Writing NetBSD drivers with the bus_space(9) framework

Radoslaw Kujawa – rkujawa@NetBSD.org

The NetBSD Foundation

October 15, 2012

Table of Contents I

1. Introduction
   - Why was this tutorial created?
   - What won't be covered here?
   - What is a driver anyway?
   - What do you need to write a driver?
2. The NetBSD driver model
   - The NetBSD kernel basics
   - Kernel autoconfiguration framework
3. Example driver from scratch
   - Development environment
   - Quick introduction to `GXemul`
   - Our hardware - a fake PCI card
   - Adding a new driver to the NetBSD kernel
   - Matching the PCI device
   - Attaching to the PCI device

Table of Contents II

- Variable types used with `bus_space`
- Mapping the hardware resources
- Accessing the hardware registers

4. Interacting with userspace
   - Device files
   - Using ioctls
   - An example user space program

5. A few tips
   - Avoiding common pitfalls
   - Basic driver debugging

6. Summary
   - The end

Section 1

Introduction

Why was this tutorial created?

- Introductory-level documentation is scarce
- Writing device drivers is often considered black magic
- Reading the `man` pages won't give you the big picture
- BSD systems are always in need of new drivers
- Device drivers are fun ☺

What won't be covered here?

We don't have much time, so several ~~advanced~~ topics were omitted:

- Interrupt handling
- Direct Memory Access and the `bus_dma` framework
- Power management
- Driver detachment
- Drivers as kernel modules
- Examples for buses other than PCI
- Pretty much everything else...

However, once you finish this tutorial, you should be able to pursue this knowledge yourself.

What is a driver anyway?

- The interface between user space and hardware, implemented as a part of the kernel
- The NetBSD drivers are written mostly in C
- Sometimes they have machine dependent assembler parts, but this is a rare case

What do you need to write a driver?

- C programming skills
- Hardware documentation or the ability to reverse engineer the hardware
- A reference driver implementation will help but is not essential
- A NetBSD installation and kernel source, or a cross-build environment (the latter is usually preferred for development of drivers)
- A lot of time, coffee and patience ☺

Why is writing the device drivers considered difficult?

- It's not as difficult as you may expect, in fact during this tutorial we'll prove that it's quite easy
- You need to think on a very low level
  - Good understanding of computer architecture is a must
- Often documentation is the main problem – writing the driver is not possible if you don't understand how the device works
  - No access to documentation (uncooperative hardware vendors, vendors out of business)
  - Documentation is incomplete or plain wrong
  - Reverse engineering can solve these problems but it's a very time consuming process

Section 2

The NetBSD driver model

The NetBSD kernel basics

- NetBSD has a classic monolithic UNIX-like kernel - all drivers are running in the same address space
- Thanks to the above, communication between drivers and other kernel layers is simple
- However, it also means that one badly written driver can affect the whole kernel
- Numerous in-kernel frameworks standardise the way drivers are written (`bus_space`, `autoconf`, etc.)

The NetBSD source directory structure

- We'll only cover parts interesting for a device driver programmer
- `src/sys/` - kernel source directory
- `src/sys/dev/` - machine-independent device drivers
- `src/sys/arch/` - port-specific or architecture-specific parts (such as the low-level system initialisation procedures or machine-dependent drivers)
- `src/sys/arch/$PORTNAME/conf/` - kernel configuration files for a given port

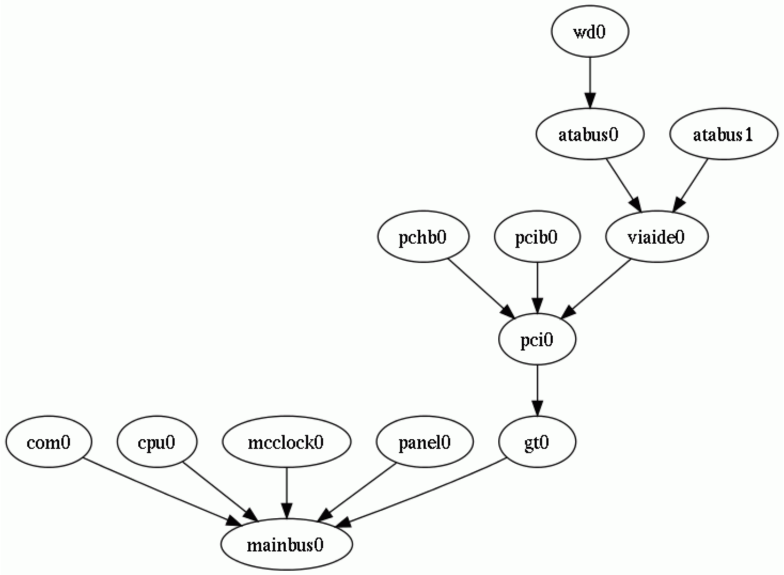
Kernel autoconfiguration framework - autoconf(9)

- Autoconfiguration is the process of matching hardware devices with an appropriate device driver
- The kernel message buffer (dmesg) contains information about autoconfiguration of devices
- `driver0 at bus0:  Foo hardware`
  - Instance 0 of the driver has attached to instance 0 of the particular bus
  - Such messages often carry additional bus-specific information about the exact location of the device (like the device and function number on the PCI bus)
- `driver0:  some message`
  - Additional information about the driver state or device configuration

Autoconfiguration as seen in the dmesg

```
NetBSD 6.99.12 (GENERIC) #7: Fri Oct  5 18:43:21 CEST 2012
        rkujawa@saiko.local:/Users/rkujawa/netbsd-eurobsdcon2012/src/sys/arch/cobalt/compile/obj/GENERIC
Cobalt Qube 2
total memory = 32768 KB
avail memory = 27380 KB
mainbus0 (root)
com0 at mainbus0 addr 0x1c800000 level 3: ns16550a, working fifo
com0: console
cpu0 at mainbus0: QED RM5200 CPU (0x28a0) Rev. 10.0 with built-in FPU Rev. 1.0
cpu0: 48 TLB entries, 256MB max page size
cpu0: 32KB/32B 2-way set-associative L1 instruction cache
cpu0: 32KB/32B 2-way set-associative write-back L1 data cache
mcclock0 at mainbus0 addr 0x10000070: mc146818 compatible time-of-day clock
panel0 at mainbus0 addr 0x1f000000
gt0 at mainbus0 addr 0x14000000
pci0 at gt0
pchb0 at pci0 dev 0 function 0: Galileo GT-64011 System Controller, rev 1
pcib0 at pci0 dev 9 function 0
pcib0: VIA Technologies VT82C586 PCI-ISA Bridge, rev 57
viaide0 at pci0 dev 9 function 1
viaide0: VIA Technologies VT82C586 (Apollo VP) ATA33 controller
viaide0: primary channel interrupting at irq 14
atabus0 at viaide0 channel 0
viaide0: secondary channel interrupting at irq 15
atabus1 at viaide0 channel 1
wd0 at atabus0 drive 0
wd0: <netbsd-cobalt.img>
wd0: 750 MB, 1524 cyl, 16 head, 63 sec, 512 bytes/sect x 1536192 sectors
```

Autoconfiguration as seen in the dmesg



The bus_space(9) framework

- "The goal of the bus_space functions is to allow a single driver source file to manipulate a set of devices on different system architectures, and to allow a single driver object file to manipulate a set of devices on multiple bus types on a single architecture."
- Provides a set of functions implementing common operations on the bus like mapping, reading, writing, copying, etc.
- The bus_space(9) is implemented at the machine-dependent level (typically it's a part of architecture-specific code), but all implementations present the same interface[1]

[1] At least they should, some functions are missing on less popular ports

Machine independent drivers

- If possible drivers should work on any hardware platform
- High quality, machine-independent (MI) drivers are an important factor that adds to NetBSD portability
- Some drivers are completely MI, some have MD or bus dependent attachments and some are completely MD
  - A driver for a typical PCI card will be completely MI
  - A driver for the components of a SoC will usually be completely MD
- The `bus_space` abstraction helps to achieve portability, transparently handling endianness issues and hiding bus implementation details from the device driver
- Even if we have MI drivers, writing the drivers is always significant part of effort needed to port NetBSD to new hardware

Section 3

Example driver from scratch

Development environment

- Out of scope of this course, but very well documented
- Cross compiling is an easy task with the `build.sh` script
- Described in Part V of the NetBSD Guide
- Check out the NetBSD sources
- `$ build.sh -m cobalt tools` will build compiler, assembler, linker, etc. for cobalt port
- `$ build.sh -m cobalt kernel=GENERIC` will build the GENERIC kernel for cobalt
- Call `build.sh` with a `-u` parameter to update (won't rebuilding everything)
- `build.sh` is calling `nbconfig` and `nbmake` tools, no magic involved

Quick introduction to GXemul

- A framework for full-system computer architecture emulation, excellent for educational purposes
- Capable of emulating several real machines supported by NetBSD
- We'll emulate a Cobalt, MIPS-based micro server with PCI bus
- I've modified `GXemul` and implemented an emulation of an additional PCI device
- It will be used to show (almost) a real-life example of the driver development process

Our hardware - functional description

- Business applications often use arithmetic operations like addition
- Fake Cards Inc. responded to market needs and created a new product, Advanced Addition Accelerator
- Pointy Haired Bosses will certainly buy it to accelerate their business applications, so let's create a driver for NetBSD!

Our hardware - technical details

- Overview
  - Implemented as a PCI device
  - Arithmetic unit capable of addition of two numbers
  - Four[2] registers in the PCI memory space
- PCI configuration space
  - Identified by the PCI vendor ID `0xfabc` and product ID `0x0001`
  - Base Address Register 0x10 used to configure the engine address
  - 4 x 32-bit registers = 16 bytes
  - Other configuration registers irrelevant

[2] Only three of these registers are of any importance for us at this moment

Our hardware - technical details (memory mapped register set)

- Advanced Addition Acceleration registers

Register Name	Offset	Description
COMMAND	0x4	Register used to issue commands to the engine
DATA	0x8	Register used to load data to internal engine registers
RESULT	0xC	Register used to store the result of arithmetic operation

- COMMAND register

Bit	R/W	Description
0	W	Execute ADD operation on values loaded into internal register A and B
1	R/W	Select internal register A for access through DATA register
2	R/W	Select internal register B for access through DATA register

- Selecting internal register A and B at the same time will lead to undefined behaviour

Our hardware - technical details (memory mapped register set)

- DATA register

Bit	R/W	Description
0:31	R/W	Read/write the value in internal engine register

- RESULT register

Bit	R/W	Description
0:31	R	Holds the result of last ADD operation

Our hardware - technical details (operation algorithm)

- Select the internal register `A` for access (write `0x2` into `COMMAND` register)
- Write the first number into `DATA` register
- Select the internal register `B` for access (write `0x4` into `COMMAND` register)
- Write the second number into `DATA` register
- Issue the `ADD` operation (write `0x1` into `COMMAND` register)
- Read the result from `RESULT` register

Adding a new driver to the NetBSD kernel

- We'll discuss the steps needed to add a new MI PCI device driver to the NetBSD kernel
  - Add the vendor and device ID to the database of PCI IDs
  - Create a set of the driver source files in `src/sys/dev/$BUSNAME/`
  - Add the new driver to a `DEVNAMES` file
  - Add the new driver to a `src/sys/dev/$BUSNAME/$BUSNAME.files` file

Modifying the PCI device database

```
unmatched vendor 0xfabc product 0x0001 (Co-processor
processor, revision 0x01) at pci0 dev 12 function 0
not configured
```

- The kernel does not know anything about this vendor and device
- Add it to the PCI device database - `src/sys/dev/pci/pcidevs`
- `vendor VENDORNAME 0xVENDORID Long Vendor Name`
- `product VENDORNAME PRODUCTNAME 0xPRODUCTID Long Product Name`
- To regenerate pcidevs*.h run `awk -f devlist2h.awk pcidevs` or `Makefile.pcidevs` if you're on NetBSD

Modifying the PCI device database - example

```
--- pcidevs 29 Sep 2012 10:26:14 -0000 1.1139
+++ pcidevs 5 Oct 2012 08:52:59 -0000
@@ -669,6 +669,7 @@
 vendor CHRYSALIS 0xcafe Chrysalis-ITS
 vendor MIDDLE_DIGITAL 0xdeaf Middle Digital
 vendor ARC 0xedd8 ARC Logic
+vendor FAKECARDS 0xfabc Fake Cards
 vendor INVALID 0xffff INVALID VENDOR ID

 /*
@@ -2120,6 +2121,9 @@
 /* Eumitcom products */
 product EUMITCOM WL11000P 0x1100 WL11000P PCI WaveLAN/IEEE 802.11

+/* FakeCards products */
+product FAKECARDS AAA 0x0001 Advanced Addition Accelerator
+
 /* O2 Micro */
 product O2MICRO 00F7 0x00f7 Integrated OHCI IEEE 1394 Host Controller
 product O2MICRO OZ6729 0x6729 OZ6729 PCI-PCMCIA Bridge
```

Modifying the PCI device database - example

```
Fake Cards Advanced Addition Accelerator (Co-processor
processor, revision 0x01) at pci0 dev 12 function 0
not configured
```

- Now the kernel knows the vendor and product ID
- But there's still no driver for this device

Adding the new PCI driver

- Choose a name - short, easy to remember, avoid numbers
  - `faa` looks like a good name, but you can choose any name you like
- Create a set of new files in `src/sys/dev/pci`
  - `faa.c` - main driver code
  - `faareg.h` - register definitions[3]
  - `faavar.h` - driver structures and functions used in other parts of the kernel[4]
- Modify driver definitions
  - `src/sys/dev/pci/files.pci`
  - `src/sys/dev/DEVNAMES`
- Add the driver to a port-specific kernel configuration file - `src/sys/arch/$PORTNAME/conf/GENERIC`

[3] Might not exist if the driver is only a simple passthrough from a specific bus to another MI driver.

[4] Omitted if not needed.

Adding the new PCI driver - main driver

- Kernel includes are at the beginning, followed by machine-specific and bus-specific includes
- Should also include `faareg.h` and `faavar.h` files
- A minimal driver needs just two functions
  - `faa_match` (or `faa_probe` for some buses)
  - `faa_attach`
- The `CFATTACH_DECL_NEW` macro plugs the above functions into `autoconf(9)` mechanism

Adding the new PCI driver - main driver

- `static int faa_match(device_t parent, cfdata_t match, void *aux);`
  - Check if the driver should attach to a given device (for example in case of PCI bus, it will be used to check vendor and product ID)
  - `parent` - pointer to parent's driver device structure
  - `match` - pointer to `autoconf(9)` details structure
  - `aux` - despite the name the most important argument, usually contains bus-specific structure describing device details
- `static void faa_attach(device_t parent, device_t self, void *aux);`
  - Attach the driver to a given device
  - `parent` - same as with match function
  - `self` - pointer to driver's device structure
  - `aux` - same as with match function
- See definitions of these functions in the driver(9) man page.

Adding the new PCI driver - main driver cont'd

- `CFATTACH_DECL_NEW(faa, sizeof(struct faa_softc), faa_match, faa_attach, NULL, NULL);`
    - driver name
    - size of softc structure containing state of driver's instance
    - match/probe function
    - attach function
    - detach function
    - activate function
- The "_NEW" name is not fortunate
- Pass `NULL` for unimplemented functions
- We won't cover `detach` and `activate` now, as they are not needed for a simple driver

Adding the new PCI driver - main driver example

- `src/sys/dev/pci/faa.c`

```
#include <sys/cdefs.h>
__KERNEL_RCSID(0, "$NetBSD$");
#include <sys/param.h>
#include <sys/device.h>
#include <dev/pci/pcivar.h>
#include <dev/pci/pcidevs.h>
#include <dev/pci/faareg.h>
#include <dev/pci/faavar.h>

static int      faa_match(device_t, cfdata_t, void *);
static void     faa_attach(device_t, device_t, void *);

CFATTACH_DECL_NEW(faa, sizeof(struct faa_softc),
    faa_match, faa_attach, NULL, NULL);

static int
faa_match(device_t parent, cfdata_t match, void *aux)
{
        return 0;
}

static void
faa_attach(device_t parent, device_t self, void *aux)
{
}
```

Adding the new PCI driver - auxiliary includes

- `src/sys/dev/pci/faareg.h`

```
#ifndef FAAREG_H
#define FAAREG_H
/*
 * Registers are defined using preprocessor:
 * #define FAA_REGNAME 0x0
 * We'll add them later, let's leave it empty for now.
 */
#endif /* FAAREG_H */
```

- `src/sys/dev/pci/faavar.h`

```
#ifndef FAAVAR_H
#define FAAVAR_H

/* sc_dev is an absolute minimum, we'll add more later */
struct faa_softc {
        device_t sc_dev;
};
#endif /* FAAVAR_H */
```

Adding the new PCI driver - registering the driver

- src/sys/dev/DEVNAMES

```
--- DEVNAMES 1 Sep 2012 11:19:58 -0000 1.279
+++ DEVNAMES 6 Oct 2012 19:59:06 -0000
@@ -436,6 +436,7 @@
 ex MI
 exphy MI
 ezload MI Attribute
+faa MI
 fb luna68k
 fb news68k
 fb newsmips
```

Adding the new PCI driver - registering the driver

- `src/sys/dev/pci/files.pci`

```
--- pci/files.pci 2 Aug 2012 00:17:44 -0000 1.360
+++ pci/files.pci 6 Oct 2012 19:59:10 -0000
@@ -1122,3 +1122,9 @@
 device tdvfb: wsemuldisplaydev, rasops8, vcons, videomode
 attach tdvfb at pci
 file dev/pci/tdvfb.c tdvfb
+
+# FakeCards Advanced Addition Accelerator
+device faa
+attach faa at pci
+file dev/pci/faa.c faa
+
```

Adding the new PCI driver to the kernel configuration

- `src/sys/arch/cobalt/conf/GENERIC`

```
--- GENERIC 10 Mar 2012 21:51:50 -0000 1.134
+++ GENERIC 6 Oct 2012 20:12:37 -0000
@@ -302,6 +302,9 @@
 #fms* at pci? dev ? function ? # Forte Media FM801
 #sv* at pci? dev ? function ? # S3 SonicVibes

+# Fake Cards Advanced Addition Accelerator
+faa* at pci? dev ? function ?
+
 # Audio support
 #audio* at audiobus?
```

- The above definition means that an instance of `faa` may be attached to any PCI bus, any device, any function
- The exact position of the rule in the configuration file is not important in this case
- See config(5) for a description of the device definition language

Adding the new PCI driver - example

- The driver should compile now
- The driver's match function will check if the driver is able to work with a given device
- Since it is not implemented, the kernel will not attach the driver

Matching the PCI device

- Modify the `faa_match` function to match the specified PCI device
- Use `PCI_VENDOR` and `PCI_PRODUCT` macros to obtain the IDs

```
static int
faa_match(device_t parent, cfdata_t match, void *aux)
{
        const struct pci_attach_args *pa = (const struct pci_attach_args *)aux;

        if ((PCI_VENDOR(pa->pa_id) == PCI_VENDOR_FAKECARDS)
            && (PCI_PRODUCT(pa->pa_id) == PCI_PRODUCT_FAKECARDS_AAA))
                return 1;

        return 0;
}
```

Attaching to the PCI device

```
faa0 at pci0 dev 12 function 0
```

- The driver has successfully matched and attached to the PCI device but still is not doing anything useful
- Let's fill an attach function and actually program the hardware

Variable types used with bus_space

- bus_space_tag_t – type used to describe a particular bus, usually passed to the driver from MI bus structures
- bus_space_handle_t – used to describe a mapped range of bus space, usually created with the bus_space_map() function
- bus_addr_t – address on the bus
- bus_size_t – an amount of space on the bus
- Contents of these types are MD, so avoid modifying from within the driver[5]

[5] although you'll often have to use bus_size_t

Why do we need to “map” the resources?

- In a memory-protected environment like NetBSD one cannot directly access physical addresses
- The kernel has its own virtual address space
- Physical space can be made visible in kernel virtual address space through the process of mapping
- It’s a machine-dependent process but it’s also conveniently hidden from the programmer by the bus_space framework
- “The bus space must be mapped before it can be used, and should be unmapped when it is no longer needed”

Mapping the hardware resources

- The generic bus_space(9) way to map space

```
bus_space_map(bus_space_tag_t space, bus_addr_t address,
bus_size_t size, int flags, bus_space_handle_t  *handlep);
```

- `bus_space_map` creates a mapping from the physical address to a kernel virtual address
- `space` – represents the bus on which the mapping will be created
- `address` – typically represents the physical address for which a mapping will be created
- `size` – describes the amount of bus space to be mapped
- `handlep` – pointer to mapped space (filled after successful mapping)
- Separate space and address

Mapping the hardware resources

- The PCI-specific way to map space

```
pci_mapreg_map(const struct pci_attach_args *pa, int reg, pcireg_t type,
int busflags, bus_space_tag_t *tagp, bus_space_handle_t *handlep,
bus_addr_t *basep, bus_size_t *sizep);
```

- `pci_mapreg_map` creates mapping from physical address present in specified BAR register to kernel virtual address
- `pa` – struct describing PCI attachment details (passed through aux)
- `reg` – BAR register number
- `type` – Select mapping type (I/O, memory)
- `busflags` – Passed to `bus_space_map` flags argument
- `tagp` – pointer to `bus_space_tag`
- `handlep` – pointer to a mapped space
- `basep` – address of a mapped space
- `sizep` – size of mapped space (equivalent to BAR size)
- The last four parameters are filled after successful mapping

Mapping the registers using BAR - adding auxiliary includes

- src/sys/dev/pci/faareg.h

```
#define FAA_MMREG_BAR   0x10
```

- src/sys/dev/pci/faavar.h

```
struct faa_softc {
        device_t sc_dev;

        bus_space_tag_t sc_regt;
        bus_space_handle_t sc_regh;
        bus_addr_t sc_reg_pa;

};
```

Mapping the registers using BAR - main driver code

- src/sys/dev/pci/faa.c

```
static void
faa_attach(device_t parent, device_t self, void *aux)
{
        struct faa_softc *sc = device_private(self);
        const struct pci_attach_args *pa = aux;

        sc->sc_dev = self;

        pci_aprint_devinfo(pa, NULL);

        if (pci_mapreg_map(pa, FAA_MMREG_BAR, PCI_MAPREG_TYPE_MEM, 0,
            &sc->sc_regt, &sc->sc_regh, &sc->sc_reg_pa, 0) != 0 ) {
            aprint_error_dev(sc->sc_dev, "can't map the BAR\n");
            return;
        }

        aprint_normal_dev(sc->sc_dev, "regs at 0x%08x\n", (uint32_t) sc->
             sc_reg_pa);
}
```

Accessing the hardware registers

- The `bus_space_read_*` and `bus_space_write_*` functions are basic methods of reading and writing the hardware registers
- `uintX_t bus_space_read_X(bus_space_tag_t space, bus_space_handle_t handle, bus_size_t offset);`
- `void bus_space_write_X(bus_space_tag_t space, bus_space_handle_t handle, bus_size_t offset, uintX_t value);`
  - `space` - tag describing the bus
  - `handle` - describes the exact location on the bus where read/write should occur, this handle is obtained by `bus_space_map`
  - `offset` - offset from handle location
  - The read function returns the data read from the specified location, while write has an argument `value` which should be filled with data to be written

Variants of bus_space_read and bus_space_write

Data	Read function	Write function
8-bit	bus_space_read_1	bus_space_write_1
16-bit	bus_space_read_2	bus_space_write_2
32-bit	bus_space_read_4	bus_space_write_4
64-bit	bus_space_read_8	bus_space_write_8

- There are many more variants of read and write functions and they are useful in certain situations, see the bus_space(9) man page

Accessing the hardware registers - example

- Create a function that will write a value into the DATA register of our device, then read it back and check if the value is the same as written
- Define the DATA register in the driver
- `src/sys/dev/pci/faareg.h`

```
#define FAA_DATA                0x8
#define FAA_COMMAND             0x4
#define FAA_COMMAND_STORE_A         __BIT(1)
```

- Define the new function in main driver code
- `static bool faa_check(struct faa_softc *sc);`

Accessing the hardware registers - example

- src/sys/dev/pci/faa.c

```
static void
faa_attach(device_t parent, device_t self, void *aux)
{
   /* ... */
   if (!faa_check(sc)) {
    aprint_error_dev(sc->sc_dev, "hardware not responding\n");
        return;
   }
}

static bool
faa_check(struct faa_softc *sc)
{
        uint32_t testval = 0xff11ee22;
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_COMMAND,
             FAA_COMMAND_STORE_A);
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_DATA, testval);
        if (bus_space_read_4(sc->sc_regt, sc->sc_regh, FAA_DATA) == testval)
                return true;

        return false;
}
```

Accessing the hardware registers - running the example

- Update the kernel binary and run it again
- Check the GXemul log

```
[ faa: COMMAND register (0x4) WRITE value 0x2 ]
[ faa: DATA register (0x8) WRITE value 0xff11ee22 ]
[ faa: DATA register (0x8) READ value 0xff11ee22 ]
```

- GXemul will conveniently display all accesses to our device
- The `faa` driver still does attach without error, which means that the check function is working properly

```
faa0 at pci0 dev 12 function 0: Fake Cards Advanced Addition Accelerator (rev. 0x01)
faa0: registers at 0x10110000
```

Implementing addition using the hardware

- The basic principle of device operation should be laid out in the data sheet
- We need to implement an algorithm based on this description
  Jump to device description
- Writing such an algorithm is often not needed, since the NetBSD kernel already has frameworks for common device types (such as `atabus/wd` for IDE and SATA hard disk controllers, `wsdisplay/wscons` for frame buffers, etc.)

Implementing addition using the hardware

- Define all registers
- `src/sys/dev/pci/faareg.h`

```
#define FAA_STATUS              0x0
#define FAA_COMMAND             0x4
#define FAA_COMMAND_ADD             __BIT(0)
#define FAA_COMMAND_STORE_A         __BIT(1)
#define FAA_COMMAND_STORE_B         __BIT(2)
#define FAA_DATA                0x8
#define FAA_RESULT              0xC
```

Implementing addition using the hardware

- Add a new function to the main driver code
- `src/sys/dev/pci/faa.c`

```
static void
faa_attach(device_t parent, device_t self, void *aux)
{
        /* ... */
        aprint_normal_dev(sc->sc_dev, "just checking: 1 + 2 = %d\n", faa_add(sc,
            1, 2));
}

static uint32_t
faa_add(struct faa_softc *sc, uint32_t a, uint32_t b)
{
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_COMMAND,
            FAA_COMMAND_STORE_A);
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_DATA, a);
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_COMMAND,
            FAA_COMMAND_STORE_B);
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_DATA, b);
        bus_space_write_4(sc->sc_regt, sc->sc_regh, FAA_COMMAND, FAA_COMMAND_ADD
            );
        return bus_space_read_4(sc->sc_regt, sc->sc_regh, FAA_RESULT);
}
```

Implementing addition using the hardware - running the example

- Update the kernel binary and run it again
- Check GXemul log

```
[ faa: COMMAND register (0x4) WRITE value 0x2 ]
[ faa: DATA register (0x8) WRITE value 0x1 ]
[ faa: COMMAND register (0x4) WRITE value 0x4 ]
[ faa: DATA register (0x8) WRITE value 0x2 ]
[ faa: COMMAND register (0x4) WRITE value 0x1 ]
[ faa: RESULT register (0xC) READ value 0x3 ]
```

- Looks like it worked!

```
faa0 at pci0 dev 12 function 0: Fake Cards Advanced Addition Accelerator (rev. 0x01)
faa0: registers at 0x10110000
faa0: just checking: 1 + 2 = 3
```

Section 4

Interacting with userspace

The kernel-user space interface

- Now that the core functionality of the kernel driver is working, it should be exposed to user space
- The interface between kernel driver and userspace can be designed in many different ways
- The classic UNIX way of interfacing between the kernel and user space is a device file
- Even when using device files there is no single interfacing method that fits all use cases
- It's up to the programmer to define the communication protocol

Device files

- `crw-r----- 1 root wheel 101, 1 Aug 12 21:53 /dev/file`
- The kernel identifies which driver should service the request to this file by using major and minor numbers (101 and 1 in the example above)
- The major number identifies the driver
- The minor number usually identifies the driver instance, although the driver is free to use it in any other way
- In NetBSD device files are created statically
  - By the `MAKEDEV` script during installation or boot
  - Manually by using the `mknod` utility

Operations on device files

- open(2) and close(2)
- read(2) and write(2)
- ioctl(2)
- poll(2)
- mmap(2)
- and more...
- Any mix of the above system calls might be used to interface between the kernel and user space
- We'll later implement an `ioctl(2)`-based communication mechanism

Adding `cdevsw`

- `cdevsw` is used to decide which operation on the character device file calls which driver function
- Not all calls have to be implemented, although some device layers define a set of calls that a driver must implement
- For example disk drivers must implement open, close, read, write and ioctl

- `src/sys/dev/pci/faa.c`

```
dev_type_open(faaopen);
dev_type_close(faaclose);
dev_type_ioctl(faaioctl);

const struct cdevsw faa_cdevsw = {
        faaopen, faaclose, noread, nowrite, faaioctl,
        nostop, notty, nopoll, nommap, nokqfilter, D_OTHER
};
```

Prototyping the cdevsw operations

- The dev_type* macros are used to prototype the functions passed to cdevsw
- Pass no followed by a function name to the appropriate cdevsw field if it is not implemented
- There's also bdevsw for block devices, but we won't use it in this example
- The last member of the cdevsw structure defines the device flags, originally it was used to define the device type (still used for disks, tape drives and ttys, for other devices pass D_OTHER)

Implemeting the cdevsw operations - open / close

- src/sys/dev/pci/faa.c

```
int
faaopen(dev_t dev, int flags, int mode, struct lwp *l)
{
        struct faa_softc *sc;
        sc = device_lookup_private(&faa_cd, minor(dev));

        if (sc == NULL)
                return ENXIO;
        if (sc->sc_flags & FAA_OPEN)
                return EBUSY;

        sc->sc_flags |= FAA_OPEN;
        return 0;
}
int
faaclose(dev_t dev, int flag, int mode, struct lwp *l)
{
        struct faa_softc *sc;
        sc = device_lookup_private(&faa_cd, minor(dev));

        if (sc->sc_flags & FAA_OPEN)
                sc->sc_flags =~ FAA_OPEN;

        return 0;
}
```

Defining the `ioctls`

- `ioctl(2)` can be used to call kernel-level functions and exchange data between the kernel and user space
- The classic way of passing data is by using structures, their definitions are shared between the kernel and user space code
- The driver might support more than one `ioctl`, the `_IO*` macros are used to define the operation and associated structure used to exchange data
  - `_IO` - just a kernel function call, no data exchange
  - `_IOR` - kernel function call and data pass from kernel to user space
  - `_IOW` - kernel function call and data pass from user space to kernel
  - `_IOWR` - kernel function call and data exchange in both directions
  - `#define DRIVERIO_IOCTLNAME _IOXXX(group, ioctl_number, data structure)`

Defining the ioctls

- `src/sys/dev/pci/faaio.h`

```
#include <sys/ioccom.h>

#define FAAIO_ADD _IOWR(0, 1, struct faaio_add)

struct faaio_add {
    uint32_t a;
    uint32_t b;
    uint32_t *result;
};
```

- In the above example the `ioctl` group is not defined (`0`), but a single letter identifier could appear as first argument to `_IOWR`

Implemeting the cdevsw operations - ioctl

- src/sys/dev/pci/faa.c

```
int
faaioctl(dev_t dev, u_long cmd, void *data, int flag, struct lwp *l)
{
        struct faa_softc *sc = device_lookup_private(&faa_cd, minor(dev));
        int err;

        switch (cmd) {
        case FAAIO_ADD:
                err = faaioctl_add(sc, (struct faaio_add *) data);
                break;
        default:
                err = EINVAL;
                break;
        }
        return(err);
}
static int
faaioctl_add(struct faa_softc *sc, struct faaio_add *data)
{
        uint32_t result; int err;

        aprint_normal_dev(sc->sc_dev, "got ioctl with a %d, b %d\n",
            data->a, data->b);

        result = faa_add(sc, data->a, data->b);
        err = copyout(&result, data->result, sizeof(uint32_t));
        return err;
}
```

Using copyout to pass data to userspace

- The `copy(9)` functions are used to copy kernel space data from/to user space
- `copyout(kernel_address, user space_address, size);`
- Actually on Cobalt we could just do `*data->result = faa_add();` instead of calling the copyout function, but that is a bad idea
- Some architectures (such as `sparc64`) have totally separate kernel and user address spaces $\Longrightarrow$ user space addresses are meaningless in the kernel

Defining device major number

- Device major numbers for hardware drivers are usually defined in a per-port manner[6]
- `src/sys/arch/$PORTNAME/conf/majors.$PORTNAME`
- `src/sys/arch/cobalt/conf/majors.cobalt`
- The following defines a new character device file called `/dev/faa*` with major number 101, but only if the `faa` driver is included in the kernel (last argument)
- `device-major faa char 101 faa`

[6] It's also possible to define a major in a machine-independent way in `src/sys/conf/majors`

Creating the device node

- The `mknod` utility can be used to create the device file manually
- The driver name can be specified instead of the major number - it will be automatically resolved into the correct major number
- `mknod name [b | c] [major | driver] minor`
- `mknod /dev/faa0 c faa 0`
- Created successfully
- `crw-r--r-- 1 root wheel 101, 0 Oct 8 2012 /dev/faa0`

An example user space program

- The example program will open the device file and call `ioctl(2)` on it
- As simple as possible, just to show how communication is done
- Using `ioctls` from the user space
  - Open the device file with `O_RDWR`
  - Call `ioctl(2)` with the operation number and structure as parameters

An example user space program - source

```
void add(int, uint32_t, uint32_t);

static const char* faa_device = "/dev/faa0";

int
main(int argc, char *argv[])
{
        int devfd;

        if (argc != 3) {
                printf("usage: %s a b\n", argv[0]);
                return 1;
        }
        if ( (devfd = open(faa_device, O_RDWR)) == -1) {
                perror("can't open device file");
                return 1;
        }

        add(devfd, atoi(argv[1]), atoi(argv[2]));

        close(devfd);
        return 0;
}
```

An example user space program - source

```
void
add(int devfd, uint32_t a, uint32_t b)
{
        struct faaio_add faaio;
        uint32_t result = 0;

        faaio.result = &result;
        faaio.a = a;
        faaio.b = b;

        if (ioctl(devfd, FAAIO_ADD, &faaio) == -1) {
                perror("ioctl failed");
        }
        printf("%d\n", result);
}
```

An example user space program - running it

```
# make
cc -o aaa_add aaa_add.c
# ./aaa_add 3 7
faa0: got ioctl with a 3, b 7
10
```

- The program is successfully accessing the `faa` driver through the `ioctl`
- The `faa0:...` line is a kernel message, normally only seen on the console terminal

Section 5

A few tips

Avoiding common pitfalls

- Always free resources allocated in the `match` or `probe` functions
- Always use `bus_space` methods, don't access the hardware using a pointer dereference
- If possible test on more than one hardware architecture, some bugs may surface
- Don't reinvent the wheel, try to use existing kernel frameworks as much as possible
- Use `copy(9)` (or `uiomove(9)` or `store(9)`/`fetch(9)`) to move data between the kernel and user space

Basic driver debugging

- Use `aprint_debug` to print debug-level messages on console and log them (enabled by passing `AB_DEBUG` from the boot loader)
- Use the built-in DDB debugger
  - Enabled by the kernel option `DDB`
  - A kernel panic will start DDB if the `DDB_ONPANIC=1` kernel option is specified or the `ddb.onpanic` sysctl is set to 1.
  - Run `# sysctl -w kern.panic_now=1` to trigger a panic manually (`DIAGNOSTIC` option)
- Remote debugging is possible on some ports
  - With KGDB through the serial port
  - With IPKDB through the network

Section 6

Summary

Further reading

- Documentation, articles:
  - A Machine-Independent DMA Framework for NetBSD, Jason R. Thorpe
  - Writing Drivers for NetBSD, Jochen Kunz
  - NetBSD Documentation: Writing a pseudo device
  - autoconf(9), bus_space(9) bus_dma(9) driver(9), pci(9) `man` pages, etc.
- Example source code of drivers:
  - `tdvfb`, `voodoofb` are fairly good frame buffer driver examples with documentation publicly available.
  - `etsec` is a nice example of a more complicated network interface driver

Get the source code

- Download the source code and materials for this tutorial
- `https://github.com/rkujawa/busspace-eurobsdcon2012`
- `https://github.com/rkujawa/gxemul-eurobsdcon2012`

Questions?

- Do you have any questions?

The End. . .

Thank you!