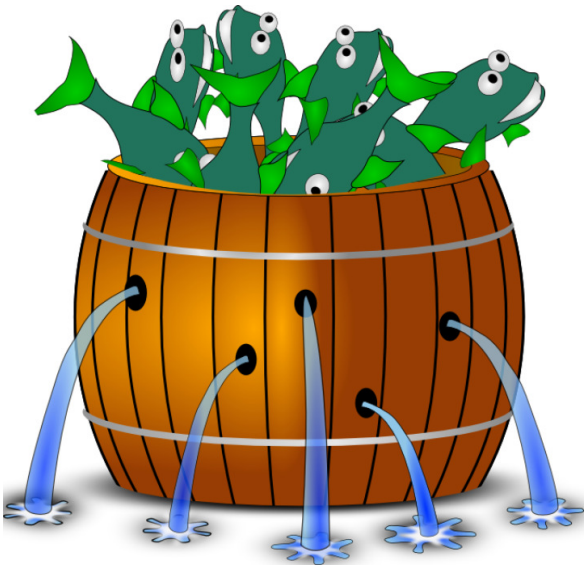
Barrelfish Project
ETH Zurich



Device Drivers in Barrelfish

Barrelfish Technical Note 19

Barrelfish project

16.05.2017

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
0.1	05.12.2013	GZ	Initial Version
0.1	16.05.2017	GZ	Update with info about new driver structure

Contents

Chapter 1

New device driver interface

This document describes how we write device drivers in Barrelfish. It will walk you through the necessary steps to integrate your driver in the Barrelfish infrastructure and gives an overview of available APIs, libraries and tools to help you with the process.

1.1 Overview

There are three main entities when discussing drivers:

Driver Domain Is a domain that executes one or more drivers. It is special in that it communicates with Kaluga to act on requests to spawn or destroy new driver instances.

Driver Module A Barrelfish driver module is a piece of code (typically a library) that contains the logic for a device driver. It follows a well defined structure that allows Kaluga to interact with an instantiated driver (see Driver Instance) in order to control its life-cycle.

Driver Instance A driver instance is the runtime object instantiated from a given driver module. In practice, any number of instances can be created from a driver module and executed within one or more driver domains.

1.2 Driver Domain

A driver domain is a regular domain (process) in Barrelfish. The template driver domain can be found in `usr/drivers/domain/main.c`, with most of the respective logic implemented in `lib/driverkit`. Driver domains are typically started by Kaluga (device manager), and then continue to communicate with Kaluga using the ddomain interface (`if/ddomain.if`).

The ddomain interface exposes two API calls:

`create` Creates a driver instance from a driver module.

`destroy` Destroys a previously created driver instance.

A driver domain can "dynamically" instantiate driver instances if a create call is made to spawn an instance of a specific module. The driver domain uses a basic C-style module system (similar to Linux kernel or Tensorflow modules). The module system uses a custom ELF

section (called .bfdrivers) where structs with module informations are stored. This is achieved using a special linker file located in `lib/driverkit/bfdrivers.ld`.

1.3 Driver Module

The driver module is written as a library. A template module can be found in `usr/drivers/domain/drivertpl` The driver is then statically linked with the driver domain. Currently, there is no support to dynamically load modules since everything is statically linked in Barrelfish anyways. It's important that the module is not linked like a regular library with the domain, but rather as a module (*addModules = ["drivermodule"]*). This makes sure the linker will not throw away the (seemingly unused) symbols during linking. For an example, take a look at `usr/drivers/domain/Hakefile`.

A driver must provide an implementation for the following five functions:

init Initialize driver an device.

detach Release ownership of the device temporarily.

attach Regain ownership of the device (after *detach*).

set_sleep_level Change power state of the device.

destroy Destruct the driver instance, release ownership of device.

A driver implementation can register itself as a module using the *DEFINE_MODULE* macro:

```
DEFINE_MODULE(uart, init, attach, detach, set_sleep_level, destroy);
```

Listing 1.1: Registering an uart driver module

1.4 Driver Instance

At runtime, the driver domain creates instances of a given module. This means a having a specific, per-instance state coupled with the functions provided in module along with the dcontrol interface (`if/dcontrol.if`) that is exported for every driver instance.

Kaluga connects to the dcontrol interface of a driver instance to have more fine grained control over the life-cycle of the instance. It exports the following messages which basically act as wrappers for the driver functions described in Section 1.3:

attach Calls attach on the driver instance.

detach Calls detach on the driver instance.

set_sleep_level Calls *set_sleep_level* on the driver instance.

Chapter 2

Legacy device drivers

Legacy drivers live within a 9single) domain and the structure, as well as the control interface of the driver is completely up to the programmer. No one was really happy with this so we introduced some more rules of how a device driver is written as described in the previous chapter. Unfortunately, most of our current device drivers are still legacy drivers. So in order to understand what is going on in such a driver, this chapter is kept around for now.

In a first step, we will look at the necessary bits and pieces to write a new driver in Barrelfish on the ARM platform, by using an existing, very simple driver as a walk-through from the Barrelfish code base – the FDIF driver, a device used for face detection typically found on OMAP chips.

The relevant code for the driver resides in the tree at `usr/drivers/omap44xx/fdif/`. In order to get an idea of what the driver entails, we will look at its Hakefile. Hakefiles are the declarative description of a how a program is built in Barrelfish and what its dependencies are. If you need more informations on our build system, you should have a look at the Hake technote [1], which provides a detailed description of Hake, the Barrelfish build system. We will look at the Hakefile for the FDIF driver in Listing 2.1 (a Haskell program, really) and explain what it means for the driver program. `cFiles` is a list of C source files that are compiled and linked for this driver. In our case, we use hake to just search for every file that has a "c" file extension in the current directory and return this as a list of files for `cFiles`. If you go and look at `usr/drivers/omap44xx/fdif` you will find that this will encompass only two files, `fdif.c` and `picture.c`.

The next attribute, `mackerelDevices`, contains a list Mackerl files, these are devices that we have specified in our domain specific language called Mackerel, and we would now like to use in our driver. If you do not know what Mackerel is, let me explain it to you: Mackerel is a DSL for describing device registers. You can find an abundance of Mackerel files in the `devices` directory inside the source tree. For example, the `omap44xx_fdif` device mentioned in the Hakefile at the third position is found in `devices/omap/omap44xx_fdif.dev`. They all, describe a particular device we use in Barrelfish to varying degrees of detail, ranging from ethernet cards to xAPIC or even descriptions of the file- allocation-table for the FAT file-system. Now, you might think, well, why can't I just use a regular C struct or even integer types with bit operations for that? And in general you could. However, the Mackerel compiler gives you a lot of nice things on top of this description. For one thing, it will generate for you all the code you need to program the register with certain values, that means it takes care of all the bit-shifting and masking operations based on your description. On the other hand, it will

generate functions that let you print the contents of the register in a human readable way, which is a very useful thing for debugging. For more information on Mackerel you should read the Mackerel Technote [3].

The `addLibraries` entry specifies the libraries your application needs in order to run. In this example, we add driverkit, a helper library for finding, and mapping our device registers in the virtual memory space of the application.

The last argument specifies the architectures we want to build this driver for. Since we are currently using this device on ARM/OMAP4 platforms, this is set to ARMv7 and ARMv7-M architectures.

```
[   build application {
        target = "fdif",
        cFiles = (find withSuffices [".c"]),
        mackerelDevices = [
            "omap/omap44xx_cam_prm",
            "omap/omap44xx_cam_cm2",
            "omap/omap44xx_fdif",
            "omap/omap44xx_device_prm" ],
        addLibraries = ["driverkit"],
        architectures = ["armv7", "armv7-m"]
}]
```

Listing 2.1: A Hakefile for a simple device driver

Now let's have a look at `usr/drivers/omap44xx/fdif/fdif.c`, the device driver source code. If you open the file, aside from the comment header, you will see a bunch of included header files that are of interest to us (Listing 2.2).

```
#include <dev/omap/omap44xx_cam_prm_dev.h>
#include <dev/omap/omap44xx_cam_cm2_dev.h>
#include <dev/omap/omap44xx_fdif_dev.h>
```

Listing 2.2: Mackerel includes in driver source code.

These look very familiar to our specified `mackerelDevices` in the Hakefile, and in fact, they are the generated header files based on our mackerel files. If you want to have a look at them, you can find these header files in your build directory in `<arch>/include/dev/omap`. In our code, we use various functions (the ones starting with `omap44xx_`) that are defined in these header files and use them to access device registers.

Another interesting part is early on in the main function (Listing 2.3).

```
err = map_device_register(0x4A10A000, 4096, &vbase);
```

Listing 2.3: Mapping device registers in virtual memory.

`map_device_register` is a function provided by the driverkit library. We will talk more about it later, but for now, here is what it does: It takes the physical address of a device register (`0x4A10A000`) the size of the register (`4096`) and will map this at a random virtual address in your address space (given back to you by `vbase`). Since our device drivers all run in user-space this function ensures that you can access the device in your address space. There is also the issue of how, and which programs we allow access to what device registers. We will discuss this in the next chapter.

For the the FDIF device, we can receive an interrupt from the device in case the face processing is done. Since we run in user space, we have to invoke a system call to register for the interrupt in the kernel. Once the interrupt arrives, the kernel will use the message passing infrastructure of Barrelfish to forward the IRQ to us. Fortunately, libbarrelfish provides us with a high- level interface to do just that. In the function *enable_irq_mode*, we register an interrupt for the device by using *inthandler_setup_arm* (Listing 2.4). It takes as arguments a handler function (*irq_handler*) that is executed in case the interrupt arrives, an additional state argument that is passed to that function for this particular interrupt, in our case NULL, and the interrupt number or vector we are interested in.

```
err = inthandler_setup_arm(irq_handler, NULL, FDIF_IRQ);
```

Listing 2.4: Register to receive an Interrupt.

This concludes our walkthrough on the FDIF driver. So far, you have seen a glimpse of the user-level side on writing device drivers for ARM. It consists of an interplay of the build system, mackerel device descriptions, mapping device registers, interrupt registrations and your actual driver code. In the next chapters we will have a closer look on what is actually happening behind the scenes and how to adapt the infrastructure for new architcures or boards.

Note that the FDIF driver is a very minimal example of a driver. We use it to teach students about the basic concepts of device drivers. However, if you would want to write a real driver, you also need to export a interface for clients. In Barrelfish, the typical way is to export a message passing interface for the driver, so that applications can connect and communicate with the driver using messages. There are many source code examples in the tree on how to do this, as a starting point, have a look at `usr/examples/xmpl-call-response` in the source tree and the tech- note on inter-dispatcher communication [2] to get started.

2.1 ARM and the simple SKB

If you look in the source tree of the SKB (`usr/skb/`) you will find that there are currently two different versions of the SKB built. One is the SKB based on the ECLiPSe runtime engine for x86 systems, the other is the SKB simple for ARM. This is due to portability issues of the ECLiPSe runtime for ARM. So, what is the simple SKB? It is an implementation of the Octopus API. It is important to have at least a minimal a implementation of Octopus for all architectures we run on. Because it provides essential system features such as the name-service which is used most for of the service look-ups.

Not having the constraint logic programming interface unfortunately means we can currently not use the APIs in `lib/skb` on ARM. We are currently investigating alternatives for a constraint solver that will run on both platforms.

2.2 Kaluga – The device manager

Kaluga is the device manager in Barrelfish. Its responsibility is to manage the periperhals of a system. That encompasses starting the correct drivers, once a device is discovered, in the right order and making sure that each driver has the permissions (capabilities) to access the device' memory areas or I/O ports. In this chapter wewill learn how Kaluga interacts with the rest of the system for device discovery and driver start-up.

For reasons stated in Section 2.1 we currently do not have the full system knowledge base on non-x86 platforms. Also, the ARM platforms we support right now, do not have an infrastructure like PCI that brings automatic device discovery – there are device trees, but we do not have support for them yet.

This means that right now, the way Kaluga finds the available devices differs quite a bit based on the platform we are running on. This ranges from automatic discovery using PCI, Octopus and the SKB on x86, to hardcoded information in Kaluga for the OMAP4 SoC. However, the general way of discovering what drivers are available and how we start them remains the same. We will briefly look at what operations Kaluga provides for finding binaries and how you can program it to start drivers the right way in your system.

If you look inside of the main function in Kaluga, you can see a call to the *init_boot_modules* function. We usually rely on multiboot to provide us with a set of ELF files at start-up. The *init_boot_modules* function parses the information provided by multiboot (your menu.lst file) to find a list of available binaries. It then looks at the arguments that are hardcoded next to those binaries and follows a simple policy for these, if a binary has the argument ''auto'' next to it, it considers this binary as a driver and will start it, if it finds a suitable device. How Kaluga finds a suitable device is explained in the following sections. Drivers are started in different ways, ranging from just starting one driver binary to a number of binaries or sending notifications to other subsystems and starting a driver. Kaluga supports custom start-up policies for different binaries in your system, you can set a start-up policy per driver binary using the *set_start_function*. The default start function, the one that is chosen if no special start-up function, is set for a binary, is defined in `usr/kaluga/driver_startup.c`. For example, on x86, this function will just spawn the binary and provide as arguments the PCI device identifiers (bus, class, function etc.) to the driver program.

As we mentioned before, a driver usually needs a special set of permissions to gain access to the device registers. For historical reasons, the way we provide this permissions currently differs between x86 and ARM. Unifying this interface is part of future work.

2.3 Starting PCI drivers on x86

On x86, peripherals are usually in the form of PCI or PCI express cards. PCI supports automatic discovery of periperhals using PCI bus enumeration. In Barrelfish, the PCI related code lives in `usr/pci`. PCI is structured as a hierarchical tree with it's leaves being devices. The root node, called PCI root bridge, forms the entry point to such a tree. PCI root bridges are found by reading the ACPI tables. ACPI, short for Advanced Configuration and Power Interface, is an open standard for device configuration and power management in operating systems. ACPI related code in Barrelfish lives in `usr/acpi`.

The bootstrapping of an x86 machine in Barrelfish works as follows: After parsing the boot script, Kaluga starts ACPI. ACPI will then add specific Octopus records for every PCI root bridge it finds. Meanwhile, Kaluga will receive notification for all the root bridges added to Octopus. If a root bridge is found, Kaluga will start the PCI domain which in turn will do a PCI bus enumeration. Devices found during PCI bus enumeration are again added to Octopus and propagated to Kaluga which will start individual device drivers to handle the peripherals. How does Kaluga know what driver to start for each device record? We already discussed how Kaluga uses different start functions for different types of devices. But how do we choose

the right binary? Kaluga uses the SKB that stores a mapping from PCI identifiers to driver binaries. This mapping is retrieved from the SKB once Kaluga receives a Octopus record for a new device. You will find the mapping database in `usr/skb/programs/device_db.pl`. If you want to start your PCI driver with Kaluga, you will need to add it there and provide at least the corresponding device and vendor id.

Barrelfish has a number of drivers for PCI cards. Mostly for network interfaces. Barrelfish drivers, including the ones for PCI, are located in the source tree in `usr/drivers/`.

2.4 Writing PCI drivers

In order to write a PCI driver, one has to communicate with the PCI domain. There is a client library that provides a helpful API in `lib/pci` that helps doing that. One of the first steps is to initialize the client library by connecting to the PCI domain:

```
err = pci_client_connect();
```

Listing 2.5: A client connects to the PCI subsystem.

After that, you are able to invoke the library functions in Listing 2.6 to initialize devices. The functions allow to gain control for a specific PCI device. The device is identified by using the numerous PCI identifiers (subclass, prog_if, vendor et. al.). The caller provides a callback function (*init_func*) that gets called by the library once it has registered the device with the PCI domain. *init_func* takes as an argument an array of `struct device_mem`. A description of the basic address registers (BAR) for this PCI device and also permissions (capabilities) to map these address registers in the drivers address space. You can use the defined in helper functions in `include/pci/mem.h` to map these BARs into the address space of the client. For legacy devices (such as a serial driver for example) that live in the I/O address space and do not use memory mapped registers you can use the *pci_register_legacy_driver_irq* function.

```
errval_t pci_register_driver_noirq(pci_driver_init_fn init_func, uint32_t class,
                                   uint32_t subclass, uint32_t prog_if,
                                   uint32_t vendor, uint32_t device,
                                   uint32_t bus, uint32_t dev, uint32_t fun);

errval_t pci_register_driver_irq(pci_driver_init_fn init_func, uint32_t class,
                                 uint32_t subclass, uint32_t prog_if,
                                 uint32_t vendor, uint32_t device,
                                 uint32_t bus, uint32_t dev, uint32_t fun,
                                 interrupt_handler_fn handler, void *handler_arg);

errval_t pci_register_legacy_driver_irq(legacy_driver_init_fn init_func,
                                        uint16_t iomin, uint16_t iomax, int irq,
                                        interrupt_handler_fn handler,
                                        void *handler_arg);
```

Listing 2.6: A driver uses one of the following functions to register for PCI devices.

Note that the discussed PCI API is rather low-level and provides a lot of freedom in who can register for PCI devices. In the future the plan is for x86 to push more of that complexity in Kaluga. The device registration with PCI should be done by Kaluga before the driver is started, the driver then only receives a list of capabilities for a particular device which it can map in its address space. That means a driver no longer has the need to call these functions.

2.5 Writing drivers for ARM and System-on-Chip platforms

You have already encountered most of the provided functionality for ARM drivers in the overview in Chapter 1.1. This section will focus on how we currently support the OMAP platform to start drivers in Kaluga.

On ARM the situation differs compared to x86. There is currently no established standard like PCI for x86. That means that the way we have to integrate ARM differs from platform to platform. We also have no support for device trees at the moment. Therefore, if you look at the *main* function in Kaluga, you will find that we currently look-up the binaries using the *find_module* function and hardwire the start-up of these drivers for the pandaboard platform. In `omap_startup.c` we define the start function for these binaries. If you look at the code in the file you'll also see that we use the function *spawn_program_with_caps* to start the driver and pass the driver a list of memory capabilities to access the device memory. This is the service part of the driverkit library we have seen in Chapter 1.1, it makes sure the capabilities are actually given to the driver in a way that driverkit can map them. What capabilities we give to a driver for the OMAP chip is also hardcoded at the moment, you can find a series of `struct allowed_registers` in the same file that defines for a given driver, what memory ranges it is allowed to access. The situation is not solved sufficiently right now, in the future, we would like to store this information in a SKB like system that also runs on ARM and lets us query for information about various platforms.

If you go on and read the capability technote [4] you'll learn that capabilities can only be created in the kernel, and the representation we have in user-space, are references to capabilities. So, a valid question here is how Kaluga gets the capabilities for these devices in the first place. For that we have to look at the *device_caps.c* file inside Kaluga. The file contains the capability manager or memory manager for Kaluga, it is an instance of the memory management library found in `lib/mm`. The memory manager (`libmm`) manages capabilities for you, in reality it is a B+-tree structure that will manage a certain range of memory, in our case device memory. It allows you to request a smaller range from this usually very large range that we initalize our memory manager with and, `libmm` will split up the inital capability we gave to the instance at the beginning into smaller pieces and hand them out to you, giving you a way to have fine grained, page level access control on memory. In practice, because capabilities can only be created and split in half in the kernel, it has to invoke system calls to do that.

As a note aside, there are three important memory managers in the system. The one found in memserv (`usr/memserv`), it manages all physical memory, the one in ACPI (`usr/acpi`), it handles all device memory on x86 and should really be merged with the third one in Kaluga that we use for ARM.

If you look at the function *init_cap_manager* in `usr/kaluga/device_caps.c` you will find a call to the monitor to request the I/O capability:

```
err = cl->vtbl.get_io_cap(cl, &requested_cap, &error_code);
```

Listing 2.7: RPC call to receive the I/O capability from the monitor.

In the case of the ARM Pandaboard, the requested capability allows one to access the whole space of the device memory. We pass this capability on to the device manager in the *mm_add* call further down. Now, we are free to use the *get_device_cap* function, also defined in this file to create fine grained capabilities for this entire memory range. If you go back and look at

code in usr/kaluga/omap_startup.c you will find it actually uses *get_device_cap* to create the capabilities it needs to pass on to the device drivers.

Now you should understand how the user-space side works if you want to write user-space drivers for your own platform. We have not covered yet how we actually create a capability in the kernel and how it ends up in the monitor, but we will cover that shortly in Section 2.8.

2.6 Kernel support for user-space drivers

In this chapter, we will look at the necessary support in the kernel, if we want to write user-level device drivers on a new, unsupported platform. We cover the main parts that are needed in this case: How do we forward interrupts to user-space and how we create capabilities for device memory.

2.7 Interrupts

In Chapter 1.1 we have already seen how we can register to receive interrupts using the message passing architecture in Barrelfish. In this section we will look at what the kernel does in order to forward the interrupt to you. It all starts with having a driver for your interrupt controller. We have support for a number of interrupt controllers already in Barrelfish, like the xAPIC on x86 (kernel/x86/apic.c) or the GIC in ARMv7 (kernel/arch/armv7/gic.c). If there is currently no interrupt controller for your architecture, you'll have to write one yourself. In any case, if you want to forward interrupts to user-space, you can rely on the *send_user_interrupt* function provided by the architecture independent part of the CPU driver. It allows you to forward interrupts from the kernel to a domain running on its core using the message passing infrastructure of Barrelfish [2].

2.8 Device Memory

In Section 2.5 we talked about how Kaluga constructs a series of smaller capabilities for device drivers from an initial, huge capability it receives from the monitor. We also mention that capabilities are created in the kernel. In this Section we look at what is necessary to create capabilities for device memory and how we can pass it on to user-space.

First you need to know what memory areas your devices are in. On x86 we usually ask the BIOS to get a list of memory regions for RAM and device memory. In Barrelfish, we construct capabilities for these regions and we hand the device regions to ACPI which is the domain that initializes the ACPI subsystem and does the memory book keeping for PCI drivers. On an ARM platform, the device memory usually lives in a statically pre-defined range. In kernel/arch/omap44xx/startup_arch.c in *spawn_init_common* we can see how we construct a capability for the device memory range of the the OMAP4 platform. The relevant parts are given in Listing 2.8.

```
struct cte *iocap = caps_locate_slot(CNODE(spawn_state.taskcn), TASKCN_SLOT_IO);
errval_t  err = caps_create_new(ObjType_DevFrame, 0x40000000, 30, 30, iocap);
```

Listing 2.8: Creating a cabaility in the kernel and placing it in the I/O slot in a task cnode.

spawn_init_common is setting up a new dispatcher control block, for the first user-space program called init, in the system. Similar to a UNIX based OS, all subsequent programs are children of init. The call to *caps_create_new* creates a new capability of type `ObjType_DevFrame`, a special type for device memory that makes sure the pages are not zeroed before mapping it for the first time. The next two arguments are the physical base of the address range and the size (in bits) of the range. This particular capability covers a memory range of 2^{30} bytes, or one GiB, starting from address `0x40000000` – the device memory region of the OMAP4 chip. The last argument specifies where this new capability is stored. The location is defined by the preceding *caps_locate_slot* function call. You can think of the *caps_locate_slot* function as an array look-up. We use the task CNode (a table of capabilities) of `spawn_state`, a struct representing the kernel state for the init domain. We use `TASKCN_SLOT_IO` as an index to the cnode table. Once init is started, it can refers to this capability by using the `TASKCN_SLOT_IO` offset to find it. If you look inside `usr/init/spawn.c` you will find the code (Listing 2.9) doing just that to propagate the capability on to the monitors task cnode. Notice that *cap_copy* is now a system call. The monitor then can use the I/O capability in his task cnode if somebody requests it (for example by using *get_io_cap*, seen in Listing 2.7).

```
/* Give monitor IO */
dest.cnode = si->taskcn;
dest.slot  = TASKCN_SLOT_IO;
src.cnode = cnode_task;
src.slot  = TASKCN_SLOT_IO;
err = cap_copy(dest, src);
if (err_is_fail(err)) {
    return err_push(err, INIT_ERR_COPY_IO_CAP);
}
```

Listing 2.9: Copy of the I/O capability from `src` to `dest`.

2.9 Limitations & Work in Progress

Altough we currently have the necessary support for user-space drivers on both major platforms Barrelfish runs on we do not yet have an unified interface between ARM and x86 architectures. In this technote we have seen both approaches explained to varying levels of details and we mentioned briefly where the two approaches differ. In the future we will most likely unify both platforms under a standardized API which will have the best of both worlds.

References

[1] Hake. Technical Report 003, Systems Group, ETH Zurich, Apr. 2010.

[2] Inter-dispatcher communication in Barrelfish. Technical Report 011, Systems Group, ETH Zurich, Oct. 2010.

[3] Mackerel 1.2 User Guide. Technical Report 002, Systems Group, ETH Zurich, Apr. 2010.

[4] Capability Management in Barrelfish. Technical Report 013, Systems Group, ETH Zurich, Mar. 2011.