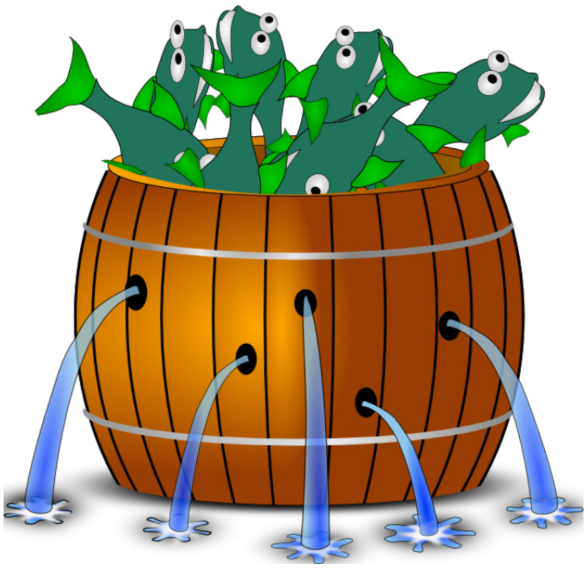
Barrelfish Project
ETH Zurich



Barrelfish Glossary

Barrelfish Technical Note 001

Barrelfish Project

01.12.2013

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
1.0	31.05.2010	TR	Initial version
2.0	01.12.2013	TR	Updated with new developments

Glossary

Ancestor: A capability A is an ancestor of a capability X if X is a descendant of A.

APP Core: Application processor, processors booted either by the BSP or other APP cores and not the initial boot-loader or firmware.

Aquarium: A visualization tool for Barrelfish trace data. Aquarium is written in C# and can accept data as a stream over the network from a running Barrelfish machine, or from a trace file. It displays a time line showing which dispatchers are running on which cores, messages between dispatchers, and other system events.

Beehive: Beehive was an experimental soft-core processor designed by Chuck Thacker at Microsoft Research Silicon Valley. Beehive was implemented in a simulator and on FPGAs, in particular the BEE3 processor emulation board (which could run up to 15 Beehive cores at a time). The architecture had a number of unusual features, in particular, a ring interconnect for message passing, a software-visible FIFO on each core for incoming messages, and a memory system implemented using message passing (loads and stores became RPCs to the memory system). Barrelfish was ported to the Beehive processor but support for the architecture was eventually dropped after the Beehive project completed.

Boot driver: A piece of code running on a "home core" to manage a "target core". It encapsulates the hardware functionality to boot, suspend, resume, and power-down the latter.

BSP Core: Refers to the bootstrap processor, meaning the first processor that is usually booted by the boot-loader or firmware on a hardware architecture.

cap: See *capability*.

capability *(cap)*: Barrelfish uses "partitioned capabilities" to refer to most OS resources (most of which are actually typed areas of memory). Operations on such resources are typically carried out "invoking" an operation on the capability via a system call to the local CPU driver. Capabilities are fixed-size data structures which are held in areas of memory called CNodes, and which cannot be directly read from or written to by user mode programs. Instead, users move capabilities between CNodes by invoking operations on the capability that refers to the CNode. Capabilities can be "retyped" and progressively refined from pure memory capabilities to ones which can be mapped into an address space, for example, or used as CNodes. The set of capability types understood by Barrelfish is defined using the Hamlet language.

capability node *(cnode)*: A region of RAM containing capabilities. A CNode cannot be mapped into a virtual address space, and so can only be accessed by operations on the CNode capability which refers to it – this is how strict partitioning of capability memory from normal memory is achieved. The set of CNodes which can be referenced by a single dispatcher is called the cspace.

capability space *(cspace)*: The set of capabilities held by a dispatcher on a core is organized in a guarded page table structure implemented as a tree of CNodes, where internal nodes are CNodes and leaf nodes are non-CNode capabilities. This allows any capability held by a dispatcher to be referred to using a 32-bit address; when invoking a capability, the CPU driver walks the cspace structure resolving a variable number of bits of this address at each level. This means that capabilities used frequently should be stored near the top of the tree, preferably in the root CNode. Some CPU drivers allow a fast path where the 32-bit address is implicitly assumed to simply an offset in the root CNode. The capability for the root CNode can be efficiently found from the DCB; note that, unlike the vspace, the cspace is purely local to a core and cannot be shared between dispatchers on difference cores.

CC-UMP: See *User-level Message Passing*.

Channel: A uni-directional kernel-mediated communication path between dispatchers. All messages travel over channels. Holding a capability for a channel guarantees the right to send a message to it (although the message may not be sent for reasons other than protection).

cnode: See *capability node*.

CNode capability: A capability type referring to a CNode.

CPU driver: The code running on a core which executes in kernel or privileged mode. CPU drivers are the Barrelfish equivalent of a kernel, except that there is one per core, and they share no state or synchronization. CPU drivers are typically non-preemptible, single-threaded, and mostly stateless.

cspace: See *capability space*.

DCB: See *dispatcher control block*.

Descendant: A capability X is a descendant of a capability A if: X was retyped from A, or X is a descendant of A1 and A1 is a copy of A, or X is a descendant of B and B is a descendant of A, or X is a copy of X1 and X1 is a descendant of A.

device frame capability: A capability type which refers to a region of physical address space containing memory-mapped I/O registers. A direct subtype of a Physical Address capability.

dispatcher: The data structure representing an application's runnability on a core. Dispatchers contain scheduling state, message end points, and upcall entry points; they are the nearest equivalent to processes in Unix. Dispatchers are always tied to a core. A Barrelfish application can be viewed as a collection of dispatchers spread across the set of cores on which the application might run, together with associated other resources. The term is from K42. Multiple dispatchers may share a vspaceor cspace.

dispatcher capability: A capability referring to the memory region occupied by a Dispatcher Control Block.

dispatcher control block *(DCB)*: The kernel object representing the dispatcher. DCBs are referred to by specially typed capabilities.

dite: A tool used to build a boot image in the proprietary Intel "32.obj" format, for booting on the Single-chip Cloud Computer.

Domain: The word domain is used to refer to the user-level code sharing a protection domain and (usually) an address space. A domain consists of one or more dispatchers.

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

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

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

Elver: An intermediate boot loader for booting 64-bit ELF images on Intel-architecture machines where the main boot loader (such as Grub) does not support entering a kernel in long mode. Elver is specified as the first module of the multiboot image, and puts the boot processor into long mode before jumping to the CPU driver, which is assumed to be in the second multiboot module.

endpoint capability: A capability referring to a (core-local) communication endpoint. Posession of such a capability enables the holder to send a message to endpoint.

errno.h: See *Fugu*.

Filet-o-Fish *(FoF)*: An embedding of C into Haskell, used for writing C code generators for Haskell-based domain specific languages. Instead of using C syntax combinators (as used in Flounder and Mackerel) FoF-based DSLs (such as Fugu and Hamlet) use backend which are actually semantic specifications of behavior, from which FoF can generates C code which is guaranteed to implement the given semantics.

Flounder: The Interface Definition Language used for communication between domains in Barrelfish. Flounder supports message type signatures, various concrete type constructors like structs and arrays, and also some opaque types like capabilities and interface references. The Flounder compiler is written in Haskell and generates code in C or THC. Flounder generates specialized code for each Interconnect Driver in a system.

FoF: See *Filet-o-Fish*.

foreign capability: A capability referring to a resource which only makes sense on a different core to the one it exists on. For example, since the cspace is purely local to core, transferring a CNode capability to another core transforms it into a Foreign CNode capability, whose only operations are to remotely manipulate (principally copy capabilities from) the original CNode.

frame capability: A capability refering to a set of page frames which can be mapped into a virtual address space. Frame capabilities are a subtype of RAM capabilities; the latter cannot be mapped.

Fugu *(errno.h)*: A small domain-specific language (implemented using Filet-o-Fish) to express error conditions and messages for Barrelfish. Fugu offloads the problem of tracking error messages and code, and also implements a short error "stack" in machine word to provide more detailed error information. Fugu generates the `errno.h` file.

Hake *(Hakefile)*: The build system for Barrelfish. The Barrelfish source tree consists of a Hakefile in each relevant source directory, which contains a single Haskell expression specifying a list of targets to be built. Hake itself aggregates all these Hakefiles, and uses them to generate a single Makefile in a separate, build directory. This Makefile contains an explicit target for *every* file that can be built for Barrelfish for every appropriate architecture. A separate, manually-edited Makefile contains convenient symbolic targets for creating boot images, etc. Since the contents of Hakefiles are Haskell expressions, quite powerful constructs can be used to specify how to build files for different targets.

Hakefile: See *Hake*.

Hamlet: The language used to specify all Barrelfish capability types, together with their in-memory formats and transformation rules when transferring a capability from one core to another. Hamlet is implemented using Filet-o-Fish and generates capability dispatch code for CPU drivers, among other things. Surprisingly, Hamlet is a type of fish.

ICD: See *interconnect driver*.

interconnect driver *(ICD)*: A partial abstraction of low-level message passing mechanism, such as a shared-memory buffer or hardware message passing primitive. ICDs do not present a uniform interface, and therefore require specific stubs to be generated by Flounder. Similarly, they do not present any particular semantics with regard to flow control, polled or upcall-based delivery, etc. A special case of an ICD is the local message passing (LMP) mechanism, which is implemented for every type of core and provide communication between dispatchers running on that core.

IO capability: A capability enabling the holder to perform IO space operations on Intel architecture machines. Each IO capability grants access to a range of IO space addresses on a specific core.

Kaluga: The Barrelfish device manager. Kaluga is responsible to starting and stopping device drivers in response to hardware events, and based on configurable system policies.

KCB: See *kernel control block*.

kernel capability: A capability enabling the holder to manipulate CPU driver data structures (in particular, the capability database). The kernel capability for a core is only held by the core's monitor.

kernel control block *(KCB)*: The kernel object representing all per-core state. KCBs are referred to by special capability types.

LMP: See *Local Message Passing*.

Local Message Passing *(LMP)*: Each Barrelfish CPU driver includes a special Interconnect Driver for passing messages between dispatchers on the same core, often based on the concepts from LRPC and L4 IPC. This is referred to as LMP.

Mackerel: The Domain Specific Language used in Barrelfish to specify device hardware registers and hardware-defined in-memory data structures. The Mackerel compiler takes such a description and outputs a C header file containing inline functions to read and write named bitfields for a device or data structure, together with string formatting code to aid in debugging. Mackerel input files are found in the `/devices` directory of the source tree, and end in `.dev`. Generated header files are found in the `/include/dev` subdirectory of the build tree.

Mapping Database: The mapping database is used to facilitate retype and revoke operations. A capability that is not of type dispatcher, can only be retyped once. The mapping database facilitates this check. When a capability is revoked, all its descendants and copies are deleted. The mapping database keeps track of descendants and copies of a capability allowing for proper execution of a revoke operation. Each core has a single private mapping database. All capabilities on the core must be included in the database.

monitor: A privileged process running on a core which handles most core OS functionality not provided by the CPU driver. Since CPU drivers do not directly communicate with each other, the task of state coordination in Barrelfish is mostly handled by Monitors. Monitors are also responsible for setting up inter-core communication channels ("binding"), and transferring capabilities between cores. When a capability is retyped or revoked, the monitors are responsible for ensuring that this occurs consistently system-wide. The monitor for a core holds a special capability (the kernel capability) which allows it to manipulate certain data structures in the CPU driver, such as the capability database.

ND: See *notification driver*.

notification driver *(ND)*: A partial abstraction of a low-level message passing mechanism which performs notification (sending an event), rather than transferring data per se. In Barrelfish these mechanisms are separated where possible for flexibility and to further decouple sender and receiver. Notification drivers exist in Barrelfish for sending inter-processor interrupts on most architectures, for example.

Octopus: A service built on (and colocated with) the SKB which provides locking functionality inspired by Chubby and Zookeeper, and publish/subscribe notification for Barrelfish processes.

physical address capability: A capability referring to a raw region of physical address space. This is the most basic type of capability; all other capability types which refer to memory are ultimately subtypes of this.

Pleco: The Domain Specific Language used in Barrelfish to specify constants for the tracing infrastructure. The Pleco compiler takes such description and outputs a C header file containing the definitions, a C source file with the constants, and a JSON file to be used by host visualization tools.

RAM capability: A capability type which refers to a region of Random Access Memory. A direct subtype of a Physical Address capability.

scheduler manifest: A description of the scheduling requirements of multiprocessor application, used to inform the various system schedulers about how best to dispatch the application's threads.

SKB: See *System Knowledge Base*.

System Knowledge Base *(SKB)*: A repository for hardware-derived system information in a running Barrelfish system. The SKB is currently a port of the ECLiPse Constraint Logic Programming system, and is used for (among other things) PCI bridge programming, interrupt routing, and constructing routing trees for intra-machine multicast. The SKB runs as a system service accessed by message passing.

THC: A dialect of C with extensions to support the asynchronous message passing flavor of most Barrelfish services by providing an `async` construct and continuations. THC also integrates with specially generated Flounder stubs.

UMP: See *User-level Message Passing*.

User-level Message Passing *(UMP, CC-UMP)*: A series of Interconnect Drivers which use cache-coherent shared memory to send cache-line sized frames between cores. UMP is based on ideas from URPC, and avoids kernel transitions on message send and receive. It is the preferred channel for communication between cores on an Intel-architecture PC, for example. However, because it operates entirely in user space, it cannot send capabilities between cores. Instead, the Flounder stubs for UMP send capabilities via another channel through LMP to the local monitor, which forwards them correctly to the destination.

vnode capability: One of a set of capability types, one for each format of page table page for each architecture supported by Barrelfish. For example, there are four Vnode capability types for the x86-64 architecture, corresponding to the PML4, PDPT, PDIR, and PTABLE pages in a page table.

vspace: An object representing a virtual address space. Unlike a cspace, a vspace can under some circumstances be shared between cores. However, a vspace is tied to a particular core architecture, and a particular physical address space, though vspaces can be manipulated on other cores and even cores of other architectures. Mappings are created in a vspace by specifying capabilities to regions of memory that are mappable (such as frame capabilities).

ZZZ terms yet to be added: asmoffsets, retype, iref