' under `top(1)` or `time(1)`.

Kernelcontext: A userland process has made a system call. Kernel code in privileged mode is executed, but still under the current process space. The time is the same as "'xx% system"' under `top(1)` or `time(1)`.

Interruptcontext: The hardware has caused an interrupt of the code execution while the CPU was in user- or kernelcontext. This interrupt has higher priority than the user- or kernelcontext. The current state of the CPU is saved and the interrupt handler called. The interrupt handler manages the hardware and reinstates the saved CPU state, continuing the code execution where it left off. User- and kernel code do not know anything about this.

The alert reader already will have noticed a problem here: an interrupt is an asynchronous event, which may happen at any time. For example, at the time that `rfstrategy()` manipulates the bufferqueue, adding a new buffer. But the interrupt handler of the `rf(4)` driver also manipulates the buffer queue (it removes a buffer). If this happens, then the buffer queue ends up in an inconsistent state. Booom, kernel panic, game over.

This is why `rfstrategy()` has to use `splbio(9)` in order to assure that no interrupt can occur at this time. This function blocks all interrupts until they are released by a call to `splx(9)`. The time, during which interrupts are blocked should always be kept to a minimum, since not only interrupts for this particular device are blocked, but *all* interrupts! If interrupt blocks are prolonged, an interrupt for a different device might be delayed, reducing the I/O rate.

To be more precise: There are different interrupt priorities. `IPL_BIO`, for example, is rather low and is used by block devices such as disks and floppies. These devices are slow anyway and usually have bigger buffers in their hardware on the controller. So it doesn't matter, if the device waits a little bit longer on its interrupts. Network cards use `IPL_NET`, which has higher priority over `IPL_BIO`. Network cards are "faster" than disks (no mechanics) and buffer overflows in a network card are more expensive, as they can cause TCP retries and thus increase the network load and decrease throughput. These priorities make it possible for an interrupt of a network card to, well, interrupt the interrupt handler of the disk driver, have the network cards driver handle the interrupt and the continue with the interrupt handler of the disk driver.

So, it is not only important to block interrupts only for a short period of time, but also with the right priority. If the priority is too low, then your own interrupt handler can't take the interrupt, if it's too high, other devices throughput may suffer. For more information on the relationship among the different priorities, see `spl(9)`.

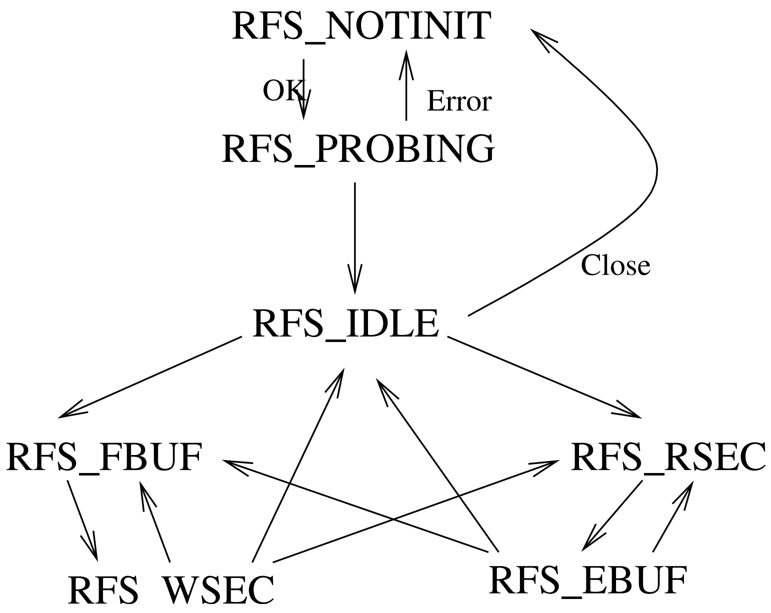

Figure 3: `rf(4)`'s interal states.

4.2.7 `rfc_intr()`

Most of the work of the driver, such as filling the buffer and working through the buffer queue is done within `rfc_intr()`. In order to perform an operation, the driver has to go through a sequence of states. After the first `open()` call, the state is set from `RFS_NOTINIT` to `RFS_PROBING` until the mediums capacity has been determined. Then, the driver enters the `RFS_IDLE` state until a buffer containing a read or write command arrives.

In order to read or write a sector, two commands needs to be sent to the controller. First, a read command `RX2CS_RSEC` (ReadSECtor) to read the internal controller buffer, then the DMA-command `RX2CS_EBUF` (EmptyBUFfer) to transfer the data from the controller into the RAM. If not all data has been transferred from the buffer queue, the driver sends the next `RX2CS_RSEC` command, then `RX2CS_EBUF`, and so on until the bufferqueue has been emptied. Similarly, when writing data they first need to be transferred from RAM into the internal controller buffer via DMA `RX2CS_FBUF` (FillBUFfer) so they can then be written to the media with a second command `RX2CS_WSEC` (WriteSECtor) until the next `RX2CS_FBUF` command ... until the next buffer from the buffer queue arrives.

These commands correspond to the `RFS_RSEC`, `RFS_EBUF`, `RFS_FBUF`, `RFS_WSEC` states. The four other commands `RX2CS_SMD`, `RX2CS_RSTAT`, `RX2CS_WDDS`, `RX2CS_REC` and their states are not used by the driver at this point. `RFS_RSEC` does not always have to follow `RFS_EBUF`. If the buffer has been finished and the buffer queue is empy, then the driver enters the `RFS_IDLE` state. if the buffer queue is not empty, but a new buffer containing data that needs to be written follows, a transition from `RFS_EBUF` to takes place. Similarly, there are transitions from `RFS_WSEC` to `RFS_IDLE` or `RFS_RSEC`. The `frc_sendcmd` function simplifies the sending of commands a little bit.

Due to the length of this function, we only give an explanation of the basics and some excerpts with special meaning: The functions consists of tw `switch` statements. The first takes care of finding the last command / state and finish the last operation. The second initializes the next command. Both `switch` statements reside within a loop, which is aborted if a new command has successfully been sent to the controller or the buffer queue is found to be empty. If an error occurred, the interrupt handler `continue`s at the beginning of the loop and tries to work the next buffer. (Some other drivers contains a `goto`, but since I prefer spaghetti over spaghetti code, I chose the loop.)

The following are the excerpts of the program that might be encountered when performing a read starting with the `RFS_IDLE` state, beginning with the `get_new_buf()` helper function.

```
struct rf_softc*
get_new_buf( struct rfc_softc *rfc_sc)
{
    struct rf_softc *rf_sc;
    struct rf_softc *other_drive;

    rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
    rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
    if (rfc_sc->sc_curbuf != NULL) {
        rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
        rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
    } else {
        RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
        other_drive = (struct rf_softc *)
            rfc_sc->sc_childs[ rfc_sc->sc_curchild == 0 ? 1 : 0];
        if (other_drive != NULL
```

```
            && BUFQ_PEEK(&other_drive->sc_bufq) != NULL) {
            rfc_sc->sc_curchild = rfc_sc->sc_curchild == 0 ? 1 : 0;
            rf_sc = other_drive;
            rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
            rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
            rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
        } else
            return(NULL);
    }
    return(rf_sc);
}
```

This function is called by `rfc_intr` if a buffer is finished in order to receive the next buffer. The controller manages two drives. First, this function checks if another buffer waits in the buffer queue of the current drive (`rfc_sc->sc_curchild`). If so, the next buffer in this queue is sued. If the buffer queue is empty, the drive is marked as idle and the buffer queue of the next drive (`other_drive`) is checked. If no buffers are available, no further action takes place. If there are any buffers, the function switches the current drive (`rfc_sc->sc_curchild`) and initializes the next buffer. The return value of the function is `NULL`, if both buffer queues are empty. Otherwise, it's a pointer to the `softc` structure of the drive whose buffer queue is being worked on.

```
        /* first switch statement */
        case RFS_IDLE:  /* controller is idle */
            if (rfc_sc->sc_curbuf->b_bcount
                % ((rf_sc->sc_state & RFS_DENS) == 0
                ? RX2_BYTE_SD : RX2_BYTE_DD) != 0) {
                /*
                 * can only handle blocks that are a multiple
                 * of the physical block size
                 */
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            }
            RFS_SETCMD(rf_sc->sc_state, (rfc_sc->sc_curbuf->b_flags
                & B_READ) != 0 ? RFS_RSEC : RFS_FBUF);
            break;
```

The if-statement is a security check. The RFS_SETCMD macro simplifies setting the rf_sc->sc_state variable. Depending on whether the current buffer contains a read or a write command, it contains RFS_RSEC or RFS_FBUF.

```
/* second switch statement */
case RFS_RSEC:  /* Read Sector */
    i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
        + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
        ((rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD);
    if (i > RX2_TRACKS * RX2_SECTORS) {
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        break;
    }
    disk_busy(&rf_sc->sc_disk);
    if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
        | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
        | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
        i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
        disk_unbusy(&rf_sc->sc_disk, 0, 1);
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
    }
    break;
```

The first instructions computes the logical block number to be read by the floppy and stores it in i. Note that the RX02 drive does not use the usual 512 (DEV_BSIZE, the value of buf->b_blkno), but 128 Bytes per sector (RX2_BYTE_SD) in single and 256 Bytes per sector (RX2_BYTE_DD) in double density. The if-statement checks of the sector requested by the buffer is higher than the capacity of the floppy and if so, sets the error flag in the buffer and aborts the operation. See the explanation in 4.2.3 for details regarding disk_busy().

Well, and finally the controller receives the command to read a sector (RX2CS_RSEC) with the interrupt bit RX2CS_IE enabled via rfc_sendcmd(). If this fails, then the error handler a the end of the loop around the two switch statements jumps in:

The rfc_intr() function checks if the error flag has been set after each of the two switch statements and brings the driver into a defined state in the case of an error:

```
        if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
            /*
             * An error occured while processing this buffer.
             * Finish it and try to get a new buffer to process.
             * Return if there are no buffers in the queues.
             * This loops until the queues are empty or a new
             * action was successfully scheduled.
             */
            rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
            rfc_sc->sc_curbuf->b_error = EIO;
            biodone(rfc_sc->sc_curbuf);
            rf_sc = get_new_buf( rfc_sc);
            if (rf_sc == NULL)
                return;
            continue;
        }
```

Ok, the interrupt handler has finished and we can return to our old context... until the controller has finished the command an causes an interrupt. Then we continue with the interrupt handler of the `rf(4)` driver:

```
/* first switch statement */
case RFS_RSEC:  /* Read Sector */ disk_unbusy(&rf_sc->sc_disk, 0, 1);
    /* check for errors */
    if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS) &
RX2CS_ERR) != 0) {
/* should do more verbose error reporting */
printf("rfc_intr: Error reading sector: %x\n",
bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2ES) );
rfc_sc->sc_curbuf->b_flags |= B_ERROR; } RFS_SETCMD(rf_sc->sc_state,
RFS_EBUF); break;
```

First we need to tell the kernel that the drive is no longer busy, that so far 0 Bytes have been transferred and that we received a read command. The if statement checks the error flag (`RX2CS_ERR`) in the CSR of the controller and aborts the process if necessary. At this point, we could use the `RX2CS_RSTAT` and `RX2CS_REC` commands to add some more detailed error diagnostics.

```
        /* second switch statement */
```

```
    case RFS_EBUF:  /* Empty Buffer */
        i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
            rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD,
            rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
        if (i != 0) {
            printf("rfc_intr: Error loading dmamap: %d\n",
            i);
            rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            break;
        }
        disk_busy(&rf_sc->sc_disk);
        if (rfc_sendcmd(rfc_sc, RX2CS_EBUF | RX2CS_IE
            | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
            | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
            | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
            & 0x30000) >>4), ((rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
            rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
            disk_unbusy(&rf_sc->sc_disk, 0, 1);
            rfc_sc->sc_curbuf->b_flags |= B_ERROR;
            bus_dmamap_unload(rfc_sc->sc_dmat,
            rfc_sc->sc_dmam);
        }
        break;
```

An upcoming DMA-transfer needs to be registered with the `bus_dma(9)` system. The UniBus has an 18 bit address space, and the QBus has a 22 bit address space (at least on VAXen). That means that a UniBus / QBus device can not fill the entire address space of the CPU, but only a part of it, similar to the ISA Bus with its 24 bit address space. But since the people working at DEC used their brains when they created the VAX, they found a solution to this problem (much in contrast to the IBM technicians responsible for the PeeCee/AT). Inside the bus adapter of the UniBus / QBus, between UniBus / QBus address space and CPU address space, there is a MMU. This MMU can be programmed such that any UniBus / QBus address can be translated into any CPU address. This way, a UniBus / QBus device can fill the entire CPU adress space via the Busmaster-DMA, assuming that the bus adapter MMU has been programmed correctly. If there is no such MMU inside the bus adapter, then we are stuck using a "bounce buffer", as under ISA (see [Tho]). But we do not need to care about all this when writing a driver, `bus_dmamap_load(9)` takes care of this.

The driver can register the floppy as busy as long as the `bus_dma(9)` system hands a DMA map to the driver and the actual DMA operation can be initiated using the `RX2CS_EBUF` command. And once again, we wait for the next interrupt...

```
/* first switch statement */
case RFS_EBUF:  /* Empty Buffer */
    i = (rf_sc->sc_state & RFS_DENS) == 0
        ? RX2_BYTE_SD : RX2_BYTE_DD;
    disk_unbusy(&rf_sc->sc_disk, i, 1);
    bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
    /* check for errors */
    if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
        RX2CS) & RX2CS_ERR) != 0) {
        /* should do more verbose error reporting */
        printf("rfc_intr: Error while DMA: %x\n",
            bus_space_read_2(rfc_sc->sc_iot,
            rfc_sc->sc_ioh, RX2ES));
        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        break;
    }
```

This call to `rfc_intr()` ends the transfer. `disk_unbusy(9)` tells the guys is statistics how many bytes have been read, the DMA map is freed and the usual error checks are made.

```
    if (rfc_sc->sc_bytesleft > i) {
        rfc_sc->sc_bytesleft -= i;
        rfc_sc->sc_bufidx += i;
```

The buffer is not quite empty yet, so we need to advance the pointer and prepare the next `RFS_RSEC` command...

```
    } else {
        biodone(rfc_sc->sc_curbuf);
        rf_sc = get_new_buf( rfc_sc);
        if (rf_sc == NULL)
```

```
                return;
        }
        RFS_SETCMD(rf_sc->sc_state,
            (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
            ? RFS_RSEC : RFS_FBUF);
        break;
```

Ah! The buffer has successfully and completely been finished, so we can tell the rest of the kernel via `biodone(9)` so. Then we need to check if other buffers are waiting in the buffer queue and if so, work on those. If there are no more buffers in the queue for this drive, set it to idle and switch to the other drive, in case new buffers have been added to that drives queue, while we were busy working on the first drive.

4.2.8 `rfioctl()`

This function is the entrance point for the `ioctl(2)` calls of the device. In this case, only die IOCTLs to read the `disklabel(5,9)` are absolutely necessary, all other IOCTLs are not required or do not make sense in this driver. The RX02 drive can not be blocked by the software, the medium can not be ejected, the hardware is incapable of performing a low-level format...

```
int
rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        /* We are going to operate on a non open dev? PANIC! */
        if (rf_sc->sc_open == 0) {
                panic("rfstrategy: can not operate on non-open drive %s (2)",
                    rf_sc->sc_dev.dv_xname);
        }
        switch (cmd) {
```

```
        /* get and set disklabel; DIOCGPART used internally */
        case DIOCGDINFO: /* get */
                memcpy(data, rf_sc->sc_disk.dk_label,
                    sizeof(struct disklabel));
                return(0);
        case DIOCSDINFO: /* set */
                return(0);
        case DIOCWDINFO: /* set, update disk */
                return(0);
        case DIOCGPART:  /* get partition */
                ((struct partinfo *)data)->disklab = rf_sc->sc_disk.dk_label;
                ((struct partinfo *)data)->part =
                    &rf_sc->sc_disk.dk_label->d_partitions[DISKPART(dev)];
                return(0);

        /* do format operation, read or write */
        case DIOCRFORMAT:
        break;
        case DIOCWFORMAT:
        break;

        case DIOCSSTEP: /* set step rate */
        break;
        case DIOCSRETRIES: /* set # of retries */
        break;
        case DIOCKLABEL: /* keep/drop label on close? */
        break;
        case DIOCWLABEL: /* write en/disable label */
        break;

/*      case DIOCSBAD: / * set kernel dkbad */
        break; /* */
        case DIOCEJECT: /* eject removable disk */
        break;
        case ODIOCEJECT: /* eject removable disk */
        break;
        case DIOCLOCK: /* lock/unlock pack */
        break;
```

```
        /* get default label, clear label */
        case DIOCGDEFLABEL:
        break;
        case DIOCCLRLABEL:
        break;
        default:
                return(ENOTTY);
        }

        return(ENOTTY);
}
```

A rf.c

```
/*
 * Copyright (c) 2002 Jochen Kunz.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of Jochen Kunz may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY JOCHEN KUNZ
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL JOCHEN KUNZ
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/*
TODO:
- Better LBN bound checking, block padding for SD disks.
- Formating / "Set Density"
- Better error handling / detaild error reason reportnig.
*/

/* autoconfig stuff */
#include <sys/param.h>
```

```
#include <sys/device.h>
#include <sys/conf.h>
#include "locators.h"
#include "ioconf.h"

/* bus_space / bus_dma */
#include <machine/bus.h>

/* UniBus / QBus specific stuff */
#include <dev/qbus/ubavar.h>

/* disk interface */
#include <sys/types.h>
#include <sys/disklabel.h>
#include <sys/disk.h>

/* general system data and functions */
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/ioccom.h>

/* physio / buffer handling */
#include <sys/buf.h>

/* tsleep / sleep / wakeup */
#include <sys/proc.h>
/* hz for above */
#include <sys/kernel.h>

/* bitdefinitions for RX211 */
#include <dev/qbus/rfreg.h>


#define RFS_DENS        0x0001          /* single or double density */
#define RFS_AD          0x0002          /* density auto detect */
#define RFS_NOTINIT     0x0000          /* not initialized */
#define RFS_PROBING     0x0010          /* density detect / verify started */
#define RFS_FBUF        0x0020          /* Fill Buffer */
#define RFS_EBUF        0x0030          /* Empty Buffer */
#define RFS_WSEC        0x0040          /* Write Sector */
```

```
#define RFS_RSEC        0x0050          /* Read Sector */
#define RFS_SMD         0x0060          /* Set Media Density */
#define RFS_RSTAT       0x0070          /* Read Status */
#define RFS_WDDS        0x0080          /* Write Deleted Data Sector */
#define RFS_REC         0x0090          /* Read Error Code */
#define RFS_IDLE        0x00a0          /* controller is idle */
#define RFS_CMDS        0x00f0          /* command mask */
#define RFS_OPEN_A      0x0100          /* partition a open */
#define RFS_OPEN_B      0x0200          /* partition b open */
#define RFS_OPEN_C      0x0400          /* partition c open */
#define RFS_OPEN_MASK   0x0f00          /* mask for open partitions */
#define RFS_OPEN_SHIFT  8               /* to shift 1 to get RFS_OPEN_A */
#define RFS_SETCMD(rf, state)   ((rf) = ((rf) & ~RFS_CMDS) | (state))



/* autoconfig stuff */
static int rfc_match(struct device *, struct cfdata *, void *);
static void rfc_attach(struct device *, struct device *, void *);
static int rf_match(struct device *, struct cfdata *, void *);
static void rf_attach(struct device *, struct device *, void *);
static int rf_print(void *, const char *);

/* device interfce functions / interface to disk(9) */
dev_type_open(rfopen);
dev_type_close(rfclose);
dev_type_read(rfread);
dev_type_write(rfwrite);
dev_type_ioctl(rfioctl);
dev_type_strategy(rfstrategy);
dev_type_dump(rfdump);
dev_type_size(rfsize);


/* Entries in block and character major device number switch table. */
const struct bdevsw rf_bdevsw = {
        rfopen,
        rfclose,
        rfstrategy,
        rfioctl,
```

```
        rfdump,
        rfsize,
        D_DISK
};

const struct cdevsw rf_cdevsw = {
        rfopen,
        rfclose,
        rfread,
        rfwrite,
        rfioctl,
        nostop,
        notty,
        nopoll,
        nommap,
        nokqfilter,
        D_DISK
};



struct rfc_softc {
        struct device sc_dev;           /* common device data */
        struct device *sc_childs[2];    /* child devices */
        struct evcnt sc_intr_count;     /* Interrupt counter for statistics */
        struct buf *sc_curbuf;          /* buf that is currently in work */
        bus_space_tag_t sc_iot;         /* bus_space IO tag */
        bus_space_handle_t sc_ioh;      /* bus_space IO handle */
        bus_dma_tag_t sc_dmat;          /* bus_dma DMA tag */
        bus_dmamap_t sc_dmam;           /* bus_dma DMA map */
        caddr_t sc_bufidx;              /* current position in buffer data */
        int sc_curchild;                /* child whose bufq is in work */
        int sc_bytesleft;               /* bytes left to transfer */
        u_int8_t type;                  /* controller type, 1 or 2 */
};



CFATTACH_DECL(
        rfc,
```

```
        sizeof(struct rfc_softc),
        rfc_match,
        rfc_attach,
        NULL,
        NULL
);



struct rf_softc {
        struct device sc_dev;           /* common device data */
        struct disk sc_disk;            /* common disk device data */
        struct bufq_state sc_bufq;      /* queue of pending transfers */
        int sc_state;                   /* state of drive */
        u_int8_t sc_dnum;               /* drive number, 0 or 1 */
};



CFATTACH_DECL(
        rf,
        sizeof(struct rf_softc),
        rf_match,
        rf_attach,
        NULL,
        NULL
);



struct rfc_attach_args {
        u_int8_t type;          /* controller type, 1 or 2 */
        u_int8_t dnum;          /* drive number, 0 or 1 */
};



struct dkdriver rfdkdriver = {
        rfstrategy
};
```

```
/* helper functions */
int rfc_sendcmd(struct rfc_softc *, int, int, int);
struct rf_softc* get_new_buf( struct rfc_softc *);
static void rfc_intr(void *);



/*
 * Issue a reset command to the controller and look for the bits in
 * RX2CS and RX2ES.
 * RX2CS_RX02 and / or RX2CS_DD can be set,
 * RX2ES has to be set, all other bits must be 0
 */
int
rfc_match(struct device *parent, struct cfdata *match, void *aux)
{
        struct uba_attach_args *ua = aux;
        int i;

        /* Issue reset command. */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS, RX2CS_INIT);
        /* Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set. */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & RX2CS_DONE) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & (RX2ES_RDY | RX2ES_ID)) != 0)
                        break;
                DELAY(100000);  /* wait 100ms */
        }
        /*
         * Give up if the timeout has elapsed
         * and the controller is not ready.
         */
        if (i >= 20)
                return(0);
```

```
        /*
         * Issue a Read Status command with interrupt enabled.
         * The uba(4) driver wants to catch the interrupt to get the
         * interrupt vector and level of the device
         */
        bus_space_write_2(ua->ua_iot, ua->ua_ioh, RX2CS,
            RX2CS_RSTAT | RX2CS_IE);
        /*
         * Wait for command to finish, ignore errors and
         * abort if the controller does not respond within the timeout
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2CS)
                    & (RX2CS_DONE | RX2CS_IE)) != 0
                    && (bus_space_read_2(ua->ua_iot, ua->ua_ioh, RX2ES)
                    & RX2ES_RDY) != 0 )
                        return(1);
                DELAY(100000);  /* wait 100ms */
        }
        return(0);
}



/* #define RX02_PROBE 1 */
#ifdef RX02_PROBE
/*
 * Probe the density of an inserted floppy disk.
 * This is done by reading a sector from disk.
 * Return -1 on error, 0 on SD and 1 on DD.
 */
int rfcprobedens(struct rfc_softc *, int);
int
rfcprobedens(struct rfc_softc *rfc_sc, int dnum)
{
        int dens_flag;
        int i;

        dens_flag = 0;
        do {
```

```
		bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS,
		    RX2CS_RSEC | (dens_flag == 0 ? 0 : RX2CS_DD)
		    | (dnum == 0 ? 0 : RX2CS_US));
		/*
		 * Transfer request set?
		 * Wait 50us, the controller needs this time to setle
		 */
		DELAY(50);
		if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
		    & RX2CS_TR) == 0) {
			printf("%s: did not respond to Read Sector CMD(1)\n",
			    rfc_sc->sc_dev.dv_xname);
			return(-1);
		}
		bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2SA, 1);
		/* Wait 50us, the controller needs this time to setle */
		DELAY(50);
		if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
		    & RX2CS_TR) == 0) {
			printf("%s: did not respond to Read Sector CMD(2)\n",
			    rfc_sc->sc_dev.dv_xname);
			return(-1);
		}
		bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2TA, 1);
		/* Wait for the command to finish */
		for (i = 0 ; i < 200 ; i++) {
			if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
			    RX2CS) & RX2CS_DONE) != 0)
				break;
			DELAY(10000);	/* wait 10ms */
		}
		if (i >= 200) {
			printf("%s: did not respond to Read Sector CMD(3)\n",
			    rfc_sc->sc_dev.dv_xname);
			return(-1);
		}
		if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
		    & RX2CS_ERR) == 0)
			return(dens_flag);
	} while (rfc_sc->type == 2 && dens_flag++ == 0);
```

```
        return(-1);
}
#endif /* RX02_PROBE */



void
rfc_attach(struct device *parent, struct device *self, void *aux)
{
        struct rfc_softc *rfc_sc = (struct rfc_softc *)self;
        struct uba_attach_args *ua = aux;
        struct rfc_attach_args rfc_aa;
        int i;

        rfc_sc->sc_iot = ua->ua_iot;
        rfc_sc->sc_ioh = ua->ua_ioh;
        rfc_sc->sc_dmat = ua->ua_dmat;
        rfc_sc->sc_curbuf = NULL;
        /* Tell the QBus busdriver about our interrupt handler. */
        uba_intr_establish(ua->ua_icookie, ua->ua_cvec, rfc_intr, rfc_sc,
            &rfc_sc->sc_intr_count);
        /* Attach to the interrupt counter, see evcnt(9) */
        evcnt_attach_dynamic(&rfc_sc->sc_intr_count, EVCNT_TYPE_INTR,
            ua->ua_evcnt, rfc_sc->sc_dev.dv_xname, "intr");
        /* get a bus_dma(9) handle */
        i = bus_dmamap_create(rfc_sc->sc_dmat, RX2_BYTE_DD, 1, RX2_BYTE_DD, 0,
            BUS_DMA_ALLOCNOW, &rfc_sc->sc_dmam);
        if (i != 0) {
                printf("rfc_attach: Error creating bus dma map: %d\n", i);
                return;
        }

        /* Issue reset command. */
        bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, RX2CS_INIT);
        /*
         * Wait for the controller to become ready, that is when
         * RX2CS_DONE, RX2ES_RDY and RX2ES_ID are set.
         */
        for (i = 0 ; i < 20 ; i++) {
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
```

```
		    & RX2CS_DONE) != 0
		    && (bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2ES)
		    & (RX2ES_RDY | RX2ES_ID)) != 0)
			break;
		DELAY(100000);	/* wait 100ms */
	}
	/*
	 * Give up if the timeout has elapsed
	 * and the controller is not ready.
	 */
	if (i >= 20) {
		printf(": did not respond to INIT CMD\n");
		return;
	}
	/* Is ths a RX01 or a RX02? */
	if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
	    & RX2CS_RX02) != 0) {
		rfc_sc->type = 2;
		rfc_aa.type = 2;
	} else {
		rfc_sc->type = 1;
		rfc_aa.type = 1;
	}
	printf(": RX0%d\n", rfc_sc->type);

#ifndef RX02_PROBE
	/*
	 * Bouth disk drievs and the controller are one physical unit.
	 * If we found the controller, there will be bouth disk drievs.
	 * So attach them.
	 */
	rfc_aa.dnum = 0;
	rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
	rfc_aa.dnum = 1;
	rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,rf_print);
#else /* RX02_PROBE */
	/*
	 * There are clones of the DEC RX system with standard shugart
	 * interface. In this case we can not be sure that there are
	 * bouth disk drievs. So we want to do a detection of attached
```

```
	 * drives. This is done by reading a sector from disk. This means
	 * that there must be a formated disk in the drive at boot time.
	 * This is bad, but I did not find an other way to detect the
	 * (non)existence of a floppy drive.
	 */
	if (rfcprobedens(rfc_sc, 0) >= 0) {
		rfc_aa.dnum = 0;
		rfc_sc->sc_childs[0] = config_found(&rfc_sc->sc_dev, &rfc_aa,
		    rf_print);
	} else
		rfc_sc->sc_childs[0] = NULL;
	if (rfcprobedens(rfc_sc, 1) >= 0) {
		rfc_aa.dnum = 1;
		rfc_sc->sc_childs[1] = config_found(&rfc_sc->sc_dev, &rfc_aa,
		    rf_print);
	} else
		rfc_sc->sc_childs[1] = NULL;
#endif /* RX02_PROBE */
	return;
}



int
rf_match(struct device *parent, struct cfdata *match, void *aux)
{
	struct rfc_attach_args *rfc_aa = aux;

	/*
	 * Only attach if the locator is wildcarded or
	 * if the specified locator addresses the current device.
	 */
	if (match->cf_loc[RFCCF_DRIVE] == RFCCF_DRIVE_DEFAULT ||
	    match->cf_loc[RFCCF_DRIVE] == rfc_aa->dnum)
		return(1);
	return(0);
}
```

```
void
rf_attach(struct device *parent, struct device *self, void *aux)
{
        struct rf_softc *rf_sc = (struct rf_softc *)self;
        struct rfc_attach_args *rfc_aa = (struct rfc_attach_args *)aux;
        struct rfc_softc *rfc_sc;
        struct disklabel *dl;

        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        rf_sc->sc_dnum = rfc_aa->dnum;
        rf_sc->sc_state = 0;
        rf_sc->sc_disk.dk_name = rf_sc->sc_dev.dv_xname;
        rf_sc->sc_disk.dk_driver = &rfdkdriver;
        disk_attach(&rf_sc->sc_disk);
        dl = rf_sc->sc_disk.dk_label;
        dl->d_type = DTYPE_FLOPPY;              /* drive type */
        dl->d_magic = DISKMAGIC;                /* the magic number */
        dl->d_magic2 = DISKMAGIC;
        dl->d_typename[0] = 'R';
        dl->d_typename[1] = 'X';
        dl->d_typename[2] = '0';
        dl->d_typename[3] = rfc_sc->type == 1 ? '1' : '2';      /* type name */
        dl->d_typename[4] = '\0';
        dl->d_secsize = DEV_BSIZE;              /* bytes per sector */
        /*
         * Fill in some values to have a initialized data structure. Some
         * values will be reset by rfopen() depending on the actual density.
         */
        dl->d_nsectors = RX2_SECTORS;           /* sectors per track */
        dl->d_ntracks = 1;                                                              /*
        dl->d_ncylinders = RX2_TRACKS;          /* cylinders per unit */
        dl->d_secpercyl = RX2_SECTORS;          /* sectors per cylinder */
        dl->d_secperunit = RX2_SECTORS * RX2_TRACKS;    /* sectors per unit */
        dl->d_rpm = 360;                        /* rotational speed */
        dl->d_interleave = 1;                   /* hardware sector interleave */
        /* number of partitions in following */
        dl->d_npartitions = MAXPARTITIONS;
        dl->d_bbsize = 0;               /* size of boot area at sn0, bytes */
        dl->d_sbsize = 0;               /* max size of fs superblock, bytes */
        /* number of sectors in partition */
```

```
        dl->d_partitions[0].p_size = 501;
        dl->d_partitions[0].p_offset = 0;       /* starting sector */
        dl->d_partitions[0].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[0].p_fstype = 0;       /* fs type */
        dl->d_partitions[0].p_frag = 0;         /* fs fragments per block */
        dl->d_partitions[1].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        dl->d_partitions[1].p_offset = 0;       /* starting sector */
        dl->d_partitions[1].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[1].p_fstype = 0;       /* fs type */
        dl->d_partitions[1].p_frag = 0;         /* fs fragments per block */
        dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS;
        dl->d_partitions[2].p_offset = 0;       /* starting sector */
        dl->d_partitions[2].p_fsize = 0;        /* fs basic fragment size */
        dl->d_partitions[2].p_fstype = 0;       /* fs type */
        dl->d_partitions[2].p_frag = 0;         /* fs fragments per block */
        bufq_alloc(&rf_sc->sc_bufq, BUFQ_DISKSORT | BUFQ_SORT_CYLINDER);
        printf("\n");
        return;
}



int
rf_print(void *aux, const char *name)
{
        struct rfc_attach_args *rfc_aa = aux;

        if (name != NULL)
                aprint_normal("RX0%d at %s", rfc_aa->type, name);
        aprint_normal(" drive %d", rfc_aa->dnum);
        return(UNCONF);
}



/* Send a command to the controller */
int
rfc_sendcmd(struct rfc_softc *rfc_sc, int cmd, int data1, int data2)
{
```

```
        /* Write command to CSR. */
        bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS, cmd);
        /* Wait 50us, the controller needs this time to setle. */
        DELAY(50);
        /* Write parameter 1 to DBR */
        if ((cmd & RX2CS_FC) != RX2CS_RSTAT) {
                /* Transfer request set? */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to CMD %x (1)\n",
                            rfc_sc->sc_dev.dv_xname, cmd);
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2DB,
                    data1);
        }
        /* Write parameter 2 to DBR */
        if ((cmd & RX2CS_FC) <= RX2CS_RSEC || (cmd & RX2CS_FC) == RX2CS_WDDS) {
                /* Wait 50us, the controller needs this time to setle. */
                DELAY(50);
                /* Transfer request set? */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2CS)
                    & RX2CS_TR) == 0) {
                        printf("%s: did not respond to CMD %x (2)\n",
                            rfc_sc->sc_dev.dv_xname, cmd);
                        return(-1);
                }
                bus_space_write_2(rfc_sc->sc_iot, rfc_sc->sc_ioh, RX2DB,
                    data2);
        }
        return(1);
}



void
rfstrategy(struct buf *buf)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
```

```
        int i;

        i = DISKUNIT(buf->b_dev);
        if (i >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[i]) == NULL) {
                buf->b_flags |= B_ERROR;
                buf->b_error = ENXIO;
                biodone(buf);
                return;
        }
        rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
        /* We are going to operate on a non open dev? PANIC! */
        if ((rf_sc->sc_state & 1 << (DISKPART(buf->b_dev) + RFS_OPEN_SHIFT))
            == 0)
                panic("rfstrategy: can not operate on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname,
                    DISKPART(buf->b_dev));
        if (buf->b_bcount == 0) {
                biodone(buf);
                return;
        }
        /*
         * BUFQ_PUT() operates on b_rawblkno. rfstrategy() gets
         * only b_blkno that is partition relative. As a floppy does not
         * have partitions b_rawblkno == b_blkno.
         */
        buf->b_rawblkno = buf->b_blkno;
        /*
         * from sys/kern/subr_disk.c:
         * Seek sort for disks.  We depend on the driver which calls us using
         * b_resid as the current cylinder number.
         */
        i = splbio();
        if (rfc_sc->sc_curbuf == NULL) {
                rfc_sc->sc_curchild = rf_sc->sc_dnum;
                rfc_sc->sc_curbuf = buf;
                rfc_sc->sc_bufidx = buf->b_un.b_addr;
                rfc_sc->sc_bytesleft = buf->b_bcount;
                rfc_intr(rfc_sc);
        } else {
                buf->b_resid = buf->b_blkno / RX2_SECTORS;
```

```
                BUFQ_PUT(&rf_sc->sc_bufq, buf);
                buf->b_resid = 0;
        }
        splx(i);
        return;
}



/*
 * Look if there is an other buffer in the bufferqueue of this drive
 * and start to process it if there is one.
 * If the bufferqueue is empty, look at the bufferqueue of the other drive
 * that is attached to this controller.
 * Start procesing the bufferqueue of the other drive if it isn't empty.
 * Return a pointer to the softc structure of the drive that is now
 * ready to process a buffer or NULL if there is no buffer in either queues.
 */
struct rf_softc*
get_new_buf( struct rfc_softc *rfc_sc)
{
        struct rf_softc *rf_sc;
        struct rf_softc *other_drive;

        rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
        rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
        if (rfc_sc->sc_curbuf != NULL) {
                rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
                rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
        } else {
                RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
                other_drive = (struct rf_softc *)
                    rfc_sc->sc_childs[ rfc_sc->sc_curchild == 0 ? 1 : 0];
                if (other_drive != NULL
                    && BUFQ_PEEK(&other_drive->sc_bufq) != NULL) {
                        rfc_sc->sc_curchild = rfc_sc->sc_curchild == 0 ? 1 : 0;
                        rf_sc = other_drive;
                        rfc_sc->sc_curbuf = BUFQ_GET(&rf_sc->sc_bufq);
                        rfc_sc->sc_bufidx = rfc_sc->sc_curbuf->b_un.b_addr;
                        rfc_sc->sc_bytesleft = rfc_sc->sc_curbuf->b_bcount;
```

```
		} else
			return(NULL);
	}
	return(rf_sc);
}



void
rfc_intr(void *intarg)
{
	struct rfc_softc *rfc_sc = intarg;
	struct rf_softc *rf_sc;
	int i;

	rf_sc = (struct rf_softc *)rfc_sc->sc_childs[rfc_sc->sc_curchild];
	do {
		/*
		 * First clean up from previous command...
		 */
		switch (rf_sc->sc_state & RFS_CMDS) {
		case RFS_PROBING:	/* density detect / verify started */
			disk_unbusy(&rf_sc->sc_disk, 0, 1);
			if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
			    RX2CS) & RX2CS_ERR) == 0) {
				RFS_SETCMD(rf_sc->sc_state, RFS_IDLE);
				wakeup(rf_sc);
			} else {
				if (rfc_sc->type == 2
				    && (rf_sc->sc_state & RFS_DENS) == 0
				    && (rf_sc->sc_state & RFS_AD) != 0) {
					/* retry at DD */
					rf_sc->sc_state |= RFS_DENS;
					disk_busy(&rf_sc->sc_disk);
					if (rfc_sendcmd(rfc_sc, RX2CS_RSEC
					    | RX2CS_IE | RX2CS_DD |
					    (rf_sc->sc_dnum == 0 ? 0 :
					    RX2CS_US), 1, 1) < 0) {
						disk_unbusy(&rf_sc->sc_disk,
						    0, 1);
```

```
                                RFS_SETCMD(rf_sc->sc_state,
                                    RFS_NOTINIT);
                                wakeup(rf_sc);
                        }
                } else {
                        printf("%s: density error.\n",
                            rf_sc->sc_dev.dv_xname);
                        RFS_SETCMD(rf_sc->sc_state,RFS_NOTINIT);
                        wakeup(rf_sc);
                }
        }
        return;
case RFS_IDLE:  /* controller is idle */
        if (rfc_sc->sc_curbuf->b_bcount
            % ((rf_sc->sc_state & RFS_DENS) == 0
            ? RX2_BYTE_SD : RX2_BYTE_DD) != 0) {
                /*
                 * can only handle blocks that are a multiple
                 * of the physical block size
                 */
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        }
        RFS_SETCMD(rf_sc->sc_state, (rfc_sc->sc_curbuf->b_flags
            & B_READ) != 0 ? RFS_RSEC : RFS_FBUF);
        break;
case RFS_RSEC:  /* Read Sector */
        disk_unbusy(&rf_sc->sc_disk, 0, 1);
        /* check for errors */
        if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
            RX2CS) & RX2CS_ERR) != 0) {
                /* should do more verbose error reporting */
                printf("rfc_intr: Error reading sector: %x\n",
                    bus_space_read_2(rfc_sc->sc_iot,
                    rfc_sc->sc_ioh, RX2ES) );
                rfc_sc->sc_curbuf->b_flags |= B_ERROR;
        }
        RFS_SETCMD(rf_sc->sc_state, RFS_EBUF);
        break;
case RFS_WSEC:  /* Write Sector */
        i = (rf_sc->sc_state & RFS_DENS) == 0
```

```
                        ? RX2_BYTE_SD : RX2_BYTE_DD;
                disk_unbusy(&rf_sc->sc_disk, i, 0);
                /* check for errors */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
                    RX2CS) & RX2CS_ERR) != 0) {
                        /* should do more verbose error reporting */
                        printf("rfc_intr: Error writing sector: %x\n",
                            bus_space_read_2(rfc_sc->sc_iot,
                            rfc_sc->sc_ioh, RX2ES) );
                        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                        break;
                }
                if (rfc_sc->sc_bytesleft > i) {
                        rfc_sc->sc_bytesleft -= i;
                        rfc_sc->sc_bufidx += i;
                } else {
                        biodone(rfc_sc->sc_curbuf);
                        rf_sc = get_new_buf( rfc_sc);
                        if (rf_sc == NULL)
                                return;
                }
                RFS_SETCMD(rf_sc->sc_state,
                    (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
                    ? RFS_RSEC : RFS_FBUF);
                break;
        case RFS_FBUF:  /* Fill Buffer */
                disk_unbusy(&rf_sc->sc_disk, 0, 0);
                bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
                /* check for errors */
                if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
                    RX2CS) & RX2CS_ERR) != 0) {
                        /* should do more verbose error reporting */
                        printf("rfc_intr: Error while DMA: %x\n",
                            bus_space_read_2(rfc_sc->sc_iot,
                            rfc_sc->sc_ioh, RX2ES));
                        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                }
                RFS_SETCMD(rf_sc->sc_state, RFS_WSEC);
                break;
        case RFS_EBUF:  /* Empty Buffer */
```

```
		i = (rf_sc->sc_state & RFS_DENS) == 0
		    ? RX2_BYTE_SD : RX2_BYTE_DD;
		disk_unbusy(&rf_sc->sc_disk, i, 1);
		bus_dmamap_unload(rfc_sc->sc_dmat, rfc_sc->sc_dmam);
		/* check for errors */
		if ((bus_space_read_2(rfc_sc->sc_iot, rfc_sc->sc_ioh,
		    RX2CS) & RX2CS_ERR) != 0) {
			/* should do more verbose error reporting */
			printf("rfc_intr: Error while DMA: %x\n",
			    bus_space_read_2(rfc_sc->sc_iot,
			    rfc_sc->sc_ioh, RX2ES));
			rfc_sc->sc_curbuf->b_flags |= B_ERROR;
			break;
		}
		if (rfc_sc->sc_bytesleft > i) {
			rfc_sc->sc_bytesleft -= i;
			rfc_sc->sc_bufidx += i;
		} else {
			biodone(rfc_sc->sc_curbuf);
			rf_sc = get_new_buf( rfc_sc);
			if (rf_sc == NULL)
				return;
		}
		RFS_SETCMD(rf_sc->sc_state,
		    (rfc_sc->sc_curbuf->b_flags & B_READ) != 0
		    ? RFS_RSEC : RFS_FBUF);
		break;
	case RFS_NOTINIT: /* Device is not open */
	case RFS_SMD:   /* Set Media Density */
	case RFS_RSTAT: /* Read Status */
	case RFS_WDDS:  /* Write Deleted Data Sector */
	case RFS_REC:   /* Read Error Code */
	default:
		panic("Impossible state in rfc_intr(1).\n");
	}

	if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
		/*
		 * An error occured while processing this buffer.
		 * Finish it and try to get a new buffer to process.
```

```
	 * Return if there are no buffers in the queues.
	 * This loops until the queues are empty or a new
	 * action was successfully scheduled.
	 */
	rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
	rfc_sc->sc_curbuf->b_error = EIO;
	biodone(rfc_sc->sc_curbuf);
	rf_sc = get_new_buf( rfc_sc);
	if (rf_sc == NULL)
		return;
	continue;
}

/*
 * ... then initiate next command.
 */
switch (rf_sc->sc_state & RFS_CMDS) {
case RFS_EBUF:  /* Empty Buffer */
	i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
	    rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
	    ? RX2_BYTE_SD : RX2_BYTE_DD,
	    rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
	if (i != 0) {
		printf("rfc_intr: Error loading dmamap: %d\n",
		i);
		rfc_sc->sc_curbuf->b_flags |= B_ERROR;
		break;
	}
	disk_busy(&rf_sc->sc_disk);
	if (rfc_sendcmd(rfc_sc, RX2CS_EBUF | RX2CS_IE
	    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
	    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
	    | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
	    & 0x30000) >>4), ((rf_sc->sc_state & RFS_DENS) == 0
	    ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
	    rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
		disk_unbusy(&rf_sc->sc_disk, 0, 1);
		rfc_sc->sc_curbuf->b_flags |= B_ERROR;
		bus_dmamap_unload(rfc_sc->sc_dmat,
		rfc_sc->sc_dmam);
```

```
                }
                break;
        case RFS_FBUF:  /* Fill Buffer */
                i = bus_dmamap_load(rfc_sc->sc_dmat, rfc_sc->sc_dmam,
                    rfc_sc->sc_bufidx, (rf_sc->sc_state & RFS_DENS) == 0
                    ? RX2_BYTE_SD : RX2_BYTE_DD,
                    rfc_sc->sc_curbuf->b_proc, BUS_DMA_NOWAIT);
                if (i != 0) {
                        printf("rfc_intr: Error loading dmamap: %d\n",
                            i);
                        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                        break;
                }
                disk_busy(&rf_sc->sc_disk);
                if (rfc_sendcmd(rfc_sc, RX2CS_FBUF | RX2CS_IE
                    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD)
                    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                    | ((rfc_sc->sc_dmam->dm_segs[0].ds_addr
                    & 0x30000)>>4), ((rf_sc->sc_state & RFS_DENS) == 0
                    ? RX2_BYTE_SD : RX2_BYTE_DD) / 2,
                    rfc_sc->sc_dmam->dm_segs[0].ds_addr & 0xffff) < 0) {
                        disk_unbusy(&rf_sc->sc_disk, 0, 0);
                        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                        bus_dmamap_unload(rfc_sc->sc_dmat,
                            rfc_sc->sc_dmam);
                }
                break;
        case RFS_WSEC:  /* Write Sector */
                i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
                    + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
                    ((rf_sc->sc_state & RFS_DENS) == 0
                    ? RX2_BYTE_SD : RX2_BYTE_DD);
                if (i > RX2_TRACKS * RX2_SECTORS) {
                        rfc_sc->sc_curbuf->b_flags |= B_ERROR;
                        break;
                }
                disk_busy(&rf_sc->sc_disk);
                if (rfc_sendcmd(rfc_sc, RX2CS_WSEC | RX2CS_IE
                    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
                    | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
```

```
	    i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
		disk_unbusy(&rf_sc->sc_disk, 0, 0);
		rfc_sc->sc_curbuf->b_flags |= B_ERROR;
	}
	break;
case RFS_RSEC:	/* Read Sector */
	i = (rfc_sc->sc_curbuf->b_bcount - rfc_sc->sc_bytesleft
	    + rfc_sc->sc_curbuf->b_blkno * DEV_BSIZE) /
	    ((rf_sc->sc_state & RFS_DENS) == 0
	    ? RX2_BYTE_SD : RX2_BYTE_DD);
	if (i > RX2_TRACKS * RX2_SECTORS) {
		rfc_sc->sc_curbuf->b_flags |= B_ERROR;
		break;
	}
	disk_busy(&rf_sc->sc_disk);
	if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
	    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
	    | ((rf_sc->sc_state& RFS_DENS) == 0 ? 0 : RX2CS_DD),
	    i % RX2_SECTORS + 1, i / RX2_SECTORS) < 0) {
		disk_unbusy(&rf_sc->sc_disk, 0, 1);
		rfc_sc->sc_curbuf->b_flags |= B_ERROR;
	}
	break;
case RFS_NOTINIT: /* Device is not open */
case RFS_PROBING: /* density detect / verify started */
case RFS_IDLE:	/* controller is idle */
case RFS_SMD:	/* Set Media Density */
case RFS_RSTAT: /* Read Status */
case RFS_WDDS:	/* Write Deleted Data Sector */
case RFS_REC:	/* Read Error Code */
default:
	panic("Impossible state in rfc_intr(2).\n");
}

if ((rfc_sc->sc_curbuf->b_flags & B_ERROR) != 0) {
	/*
	 * An error occured while processing this buffer.
	 * Finish it and try to get a new buffer to process.
	 * Return if there are no buffers in the queues.
	 * This loops until the queues are empty or a new
```

```
                         * action was successfully scheduled.
                         */
                        rfc_sc->sc_curbuf->b_resid = rfc_sc->sc_bytesleft;
                        rfc_sc->sc_curbuf->b_error = EIO;
                        biodone(rfc_sc->sc_curbuf);
                        rf_sc = get_new_buf( rfc_sc);
                        if (rf_sc == NULL)
                                return;
                        continue;
                }
        } while ( 1 == 0 /* CONSTCOND */ );
        return;
}



int
rfdump(dev_t dev, daddr_t blkno, caddr_t va, size_t size)
{

        /* A 0.5MB floppy is much to small to take a system dump... */
        return(ENXIO);
}



int
rfsize(dev_t dev)
{

        return(-1);
}



int
rfopen(dev_t dev, int oflags, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        struct rfc_softc *rfc_sc;
```

```
	struct disklabel *dl;
	int unit;

	unit = DISKUNIT(dev);
	if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
		return(ENXIO);
	}
	rfc_sc = (struct rfc_softc *)rf_sc->sc_dev.dv_parent;
	dl = rf_sc->sc_disk.dk_label;
	switch (DISKPART(dev)) {
		case 0:			/* Part. a is single density. */
			/* opening in single and double density is sensless */
			if ((rf_sc->sc_state & RFS_OPEN_B) != 0 )
				return(ENXIO);
			rf_sc->sc_state &= ~RFS_DENS;
			rf_sc->sc_state &= ~RFS_AD;
			rf_sc->sc_state |= RFS_OPEN_A;
		break;
		case 1:			/* Part. b is double density. */
			/*
			 * Opening a singe density only drive in double
			 * density or simultaneous opening in single and
			 * double density is sensless.
			 */
			if (rfc_sc->type == 1
			    || (rf_sc->sc_state & RFS_OPEN_A) != 0 )
				return(ENXIO);
			rf_sc->sc_state |= RFS_DENS;
			rf_sc->sc_state &= ~RFS_AD;
			rf_sc->sc_state |= RFS_OPEN_B;
		break;
		case 2:			/* Part. c is auto density. */
			rf_sc->sc_state |= RFS_AD;
			rf_sc->sc_state |= RFS_OPEN_C;
		break;
		default:
			return(ENXIO);
		break;
	}
	if ((rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
```

```
		rfc_sc->sc_curchild = rf_sc->sc_dnum;
		/*
		 * Controller is idle and density is not detected.
		 * Start a density probe by issuing a read sector command
		 * and sleep until the density probe finished.
		 * Due to this it is impossible to open unformated media.
		 * As the RX02/02 is not able to format its own media,
		 * media must be purchased preformated. fsck DEC marketing!
		 */
		RFS_SETCMD(rf_sc->sc_state, RFS_PROBING);
		disk_busy(&rf_sc->sc_disk);
		if (rfc_sendcmd(rfc_sc, RX2CS_RSEC | RX2CS_IE
		    | (rf_sc->sc_dnum == 0 ? 0 : RX2CS_US)
		    | ((rf_sc->sc_state & RFS_DENS) == 0 ? 0 : RX2CS_DD),
		    1, 1) < 0) {
			rf_sc->sc_state = 0;
			return(ENXIO);
		}
		/* wait max. 2 sec for density probe to finish */
		if (tsleep(rf_sc, PRIBIO | PCATCH, "density probe", 2 * hz)
		    != 0 || (rf_sc->sc_state & RFS_CMDS) == RFS_NOTINIT) {
			/* timeout elapsed and / or somthing went wrong */
			rf_sc->sc_state = 0;
			return(ENXIO);
		}
	}
	/* disklabel. We use different fake geometries for SD and DD. */
	if ((rf_sc->sc_state & RFS_DENS) == 0) {
		dl->d_nsectors = 10;		/* sectors per track */
		dl->d_secpercyl = 10;		/* sectors per cylinder */
		dl->d_ncylinders = 50;		/* cylinders per unit */
		dl->d_secperunit = 501; /* sectors per unit */
		/* number of sectors in partition */
		dl->d_partitions[2].p_size = 500;
	} else {
		dl->d_nsectors = RX2_SECTORS / 2;  /* sectors per track */
		dl->d_secpercyl = RX2_SECTORS / 2; /* sectors per cylinder */
		dl->d_ncylinders = RX2_TRACKS;     /* cylinders per unit */
		/* sectors per unit */
		dl->d_secperunit = RX2_SECTORS * RX2_TRACKS / 2;
```

```
                /* number of sectors in partition */
                dl->d_partitions[2].p_size = RX2_SECTORS * RX2_TRACKS / 2;
        }
        return(0);
}



int
rfclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfclose: can not close on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        else
                rf_sc->sc_state &= ~(1 << (DISKPART(dev) + RFS_OPEN_SHIFT));
        if ((rf_sc->sc_state & RFS_OPEN_MASK) == 0)
                rf_sc->sc_state = 0;
        return(0);
}



int
rfread(dev_t dev, struct uio *uio, int ioflag)
{

        return(physio(rfstrategy, NULL, dev, B_READ, minphys, uio));
}



int
```

```
rfwrite(dev_t dev, struct uio *uio, int ioflag)
{

        return(physio(rfstrategy, NULL, dev, B_WRITE, minphys, uio));
}



int
rfioctl(dev_t dev, u_long cmd, caddr_t data, int fflag, struct proc *p)
{
        struct rf_softc *rf_sc;
        int unit;

        unit = DISKUNIT(dev);
        if (unit >= rf_cd.cd_ndevs || (rf_sc = rf_cd.cd_devs[unit]) == NULL) {
                return(ENXIO);
        }
        /* We are going to operate on a non open dev? PANIC! */
        if ((rf_sc->sc_state & 1 << (DISKPART(dev) + RFS_OPEN_SHIFT)) == 0)
                panic("rfioctl: can not operate on non-open drive %s "
                    "partition %d", rf_sc->sc_dev.dv_xname, DISKPART(dev));
        switch (cmd) {
        /* get and set disklabel; DIOCGPART used internally */
        case DIOCGDINFO: /* get */
                memcpy(data, rf_sc->sc_disk.dk_label,
                    sizeof(struct disklabel));
                return(0);
        case DIOCSDINFO: /* set */
                return(0);
        case DIOCWDINFO: /* set, update disk */
                return(0);
        case DIOCGPART:  /* get partition */
                ((struct partinfo *)data)->disklab = rf_sc->sc_disk.dk_label;
                ((struct partinfo *)data)->part =
                    &rf_sc->sc_disk.dk_label->d_partitions[DISKPART(dev)];
                return(0);

        /* do format operation, read or write */
        case DIOCRFORMAT:
```

```
        break;
        case DIOCWFORMAT:
        break;

        case DIOCSSTEP: /* set step rate */
        break;
        case DIOCSRETRIES: /* set # of retries */
        break;
        case DIOCKLABEL: /* keep/drop label on close? */
        break;
        case DIOCWLABEL: /* write en/disable label */
        break;

/*      case DIOCSBAD: / * set kernel dkbad */
        break; /* */
        case DIOCEJECT: /* eject removable disk */
        break;
        case ODIOCEJECT: /* eject removable disk */
        break;
        case DIOCLOCK: /* lock/unlock pack */
        break;

        /* get default label, clear label */
        case DIOCGDEFLABEL:
        break;
        case DIOCCLRLABEL:
        break;
        default:
                return(ENOTTY);
        }

        return(ENOTTY);
}
```

B rfreg.h

```
/*
 * Copyright (c) 2002 Jochen Kunz.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The name of Jochen Kunz may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY JOCHEN KUNZ
 * ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL JOCHEN KUNZ
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */



/* Registers in Uni/QBus IO space. */
#define RX2CS   0       /* Command and Status Register */
#define RX2DB   2       /* Data Buffer Register */
/* RX2DB is depending on context: */
#define RX2BA   2       /* Bus Address Register */
#define RX2TA   2       /* Track Address Register */
#define RX2SA   2       /* Sector Address Register */
```

```
#define RX2WC   2       /* Word Count Register */
#define RX2ES   2       /* Error and Status Register */


/* Bitdefinitions of CSR. */
#define RX2CS_ERR       0x8000  /* Error                                 RO */
#define RX2CS_INIT      0x4000  /* Initialize                            WO */
#define RX2CS_UAEBH     0x2000  /* Unibus address extension high bit    WO */
#define RX2CS_UAEBI     0x1000  /* Unibus address extension low bit     WO */
#define RX2CS_RX02      0x0800  /* RX02                                  RO */
/*                      0x0400     Not Used                              -- */
/*                      0x0200     Not Used                              -- */
#define RX2CS_DD        0x0100  /* Double Density                        R/W */
#define RX2CS_TR        0x0080  /* Transfer Request                      RO */
#define RX2CS_IE        0x0040  /* Interrupt Enable                      R/W */
#define RX2CS_DONE      0x0020  /* Done                                  RO */
#define RX2CS_US        0x0010  /* Unit Select                           WO */
#define RX2CS_FCH       0x0008  /* Function Code high bit                WO */
#define RX2CS_FCM       0x0004  /* Function Code mid bit                 WO */
#define RX2CS_FCL       0x0002  /* Function Code low bit                 WO */
#define RX2CS_GO        0x0001  /* Go                                    WO */
#define RX2CS_NU        0x0600  /* not used bits                         -- */


#define RX2CS_UAEB      ( RX2CS_UAEBH | RX2CS_UAEBI )
#define RX2CS_FC        ( RX2CS_FCH | RX2CS_FCM | RX2CS_FCL )


/* Commands of the controller and parameter cont. */
#define RX2CS_FBUF      001     /* Fill Buffer, word count and bus address */
#define RX2CS_EBUF      003     /* Empty Buffer, word count and bus address */
#define RX2CS_WSEC      005     /* Write Sector, sector and track */
#define RX2CS_RSEC      007     /* Read Sector, sector and track */
#define RX2CS_SMD       011     /* Set Media Density, ??? */
#define RX2CS_RSTAT     013     /* Read Status, no params */
#define RX2CS_WDDS      015     /* Write Deleted Data Sector, sector and track */
#define RX2CS_REC       017     /* Read Error Code, bus address */


/* Track Address Register */
```

```
#define RX2TA_MASK      0x7f


/* Sector Address Register */
#define RX2SA_MASK      0x1f


/* Word Count Register */
#define RX2WC_MASK      0x7f


/* Bitdefinitions of RX2ES. */
/*                      <15-12> Not Used                -- */
#define RX2ES_NEM       0x0800  /* Non-Existend Memory  RO */
#define RX2ES_WCO       0x0400  /* Word Count Overflow  RO */
/*                      0x0200     Not Used             RO */
#define RX2ES_US        0x0010  /* Unit Select          RO */
#define RX2ES_RDY       0x0080  /* Ready                RO */
#define RX2ES_DEL       0x0040  /* Deleted Data         RO */
#define RX2ES_DD        0x0020  /* Double Density       RO */
#define RX2ES_DE        0x0010  /* Density Error        RO */
#define RX2ES_ACL       0x0008  /* AC Lost              RO */
#define RX2ES_ID        0x0004  /* Initialize Done      RO */
/*                      0x0002     Not Used             -- */
#define RX2ES_CRCE      0x0001  /* CRC Error            RO */
#define RX2ES_NU        0xF202  /* not used bits        -- */


#define RX2_TRACKS      77      /* number of tracks */
#define RX2_SECTORS     26      /* number of sectors / track */
#define RX2_BYTE_SD     128     /* number of bytes / sector in single density */
#define RX2_BYTE_DD     256     /* number of bytes / sector in double density */
#define RX2_HEADS       1       /* number of heads */
```

C License

D Version History

1.0 First publication.

1.0.1 Correction of various typographical errors.

1.0.1e Initial translation into English by Jan Schaumann

E Bibliography

References

[McK 99] Twenty Years of Berkeley Unix
From AT&T-Owned to Freely Redistributable
http://www.oreilly.com/catalog/opensources/book/kirkmck.html

[Tho] A Machine-Independent DMA Framework for NetBSD
Jason Thorpe
http://www.netbsd.org/Documentation/kernel/bus_dma.ps

[Li 77] A Commentary on the Sixth Edition UNIX Operating System
J. Lions, Department of Computer Science, The University of New South Wales

Index