CHERIoT Programmers' Guide

David Chisnall

Table of Contents

This is a public draft of the CHERIoT Programmers' Guide.

This is an early draft. It is public for early feedback and to help people get started with the CHERIoT Platform. It is guaranteed to contain errors, both factual and typographic.

Copyright

Chapter 1. CHERIoT Concepts

The CHERIoT platform is an embedded environment that provides a number of low-level features via a mixture of hardware and software features.

1.1. Understanding CHERI capabilities

A capability, in the abstract sense, is an unforgeable token of authority that must be presented to perform an operation. Capabilities exist in the physical world in various forms. For example, a key to a padlock is a capability to unlock the padlock. When the key is presented, the padlock can be unlocked, without the key the padlock cannot be unlocked without exploiting some security vulnerability. It doesn't matter to the padlock who presents the key, only that the correct key has been presented. Some complex building locks have different keys that authorise unlocking different sets of doors. For example, a team leader may have a key that unlocks the offices of everyone on their team and the building manager may hold a key that unlocks everything.

Capabilities can be delegated. The building manager may loan their key to someone else to unlock a door. The key and the door don't care who is holding them.

Some kinds of capabilities can also be *revoked*. This is traditionally the hardest operation to perform on capabilities. Someone may perform an audit of all of the keys and remove some of them from people that shouldn't have them. This is often solved in capabilities by adding a layer of indirection; you hold a capability that authorises you to do something with a capability in some table and your rights can be removed by deleting the capability from the table.

On a CHERI system, capabilities are used to authorise access to memory. Any instruction that takes an address in a conventional architecture takes a *CHERI capability* as the operand instead. The CHERI capability both describes a location in memory and grants access to it.

You can think of a CHERI memory capability as a pointer that the hardware understands. In C, if you hold a pointer to an object then you are allowed to access the object that it points to. If you do some pointer arithmetic that goes out of bounds, C says that this is undefined behaviour. CHERI says more concretely that it will trap: you are not authorised to access that memory *with this capability*. If you hold two pointers to objects that are adjacent in memory, then you may be authorised to access the memory, but not with the pointer that you are using. This highlights the two key security principles that capability systems are able to enforce:

- **The principle of least privilege**, which states that a piece of running code should have the rights to do what it needs to do and no more.
- **The principle of intentional use**, which states that any privileged operation must be performed by intentionally exercising the specific right that is needed.

Capability systems make it easy to implement least privilege by providing running code only with the minimal set of capabilities (with the limited set of rights) that they need. They make it easy to implement intentionality by requiring the specific capability to be presented along with each operation. The latter avoids a large category of confused deputy attacks, where a component holding one privilege is tricked into exercising it on behalf of a differently trusted component.

In a CHERIoT system, **every** pointer in a higher-level language such as C, and every implicit pointer (such as the stack pointer, global pointer, and so on) used to build the language's abstractions, is a CHERI capability. If you have used other CHERI systems then you may have seen a hybrid mode, where only some pointers are capabilities and others are integers relative to an implicit capability. CHERIoT does not have this hybrid mode. The hybrid mode is intended for running legacy binaries but makes it harder to provide fine-grained sandboxing. CHERIoT assumes all code will be recompiled for the new target.

The phrase 'differently trusted' in the previous paragraph is not an attempt to extend political correctness to software components. Capability systems do not imply hierarchical trust models. Two components may hold disjoint or overlapping sets of capabilities that allow each to perform some set of actions that the other cannot.

In a CHERI system, this can include one component having read access to an object and another write access, or two components having access to different fields of the same structure.

1.2. Decomposing permissions in CHERIoT

Any CHERI system provides a set of permissions on capabilities. Most existing CHERI systems have 64-bit addresses (and therefore 128-bit capabilities) and so have a lot of space for permissions as an orthogonal bitfield. The CHERIoT platform has 32-bit addresses (and therefore 64-bit capabilities) and so has to compress the permissions. This is done, in part, by separating the permissions into primary and derived permissions. The primary permissions (listed in Table 1) have meaning by themselves. If you use the CHERIoT RTOS logging support to print capabilities, the permissions will be listed using the letters in the first column.

Table 1. CHERIoT primary permissions

Debug output letter	Permission name	Meaning
G	Global	May be stored anywhere in memory.
R	Load (Read)	May be used to read.
W	Store (Write)	May be used to write.
X	Execute	May be used to as a jump target (executed).
S	Seal	May be used to seal other capabilities (see Section 1.4).
U	Unseal	May be used to unseal sealed capabilities.
0	User 0	Reserved for software use.

Read and write permission allow the capability to be used as an operand to load and store instructions, respectively. Execute allows the capability to be used as a jump target, where it will end up installed as the *program counter capability* and used for instruction fetch. We'll cover the sealing and unsealing permissions later.

The derived permissions (listed in Table 2) provide more fine-grained control. For many of these, it's more useful to think about what can't be done if you lack the permission than to think about

what can be done if you have it.

Table 2. CHERIoT derived permissions

Debug output letter	Permission name	Depends on	Meaning
c	Load / Store Capability	R / W	May be used to load or store capabilities as well as non-capability data.
g	Load Global	R	May be used to load capabilities with the global permission.
m	Load Mutable	R	May be used to load capabilities with write permission.
l	Store local	W	May be used to store capabilities that do *not* have global permission.
a	Access System Registers	X	Code run via this capability may access reserved special registers.

By default, the load and store permissions authorise instructions to load and store non-capability data. With the load / store capability permission, they also allow loading and / or storing capabilities. Removing this permission is useful for pure-data buffers. You can't accidentally store a valid pointer into them, and if they already contain a valid pointer then no one can load it.

Load global allows loading permissions with the global permission. Any capability loaded via a capability without this permission will have its global (and load-global) permission stripped. It can then be stored only via a capability that has the store-local permission. CHERIoT RTOS provides the store-local permission exclusively to stacks. This means that removing global gives a shallow no-capture pointer (code that it is passed to can store it in registers and on the stack but nowhere else), removing load-global gives a deep no-capture (not even objects loaded by an arbitrary amount of pointer chasing can be captured).

Similarly, the load-mutable permission allows loading writable permissions and will strip write and load-mutable permissions from any capability that is loaded. Removing write permission will give a shallow immutability (the object pointed to by this pointer cannot be modified), also removing load-mutable will give a deep immutability. The latter can be used for read-only sharing of complex data structures.

The access-system-registers permission controls access to a small number of privileged registers and is never handed out to code other than a tiny amount of TCB.

1.3. Building memory safety

Memory safety is a property of a source-level abstract machine. Memory safety for C, Java, or Rust mean different things. At the hardware level, CHERIoT is designed to enable implementations of languages to enforce memory safety, in the presence of untrusted code such as inline assembly or code written in a different language. Most importantly, it provides the tools that allow code in a compartment (see Section 1.6) to protect itself from arbitrary code in a different compartment. This

means protecting objects such that code from a different security context cannot:

- Access object unless passed pointers to them.
- Access outside the bounds of an object given a valid pointer to that object.
- Access an object (or the memory that was formerly used for the object) after the object has been freed.
- Hold a pointer to an object with automatic storage duration (an 'on-stack' object) after the end of the call in which it was created.
- Hold a temporarily delegated pointer beyond a single call.
- Modify an object passed via immutable reference.
- Modify any object reachable from an object that is passed as a deeply immutable reference.
- Tamper with an object passed via opaque reference.

The hardware provides tools for enforcing all of these properties but it's up to the compiler and the RTOS to cooperate to use them correctly. For example, in the CHERIoT ABI, each compartment has a single capability in a register that spans all of its globals and a single capability that grants access to its entire stack. The compiler will derive capabilities from these that are bounded to individual globals or on-stack objects. Inline assembly that references the global-pointer or stack-pointer registers directly can bypass spatial memory safety for these objects, but only from within the same compartment.

None of the properties relating to heap objects make sense in the absence of a heap. CHERIoT RTOS provides a shared heap (see Chapter 6), which enforces spatial and temporal safety for heap objects.

1.4. Sealing pointers for tamper proofing

We have discussed all of the primary permissions from Table 1 with the exception of those related to *sealing*. Sealing a capability transforms it from something that conveys rights and can be used to exercise those rights into an opaque token. It can be transformed back with the converse unseal operation.

A sealed capability has an *object type* associated with it. This is taken from the value (the part that would be the address in a memory capability) in the capability that authorises sealing. It can then be unsealed only with a capability that has the same value and the permit-unseal permission.

If you attempt to unseal a capability that is not sealed with the value of the permit-unseal capability then you will get back an untagged value. Sealed capabilities can therefore be used as trusted handles that can be shared with untrusted code. If the untrusted code tries to modify the value in any way, you can detect the tampering.

The CHERIoT encoding has space for only three bits of object type (in contrast with 'big CHERI' systems such as Morello that typically have 18 bits). This is sufficient for a small number of core parts of the ABI but not enough for general-purpose use. To mitigate this limitation, the memory allocator provides a set of APIs (see Section 6.7) that virtualise the sealing mechanism. The same mechanism is also used to build software-defined capabilities.

The object type in a CHERIoT capability is interpreted differently depending on whether the sealed capability is executable or not. For executable capabilities, most of the object types are reserved for sealed entry (*sentry*) capabilities. A sentry capability can be unsealed automatically by jumping to it. Return addresses are automatically sealed by the jump-and-link instructions, so you cannot modify a return address, you can only jump to it.

CHERIoT v1 does not currently differentiate between forward and backwards sentries. This is a limitation inherited from RISC-V, which lacks an explicit return instruction and so has no convenient mechanism for determining whether a branch is intended as forward or backwards control flow. This will be addressed in a future version of the CHERIoT ISA.

Sentries are also used as a building block for cross-compartment calls. A sentry can point to a region of memory that contains both code and data. The data is accessible via PC-relative addressing only after jumping into the code.

1.5. Controlling interrupt status with sentries

In conventional RISC-V (and most other architectures) the interrupt status is controlled via a special register. This register can be modified only in some privileged mode. The CHERIoT ISA allows it to be modified by any code running with the access-system-registers permission in the program counter capability.

Embedded software often wants to disable interrupts for short periods but granting the permission to toggle interrupts makes auditing availability guarantees between mutually distrusting components almost impossible. Instead, CHERIoT provides three kinds of sentries that control the interrupt status. These either enable or disable interrupts, or leave the interrupt enabled state untouched. The branch-and-link instruction captures the current exception state in the return sentry.

This allows you to provide function pointers to functions that will run with interrupts disabled and guarantee that, on return, the interrupt status is reset as it should be. In effect, this brings structured programming to interrupt status.

In the RTOS, for example, the atomics library provides a set of functions that (on single-core systems without hardware atomics) perform simple read-modify-write operations with interrupts disabled. A compartment can use these without having the ability to arbitrarily toggle interrupts, giving a limit on the amount of time that it can run with interrupts disabled.

1.6. Isolating components with threads and compartments

Most mainstream operating systems have a process model that evolved from mainframe systems. This is built around isolation, with sharing as an afterthought. The primary goal for process isolation was to allow consolidation, replacing multiple minicomputers with a single mainframe. These abstractions were designed with the assumption that they ran independent workloads that wanted to share computational resources. Gradually, communication mechanisms have been added

on top.

CHERIoT starts from a fundamental assumption that **isolation is easy, (safe) sharing is hard**. Particularly in the embedded space, it's easy to provide a separate core and SRAM if you want strong isolation without sharing. Most useful workloads involve communication between distrusting entities. For example, if you want to connect an IoT device to a back-end service, your ethernet driver needs to communicate with the TCP/IP stack, which needs to communicate with the TLS stack, which needs to communicate with a higher-level protocol stack such as MQTT, which needs to communicate with your device-specific logic.

CHERIoT provides two composable abstractions for isolation.

- Compartments are units of spatial isolation
- Threads are units of temporal isolation

A compartment owns some code and some globals. It exports a set of functions as entry points and may import some entry points from other compartments. A thread owns a register state and a stack and is a schedulable entity.

At any given point, the core is executing one thread in one compartment. Threads move between compartments via function call and return. When code in one compartment calls another, it loses access to everything that was not explicitly shared. Specifically:

- All registers except argument registers are zeroed.
- The stack capability is truncated to exclude the portion used by the caller.
- The portion of the stack that is delegated from the caller to the callee is zeroed.

On return, the stack becomes accessible again but a similar set of state clearing guarantees confidentiality from the callee to the caller.

Arguments that are passed from one compartment to another may include capabilities. At the start of execution, each compartment has a guarantee that nothing else can see or modify its globals. If one compartment passes a pointer to one of its globals to another, you now have shared memory. This can be useful with restricted permissions for sharing read-only epoch counters and similar.

1.7. Sharing code with libraries

Invoking reusable components does not always involve a change of security context. The CHERIoT software model provides *shared libraries* for cases where this is the case.

Unlike compartments, shared libraries do not have mutable globals. They are reusable code and read-only data, nothing else. They are invoked via a much lighter-weight mechanism than a full cross-compartment call. This mechanism doesn't clear the stack or registers.

Using a CHERIoT shared library is conceptually equivalent to copying the code that implements it into every compartment that uses it. Unlike simple copying, shared libraries are independently auditable and require only a single copy of the code in memory.

All entry points exported from a shared library are invoked via sentries. This means that they can

enable or disable interrupts for the duration of the call.

Some shared libraries expose very simple functions, others are a lot more complex. For example, the atomics library provides some functions that are only a handful of instructions long. In contrast, shared library that packages Microvium provides a complete JavaScript interpreter.

1.8. Auditing firmware images

When a CHERIoT firmware image starts, the loader initialises all of the capabilities that compartments hold at boot. It does this using metadata provided by the linker. This means that everything that leads to capabilities being provided is visible to the linker. The CHERIoT linker, in addition to providing the firmware image, provides a report about this structure. The report includes:

- The hashes of the sections that form each compartment.
- The list of exports from each compartment and each library.
- The list of functions imported for each compartment and each library.
- Whether each entry point runs with interrupts enabled, disabled, or inherited.
- The list of MMIO regions accessible by any compartment.
- How much memory each compartment is permitted to allocate.
- The initial entry point, stack size, and priority for each thread.

This allows automated auditing of various high-level security policies. For example, you can check that a single compartment, containing a known binary (for example, one that has been approved by regulators), is the only thing that is able to access a specified device. You can require that nothing runs with interrupts disabled except a specific set of permitted library functions. Or you can say that users can provide their own logic for controlling their IoT device, but only your code may connect to the network stack if you want to sign the image with a key that authorises release of a private key.

Chapter 2. The RTOS Core

The core of the RTOS is a set of privilege-separated components. Each core component runs with some privileges that mean that it is (at least partially) in the trusted computing base (TCB) for other things.

2.1. Starting the system with the loader

The *loader* runs on system startup. It reads the compartment headers and populates each compartment with the set of capabilities that it needs. If a compartment contains a global that is a pointer, initialised to point to another global, the loader will initialise pointer by deriving a capability from one out of the compartment's code or data capabilities. In addition, cross-compartment and cross-library calls and imported MMIO regions all require capabilities to be set up, as does the initial register file for each thread.

The loader is the most privileged component in the system. When a CHERIoT CPU boots, it will have a small set of *root capabilities* in registers. These, between them, convey the full set of rights that can be granted by a capability. Every capability in the running system is derived (often via many steps) from one of these. As such, the loader is able to do anything.

The risk from the loader is mitigated by the fact that it does not run on untrusted data. The loader operates only on the instructions generated by the linker and so it is possible to audit precisely what it will do. It is also possible to validate this by running the loader in a simulator and capturing the precise memory state after it has run.

The loader enforces some of the guarantees in the initial state. It is structured to be able to enforce some of these by construction. For example, only stacks and trusted stacks (accessible only by the switcher, see Section 2.2) have store-local permission and these do not have global permission. The scheduler derives these from a capability that has store-local but not global permissions and derives all other capabilities from one that has the store-local permission removed.

Before starting the system, the loader erases almost all of its code (leaving the stub that handles this erasure), its stack, and clears its registers. The last bit of the loader's code becomes the idle thread (a wait-for-interrupt loop). The loader's stack is used for the scheduler stack. The memory that held loader's code is used for heap memory.

2.2. Changing trust domain with the switcher

The *switcher* is the most privileged component that runs after the system finishes booting. It is responsible for transitions between threads (context switches) and between compartments (cross-compartment calls and returns). The switcher is a very small amount of code—under 500 instructions—that is expected to be amenable to formal verification.

The switcher is the only component in a running CHERIoT system that has access-system-registers permission. It uses this primarily to access a single reserved register that holds the *trusted stack*. The trusted stack is a region of memory containing the register save area for context switches and a small frame for every cross-compartment call that allows safe return even if the callee has corrupted all state that it has access to.

Trusted stacks are set up by the loader. The loader passes the scheduler (see Section 2.3) a sealed capability to each of these on initialisation. The switcher holds the only permit-unseal capability for the type used to seal trusted stacks.

The context switch path spills all registers to the current trusted stack's save area and then invokes the scheduler, which returns a sealed capability to the next thread to run. It then restores the register file from this thread and resumes. If the scheduler returns an invalid capability (one not sealed with the correct type) then the switcher will raise a fault. If the exception program counter capability on exception entry is within the switcher's capability, the switcher will terminate.

On the cross-compartment call path, the switcher is responsible for unsealing the capability that refers to the export table of the callee, clearing unused argument registers, pushing the information about the return to the trusted stack, subsetting the bounds of the stack, and zeroing the part of the stack passed to the callee. On return, it zeroes the stack again, zeroes unused return registers, and restores the callee's state.

This means that the switcher is the only component that has access to either two threads', or two compartments', state at the same time. As such, it is in the TCB for both compartment and thread isolation. This risk is mitigated in several ways:

- The switcher is small. It contains a similar number of instructions to the amount of unverified code in seL4.
- The switcher is defensive. Most errors simply forcibly unwind to the previous trusted stack frame, so a compartment that attempts to attack the switcher exits to its caller.
- Like everything else in the system, it must follow the capability rules. Unlike an operating system running in a privileged mode on mainstream hardware, it does not get to opt out of memory protection, it is not able to access beyond the bounds of capabilities passed to it or access any memory to which it does not have an explicit capability.
- It is largely stateless, all state that it modifies is held in the trusted stack.

2.3. Time slicing with the scheduler

When the switcher receives an interrupt (including an explicit yield), it delegates the decision about what to run next to the *scheduler*. The scheduler has direct access to the interrupt controller but, in most respects, is just another compartment.

The switcher also holds a capability to a small stack for use by the scheduler. This is not quite a full thread. It cannot make cross-compartment calls and is not independently schedulable. When the switcher takes an interrupt, it invokes the switcher's entry point on this stack.

The switcher also exposes other entry points that can be invoked by cross-compartment calls. These fulfil a role similar to system calls on other operating systems, for example waiting for external events or performing inter-thread communication. The scheduler implements blocking operations by moving the current thread from a run queue to a sleep queue and then issuing a software interrupt instruction to branch to the switcher. When the switcher then invokes the scheduler to make a scheduling decision, it will discover that the current thread is no longer runnable and pick another. Once the thread becomes runnable again, the switcher resumes the thread from the point where it yielded, at which point it can return from the scheduler.

The scheduler is, by definition, in the TCB for availability. It is the component that decides which threads run and which do not. A bug in the scheduler (with or without an active attacker) can result in a thread failing to run.

It is not; however, in the TCB for confidentiality or integrity. The scheduler has no mechanism to inspect the state of an interrupted thread. When invoked explicitly, it is called with a normal cross-compartment call and so has no access to anything other than the arguments.

2.4. Sharing memory from the allocator

The final core component is the memory allocator, which provides the heap, which is used for all dynamic memory allocations. This is discussed in detail in Chapter 6. Sharing memory between compartments in CHERIoT requires nothing more than passing pointers (until you start to add availability requirements in the presence of mutual distrust). This means that you can allocate objects (or complex object graphs) from a few bytes up to the entire memory of the system and share them with other compartments.

The allocator has access to the shadow bitmap and hardware revocation engine that enforce temporal safety for the heap, and is responsible for setting bounds on allocated memory. It is therefore trusted for confidentiality and integrity of memory allocated from the heap. If it incorrectly sets bounds, a compartment may gain access to memory belonging to another allocation. If it incorrectly configures revocation state or reuses memory too early then a use-after-free bug may become exploitable.

The allocator is not able to bypass capability permissions, it simply holds a capability that spans the whole of heap memory. As such, it is in the TCB only with respect to heap allocations. It cannot access globals (or code), held in other compartments and so a compartment that does not use the heap does not need to trust the allocator.

2.5. Building firmware images

CHERIoT RTOS uses the xmake build system. Xmake is a build system implemented in Lua. It was chosen because it is easy to add new kinds of build targets.

In a typical system that uses the compile-link process invented by Mary Allen Wilkes in the '60s, you compile source files to object code and then link object code to produce executables. You may have an intermediate step that produces libraries.

The CHERIoT build process was designed to enable separate compilation and binary distribution of components. Each source file is compiled either for use in a shared library or for use in a specific compartment. This means that, when building compartments, the compiler invocation must know the compartment in which the object file will be used.

Next, compartments and libraries are linked. This requires a special invocation of the linker that produces a relocatable object file with the correct structure. At this point, the only exported symbols are those for exported functions and the only undefined symbols should be those for MMIO regions or exports from other compartments (see Chapter 4 for more information).

The build system produces a `.library` or `.compartment` file for each shared library and each compartment. In theory, these can be distributed as binaries and linked into a firmware image but this is not yet handled automatically by the build system.

The final link step produces a firmware image. It also produces the JSON report that describes all cross-compartment interactions and is used for auditing.

Using the RTOS build system involves writing an `xmake.lua` file that describes the build. This starts with some boilerplate:

```
set_project("CHERIoT examaple project")
sdkdir = "../../sdk"
includes(sdkdir)
set_toolchains("cheriot-clang")
```

The first line gives a name to the project. The next three import the build system components from the CHERIoT RTOS SDK. The `sdkdir` variable should point to the location of the `sdk` directory from the RTOS repository. The other lines should be reproduced verbatim.

Next, you may wish to include some shared libraries from the RTOS. Each of these libraries has an `xmake.lua` that you can include by listing the directory containing it as an argument to `includes`. A typical image will include something like this:

```
-- Support libraries
includes(path.join(sdkdir, "lib/freestanding"),
         path.join(sdkdir, "lib/atomic"),
         path.join(sdkdir, "lib/crt"))
```

This includes the core definitions for a freestanding C implementation (`memcpy` and friends), the atomic helpers for cores without atomic instructions, and the C runtime things that are called from compiler builtins. See the `lib` directory in the SDK for a full list.

Next you need to provide the option for selecting the board:

```
option("board")
    set_default("sail")
```

You can set a default here. This refers to a board description file. If you're targeting a particular hardware platform, setting the default here allows users to avoid specifying it manually on every build.

Next, you need to add any compartments and libraries that are specific to this firmware image. In most cases, you can do this in just two lines, the first providing the name of the compartment and the second providing the list of files:

```
compartment("example")
    add_files("example.cc")

compartment("mylib")
    add_files("lib.cc")
```

The name of the compartment in the `xmake.lua` must match the name used for the exported function as described in Section 4.6. If they do not match, the compiler will raise an error that a function is defined in the wrong compartment.

Finally, you must provide a `firmware` block that defines the complete integration:

```
-- Firmware image for the example.
firmware("hello_world")
    add_deps("crt", "freestanding", "atomic_fixed")
    add_deps("example", "mylib")
    on_load(function(target)
        target:values_set("board", "$(board)")
        target:values_set("threads", {
            {
                compartment = "hello",
                priority = 1,
                entry_point = "say_hello",
                stack_size = 0x200,
                trusted_stack_frames = 1
            }
        }, {expand = false})
    end)
```

The `add_deps` lines refer to `library` and `compartment` blocks defined earlier. The first refers to the ones imported from the SDK, the second to the ones from this example. The `on_load` block then sets the CHERIoT-specific configuration. The `board` key is set to the value from the option that we defined. You can hard-code this to a single board or provide multiple firmware targets for different boards if required.

The `threads` key is set to an array of thread descriptions. Each thread must set five properties:

`compartment`
: The compartment in which this thread starts.

`entry_point`
: The name of the function for this thread's entry point. This must be a function that takes and returns `void`, exported from the compartment specified by the `compartment` key.

`priority`
: The priority of this thread. Higher numbers indicate higher priorities.

`stack_size`

The number of bytes of stack space that this thread has allocated.

`trusted_stack_frames`

The number of trusted stack frames. Each cross-compartment call pushes a new frame onto this stack and so this defines the maximum number of compartments call depth (including the entry point) for this thread.

Chapter 3. C/C++ extensions for CHERIoT

The CHERIoT platform adds a small number of C/C++ annotations to support the compartment model.

3.1. Exposing compartment entry points

Compartments are discussed in detail in Chapter 4. A compartment can expose functions as entry points via a simple attribute.

The `cheri_compartment({name})` attribute specifies the name of the compartment that defines a function. This is used in concert with the `-cheri-compartment=` compiler flag. This allows the compiler to know whether a particular function (which may be in another compilation unit) is defined in the same compartment as the current compilation unit, allowing direct calls for functions in the same compilation unit and cross-compartment calls for other cases.

This can be used on either definitions or declarations but is most commonly used on declarations.

If a function is defined while compiling a compilation unit belonging to a different compartment then the compiler will raise an error. In CHERIoT RTOS, this attribute is always used via the `__cheri_compartment({name})` macro. This makes it possible to simply use `#define __cheri_compartment(x)` when compiling for other platforms.

3.2. Exposing library entry points

Libraries are discussed in Chapter 4. Like compartments, they can export functions, via a simple annotation.

The `cheri_libcall` attribute specifies that this function is provided by a library (shared between compartments). Libraries may not contain any writeable global variables. This attribute is implicit for all compiler built-in functions, including `memcpy` and similar freestanding C environment functions. As with `cheri_compartment()`, this may be used on both definitions and declarations.

This attribute can also be used via the `__cheri_libcall` macro, which allows it to be defined away when targeting other platforms.

3.3. Passing callbacks to other compartments.

The `cheri_ccallback` attribute specifies a function that can be used as an entry point by compartments that are passed a function pointer to it. This attribute must also be used on the type of function pointers that hold cross-compartment invocations. Any time the address of such a function is taken, the result will be a sealed capability that can be used to invoke the compartment and call this function.

The compiler does not know, when calling a callback, whether it points to the current compartment. As such, calling a CHERI callback function will **always** be a cross-compartment call, even if the target is in the current compartment.

This attribute can also be used via the `__cheri_callback` macro, which allows it to be defined away when targeting other platforms.

3.4. Interrupt state control

The `cheri_interrupt_state` attribute (commonly used as a C++11 / C23 attribute spelled `cheri::interrupt_state`) is applied to functions and takes an argument that is either:

- `enabled`, to enable interrupts when calling this function.
- `disabled`, to disable interrupts when calling this function.
- `inherit`, to not alter the interrupt state when invoking the function.

For most functions, `inherit` is the default. For cross-compartment calls, `enabled` is the default and `inherit` is not permitted.

The compiler may not inline functions at call sites that would change the interrupt state and will always call them via a sentry capability set up by the loader. This makes it possible to statically reason about interrupt state in lexical scopes.

If you need to wrap a few statements to run with interrupts disabled, you can use the convenience helper with_interrupts_disabled. This is annotated with the attribute that disables interrupts and invokes the passed lambda. This maintains the structured-programming discipline for code running with interrupts disabled: it is coupled to a lexical scope.

```
template<typename T>
auto with_interrupts_disabled(T && fn)
```

Invokes the passed callable object with interrupts disabled.

3.5. Importing MMIO access

The `MMIO_CAPABILITY({type}, {name})` macro is used to access memory-mapped I/O devices. These are specified in the board definition file by the build system. The `DEVICE_EXISTS({name})` macro can be used to detect whether the current target provides a device with the specified name.

The `type` parameter is the type used to represent the MMIO region. The macro evaluates to a `volatile {type} *`, so `MMIO_CAPABILITY(struct UART, uart)` will provide a `volatile struct UART *` pointing (and bounded) to the device that the board definition exposes as `uart`.

3.6. Manipulating capabilities with C builtins

The compiler provides a set of built-in functions for manipulating capabilities. These are typically of the form `__builtin_cheri_{noun}_{verb}`. You can read all of the fields of a CHERI capability with `get` as the verb and the following nouns:

`address`

The current address that's used when the capability is used a pointer.

`base`

The lowest address that this authorises access to.

`length`

The distance between the base and the top.

`perms`

The architectural permissions that this capability holds.

`sealed`

Is this a sealed capability?

`tag`

Is this a valid capability?

`type`

The type of this capability (zero means unsealed).

The verbs vary because they express the *guarded manipulation* guarantees for CHERI capabilities. You can't, for example, arbitrarily set the permissions on a capability, you can only remove permissions. Capabilities can be modified with the nouns and verbs listed in Table 3.

Table 3. CHERI capability manipulation builtin functions

Noun	Modification verb	Operation
`address`	`set`	Set the address for the capability.
`bounds`	`set`	Sets the base at or below the current address and the length at or above the requested length, as closely as possible to give a valid capability
`bounds`	`set_exact`	Sets the base to the current address and the length to the requested length or returns an untagged capability if the result is not representable.
`perms`	`and`	Clears all permissions except those provided as the argument.
`tag`	`clear`	Invalidates the capability but preserves all other fields.

Setting the object type is more complex. This is done with `__builtin_cheri_seal`, which takes an authorising capability (something with the permit-seal permission) as the second argument and sets the object type of the result to the address of the sealing capability. Conversely, `__builtin_cheri_unseal` uses a capability with the permit-unseal capability and address matching the object type to restore the original unsealed value.

3.7. Comparing capabilities with C builtins

By default, the C/C++ `==` operator on capabilities compares only the address.

This is subject to change in a future revision of CHERI C. It makes porting some existing code easier, but breaks the substitution principle (if `a == b`, you would expect to be able to use `b` or `a` interchangeably).

You can compare capabilities for exact equality with `__builtin_cheri_equal_exact`. This returns true if the two capabilities that are passed to it are identical, false otherwise. Exact equality means that the address, bounds, permissions, object type, and tag are all identical. It is, effectively, a bitwise comparison of all of the bits in the two capabilities, including the tag bits.

Ordered comparison, using operators such as less-than or greater-than, always operate with the address. There is no total ordering over capabilities. Two capabilities with different bounds or different permissions but the same address will return false when compared with either `<` or `>`.

This is fine according to a strict representation of the C abstract machine because comparing two pointers to different objects is undefined behaviour. It can be confusing but, unfortunately, there is no good alternative. Comparison of pointers is commonly used for keying in collections. For example, the C++ `std::map` class uses the ordered comparison operators for building a tree and relies on it working correctly for keys that are pointers. Ideally, these would explicitly operate over the address, but that would require invasive modifications when porting to CHERI platforms.

In general, in new code, you should avoid comparing pointers for anything other than exact equality, unless you are certain that they have the same base and bounds. Instead, be explicit about exactly what you are testing. Do you care if the permissions are different? Do you care about the bounds? Do you care if the value is tagged? Or do you just want to care about the address? In each case, you should explicitly compare the components of the capability that you care about.

You can also compare capabilities for subset relationships with `__builtin_cheri_subset_test`. This returns true if the second argument is a subset of the first. A capability is a subset of another if every right that it conveys is held by the other. This means the bounds of the subset capability must be smaller than or equal to the superset and all permissions held by the subset must be held by the superset.

3.8. Sizing allocations

CHERI capabilities cannot represent arbitrary bases and bounds. The larger the bounds, the more strongly aligned the base and bounds must be.

The current CHERIoT encoding gives byte-granularity bounds for objects up to 511 bytes, then requires one more bit of alignment for each bit needed to represent the size, up to 8 MiB. Capabilities larger than 8 MiB cover the entire address space. This is ample for small embedded systems where most compartments or heap objects are expected to be under tens of KiBs. Other CHERI systems make different trade offs.

Calculating the length can be non-trivial and can vary across CHERI systems. The compiler provides two builtins that help.

The first, `__builtin_cheri_round_representable_length`, returns the smallest length that is larger than (or equal to) the requested length and can be accurately represented. The compressed bounds encoding requires both the top and base to be aligned on the same amount and so there's a corresponding mask that needs to be used for alignment. The `__builtin_cheri_representable_alignment_mask` builtin returns the mask that can be applied to the base and top addresses to align them.

3.9. Manipulating capabilities with `CHERI::Capability`

The raw C builtins can be somewhat verbose. CHERIoT RTOS provides a `CHERI::Capability` class in `cheri.hh` to simplify inspecting and manipulating CHERI capabilities.

These provide methods that are modelled to allow you to pretend that they give direct access to the fields of the capability. For example, you can write:

```
capability.address() += 4;
capability.permissions() &= permissionSet;
```

This modifies the address of `capability`, increasing it by four, and removes all permissions not present in `permissionSet`. Other operations are also defined to be orthogonal.

Permissions are exposed as a `PermissionSet` object. This is a `constexpr` class that provides a rich set of operations on permissions. This can be used as a template parameter and can be used in static assertions for compile-time validation of derivation chains. The loader makes extensive use of this class to ensure correctness.

The equality comparison for `CHERI::Capability` uses exact comparison, unlike raw C/C++ pointer comparison. This is less confusing for new code (it respects the substitution principle) but users may be confused that `a == b` is true but `Capability{a} == Capability{b}` is false.

See `cheri.hh` for more details and for other convenience wrappers around the compiler builtins.

Chapter 4. Compartments and libraries

Compartments in CHERIoT are somewhere between libraries and processes in mainstream operating systems. They have private code and globals and constitute a security boundary. They can export functions to be called form other compartments and can call functions exported from other compartments.

Libraries are a lightweight way of reusing code without duplicating it into different compartments. Calling a library function does not involve crossing a security boundary. Libraries contain code and read-only data but do not have mutable globals. It is possible for libraries to hold secrets but, unless library functions are written in very careful assembly, they should assume that any (immutable) globals in the library can leak to callers. Each library entry point is exposed as a sentry capability (see Section 1.4) to the callers, which means that the caller cannot directly read its code or (immutable) data.

If a library traps, the error handler for the caller compartment may see the register file for the middle of the library. Similarly, the compiler may spill arbitrary values onto the stack or leave them in registers at the end of a library function. As such, you should assume that anything processed in a library written in a compiled language will leak to the caller and anything written in assembly must be **very** careful to avoid leaking secrets. This is not normally a problem because most libraries just exist as an alternative to compiling the same functions into multiple compartments. For example, the functions that implement locks on top of futexes (see Section 5.5) are in a library to reduce overall code size, but simply copying the implementations of these functions into each caller would have no security implications.

4.1. Compartments and libraries export functions

In a UNIX-like system, a shared library can export any kind of symbol. This includes functions and global variables. In CHERIoT, compartments and libraries can export only functions as entry points. Global variables are always private to a compartment or library, unless a pointer is explicitly passed out as a function argument or return in a cross-compartment call. This design is intended to make it easier to reason about sharing between compartments.

If you declare a global in a header and define it in a library or a compartment, you will see linker errors if you try to use it in other compartments or libraries. This hold even for `const` globals exported from libraries. You can place a `static const` global in a header for a library, but that will introduce tight coupling: the value in the header may be inlined at any use site. For very large globals, this may also increase code size significantly.

As mentioned previously, (read-only) globals in a library are hidden in a software-engineering sense, but may be leaked to callers and should not be considered private in a security sense.

You can still use globals to share data but you must explicitly expose them via an accessor. This makes CHERIoT compartments and libraries similar to Smalltalk-style objects, with public methods

and private instance variables.

If you expose an interface that returns a pointer to a global, you can use CHERI permissions to restrict access. Returning a read-only pointer to a global is a common idiom for building a lightweight broadcast communication channel. The owning compartment can write to the global and other compartments can read from via their copy of the pointer, with guarantees that only the owning compartment is making changes.

4.2. Understanding the structure of a compartment

From a distance, a compartment is a very simple construct. The core of a compartment is made of just two capabilities. The program counter capability (PCC) defines (and grants access to) the range of memory covering the compartment's code and read-only globals. This has read and execute permissions. The capability global pointer (CGP) defines (and grants access to) the range of memory covering the compartment's mutable globals.

A future version of the ABI will move read-only globals out of the program counter capability region but this requires some ISA changes to be efficient and so will likely not happen before CHERIoT 2.0.

If a compartment didn't need to interact with anything else, this would be sufficient. In practice, compartments are useful only because they interact with other compartments or the outside world. The read-only data region contains an *import table*. This is the only region of memory that, at system start, is allowed to contain capabilities that grant access outside of the PCC and CGP region for the compartment. The instructions for the loader to populate these are in the firmware image and are amenable to auditing.

The import table contains three kinds of capabilities. MMIO capabilities are conceptually simple: they are just pointers that grant access to specific devices. This mechanism allows byte-granularity access to device registers and so it's possible to provide a compartment with access to a single device register from a large device.

Import tables also contain sentry capabilities for library functions. A shared library has its own PCC region (like a compartment) but does not have a CGP region. Library routines are invoked by loading the sentry from the import table and jumping to it.

Finally, import tables contain sealed capabilities referring to other compartments' *export tables*. If a compartment exports any entry points for other compartments to call, it has an export table. This contains the PCC and CGP for the compartment and a small amount of metadata for each exported function describing:

- The location of the entry point.
- Whether interrupts are enabled or disabled when invoking this function.
- How many argument registers are used (conversely, how many are unused and should be zeroed).

This is all of the information that the switcher needs to transition from one compartment to another.

Extracting code and moving it to a new compartment adds a very small amount of memory overhead, on the order of a dozen words for a typical compartment.

4.3. Adding compartments to the build system

4.4. Choosing a trust model

There are three trust models that are commonly applied to compartments:

Sandbox

A sandbox is a compartment that is used to isolate untrusted code. This model is used to protect the rest of the system. Typically, a sandbox will trust values passed to it as arguments to exported functions or return values from functions that it calls in other compartments.

Safebox

A safebox is a compartment that holds some secret or sensitive data that must be protected from the outside. For example, a safebox may be used to protect a key and perform encryption or signing on behalf of callers. A safebox does not trust any data provided from outside of the compartment, but callers may trust it to behave correctly.

Mutual distrust

Mutual distrust is the strongest model. A compartment in a mutual-distrust relationship protects itself from attacks from the outside by careful handling of inputs and expects other compartments to protect themselves from it in the same way.

This is the start of defining a threat model for your code. A compartment may simply be used for fault isolation, to limit the damage that a bug can do. You may assume that an attacker will be able to compromise some compartments (for example, those directly processing network packets) and defend yourself accordingly.

In the core of the RTOS, the scheduler is written as a safebox. It does not trust anything on the outside and assumes that everything else is trying to make it crash. The memory allocator is also written as a safebox, assuming that everything else is trying to either make it crash or leak powerful capabilities. For some operations, the scheduler invokes the allocator. The scheduler trusts the allocator to enforce heap memory safety. It does not, for example, try to check that the memory allocator is returning disjoint capabilities (it can't see every other caller of heap_allocate, and so couldn't validate this). It is; however, written to assume that other compartments may try to maliciously call allocator APIs to cause it to crash.

When thinking about trust, it's worth trying to articulate the properties that other code is trusted to enforce or preserve. For example, everything in the CHERIoT system trusts the scheduler for availability. Most things trust the allocator to enforce spatial and temporal memory safety for the heap.

4.5. Refining trust

It seems conceptually easy to say 'this code is trusted' and 'this code is untrusted', but that rarely

tells the whole story. At a high level, components are typically trusted (or not) with respect to three properties:

Confidentiality

How does information flow out of this component?

Integrity

What how can information be modified by this component?

Availability

What can this component prevent from working?

Compartments and threads are both units of isolation in a CHERIoT system. Threads are scheduled independently and provide a building block for availability guarantees. Only a higher-priority thread or code running with interrupts disabled can prevent an unrelated thread from making progress.

The relative importance of each of these varies a lot depending on context. For example, you often don't care at all about confidentiality for encrypted data, but you would not want the plain text form to leak and you definitely wouldn't want the encryption key to leak. If you're building a safety-critical system, availability is often key. Dumping twenty tonnes of molten aluminium onto the factory floor will probably kill people and cost millions of dollars, so preventing that is far more important than ensuring that no one unauthorised can inspect the state of your control network.

This kind of model helps understand where you should put compartment boundaries. If an attacker can compromise one component, what damage can they do to these properties in other compartments and in the system as a whole?

For example, consider the simplest embedded application, which just flashes an LED in a pattern. Where should you put compartment boundaries here? You might put the piece that prepares the pattern in one compartment and the part that interacts directly with the LED in another. Doing this does not add security value. Neither compartment is exposed to an attacker and so you're just protecting against bugs. The compartment with direct access to the device is just passing a value from a function argument to the device. It is unlikely that there will be a bug in this code that can affect the rest of the system. Conversely, the code that can call this can do everything that this compartment can do and so you haven't reduced the damage that a bug can cause.

Now imagine a slightly more complex device where, rather than lighting a single LED, you are driving an LED strip that takes a 24-bit colour value for each LED in the strip, encoded as a waveform down a two-wire serial line. If you generate the wrong waveform, you'll get the wrong pattern and so there is an availability property that you can protect by moving the code that pauses and toggles a GPIO pin into a separate driver compartment. This driver routine needs to run with interrupts disabled (context switching in the middle of programming the strip would cause it to reprogram the first part twice). Running with interrupts disabled has availability implications on the rest of the system because nothing else can run while this is happening. If you put the driver in a separate compartment then you are protected in both directions:

- The driver is the only thing that can touch the relevant GPIO pin and so, if the code in that driver is correct, nothing can cause the strip to be incorrectly programmed.

- The driver runs with interrupts disabled but the rest of the application does and so you can audit the driver code to ensure that it doesn't cause problems for anything else that the microcontroller is doing.

This then gives you something to build on if you decide, for example, that you want to be able to update the lighting patterns from the Internet. Now you want to add a network stack to be able to fetch the new patterns and an interpreter to run them. What does the threat model look like?

The network stack is exposed to the Internet and so is the most likely place for an attack to start. If this needs to interact with the network hardware with interrupts disabled then you probably want to put that part in a separate network driver compartment so that an attacker can't cause the network stack to sit with interrupts disabled forever. A lot of common attacks on network stacks will simply fail on a CHERIoT system because they depend on violating memory safety but it's possible that an attacker will find novel techniques and compromise the network stack.

You will want narrow interfaces between the network stack and the TLS stack, so that the worst that an attacker with full control over the network stack compartment can do is provide invalid packets (and an attacker can do that from the Internet anyway). The TLS stack will decode complete messages and forward them to the interpreter compartment. TLS packets have cryptographic integrity protection and so anything that comes through this path is probably safe, unless the TLS compartment is compromised, but putting the interpreter in a separate compartment ensures that invalid interpreter code can provide different colours to the LEDs but can't damage the LEDs and can't launch attacks over the network.

4.6. Exporting functions from libraries and compartments

Functions are exported using the attributes described in Chapter 3. Functions exported from a library are annotated with `cheri_libcall`, those from a compartment with `cheri_compartment()`, with the latter providing the name of the compartment.

If you've written shared libraries on Windows, you may have had to add DLL export and import directives on function prototypes in headers. These are usually wrapped in a macro that allows you to define the export attribute when compiling the library and import when compiling anything else.

The CHERIoT attributes are designed to avoid the need for this by operating in concert with the `-cheri-compartment=` compiler flag. When you compile a C/C++ source file that will end up in a compartment, the compiler knows the compartment that it is being built for. It can therefore generate cross-compartment calls for functions that are in other compartments and direct calls for functions in the same compartment. It can also do some additional error checking and will refuse to compile functions in one compilation unit if they are defined in another.

4.7. Validating arguments

```
template<PermissionSet, bool, bool>
bool check_pointer(auto & ptr,
                   size_t space = sizeof(std::remove_pointer< decltype(ptr)>))
```

Checks that `ptr` is valid, unsealed, has at least `Permissions`, and has at least `Space` bytes after the current offset.

`ptr` can be a pointer, or a smart pointer, i.e., any class that supports a `get` method returning a pointer, and `operator=`. This includes `Capability` and standard library smart pointers.

If the permissions do not include Global, then this will also check that the capability does not point to the current thread's stack. This behaviour can be disabled (for example, for use in a shared library) by passing `false` for `CheckStack`.

If `EnforceStrictPermissions` is set to `true`, this will also set the permissions of passed capability reference to `Permissions`, and its bounds to `space`. This is useful for detecting cases where compartments ask for less permissions than they actually require.

This function is provided as a wrapper for the `check_pointer` C API. It is always inlined. For each call site, it materialises the constants needed before performing an indirect call to `check_pointer`.

If a function that is exported from a compartment takes primitive values as arguments, there's little that an attacker can do other than provide invalid values. For things like integers, this doesn't matter, for enumerations it's important to ensure that they are valid values.

Pointers are more complicated. There are a few things that an attacker can do with pointer arguments to invoke a crash:

- Provide a pointer without write permission for an output operand.
- Provide a pointer without read permission for an input operand.
- Provide a pointer without global permission that must be captured and held across calls.
- Provide a pointer with a length that is too small.
- Provide something that isn't a valid pointer at all.
- Provide a pointer that overlaps your stack as an output argument.

Any of these (or similar attacks) will allow an attacker to cause your compartment to encounter a fault when it tries to use the pointer.

In general, you will want to check permissions and bounds on any pointer argument that you're passed. The check_pointer function helps here. It checks that a pointer has (at least) the bounds and permissions that you expect and that it isn't in your current stack region. If you don't specify a size, the default is the size of the argument type. You can use this to quickly check any pointer that's passed to you.

Checking the pointer is not the only option. A CHERI fault will invoke the compartment's error handler (see Section 4.9) and so it may be possible to recover. Some compartments chose to assume that their arguments are valid and just gracefully clean up if they aren't.

If a pointer refers to a heap location, there is one additional attack possible. In general, a pointer cannot be modified after it's been checked, but the memory that a pointer refers to may be freed. When this happens, the pointer is implicitly invalidated. In some cases, you may simply wish to disallow pointers that point to the heap.

You can check whether a pointer refers to heap memory by calling heap_address_is_valid. If this returns true, you can prevent deallocation by using the *claim* mechanism, described in Section 6.8.

```
_Bool heap_address_is_valid(const void * object)
```

Returns true if `object` points to a valid heap address, false otherwise.

Note that this does *not* check that this is a valid pointer. This should be used in conjunction with check_pointer to check validity. The principle use of this function is checking whether an object needs to be claimed. If this returns false but the pointer has global permission, it must be a global and so does not need to be claimed. If the pointer lacks global permission then it cannot be claimed, but if this function returns false then it is guaranteed not to go away for the duration of the call.

4.8. Ensuring adequate stack space

The stack is shared between compartments invoked on the same thread. The callee has access to the portion of the stack that its callers have not used. This means that a malicious compartment can consume almost all of the stack and then try to force a callee to trap when it tries to use the stack.

Before entering a compartment, the switcher will check the amount of stack space against the required amount in the export table. By default, the compiler will fill this value with the amount that is required by the function that serves as an entry point. This is sufficient for leaf functions, but if your function calls others (and they are not inlined) then this will be insufficient.

You can specify the stack space required by a function by using the `__cheriot_minimum_stack` attribute. This is a function attribute that takes a single argument, the number of bytes of stack space that the function requires. Using this attribute requires you to know how much stack space the function will use.

CHERIoT CPUs include a feature called a stack high-water mark that tracks the amount of stack that is used so that the switcher can avoid zeroing unused portions of the stack. The switcher provides a function, stack_lowest_used_address, that you can call to find the lowest address. You can then use the difference between the top of the stack capability (accessed via the `__builtin_cheri_stack_get` built-in function) to determine how much stack space has been used in a particular invocation of a compartment entry point.

```
ptraddr_t stack_lowest_used_address(void)
```

Returns the lowest address that has been stored to on the stack in this compartment invocation.

This helper checks the amount of stack usage **of the current compartment**. The switcher check is not intended to ensure that the invocation of the current compartment can succeed, only that failures are detectable and recoverable. If you want to ensure that a called compartment **also** has enough stack then you will need to add its stack requirements to those of your compartment.

The `debug.hh` header includes a C++ helper class, `StackUsageCheck`. This takes a template argument allowing it to be disabled, enabled and just log if you use more than the expected amount of stack, or enabled and trap if you use more than the expected amount of stack. This is most commonly used with a macro like this:

```
#define STACK_CHECK(expected) \
      StackUsageCheck<StackMode, expected, __PRETTY_FUNCTION__> stackCheck
```

The `StackMode` template argument is one of `StackCheckMode::Asserting`, `StackCheckMode::Logging`, or `StackCheckMode::Disabled`. Typically, you will use it in logging mode initially, then disabled mode in production. Use it in asserting mode when running representative tests in CI so that it fails if you have increased your stack requirements and not updated the caller.

It's important that the tests that you run in asserting mode have good coverage. It's typically fine for this to be function-granularity coverage: with the exception of variable-length arrays, functions stack usage does not depend on control flow within the function.

It's tempting to enable the stack checks in debug builds. This is usually a bad idea because debug builds include extra checks that increase stack usage. Enabling the stack checks in debug builds will cause you to demand more stack space than a release build actually needs, increasing overall memory pressure.

4.9. Handling errors

Asynchronous interrupts are all routed to the scheduler to wake up the relevant threads and schedule the correct thread. Synchronous faults are (optionally) delivered to the compartment that caused them. These include CHERI exceptions, invalid instruction traps, and so on: anything that can be directly attributed to the current instruction.

To handle these, implement compartment_error_handler in your compartment.

```
enum ErrorRecoveryBehaviour compartment_error_handler(struct ErrorState * frame,
                                                       size_t mcause,
                                                       size_t mtval)
```

The error handler for the current compartment.

A compartment may choose to implement this. If not implemented then compartment faults will unwind the trusted stack.

This function is passed a copy of the register file and the exception cause registers when a fault occurs. The `mcause` value will be one of the standard RISC-V exception causes, or 0x1c for CHERI faults. CHERI faults will encode the CHERI-specific fault code and the faulting register in `mtval`. You can decompose this into its component parts by calling extract_cheri_mtval.

```
std::pair<CauseCode, RegisterNumber> extract_cheri_mtval(uint32_t mtval)
```

Decompose the value reported in the `mtval` CSR on CHERI exception into a pair of `CauseCode` and `RegisterNumber`.

Will return `CauseCode::Invalid` if the code field is not one of the defined causes and `RegisterNumber::Invalid` if the register number is not a valid register number. Other bits of mtval are ignored.

The error handler is called with interrupts enabled, even if interrupts were disabled in the faulting code. Latency-critical code should never depend on the error handler for meeting its timing.

If a called compartment faults and forcibly unwinds then this will be reported as a CHERI fault with no cause (zero) in `mtval`. You can use this to propagate faults up to callers, to track the number of times a cross-compartment call has failed, and so on.

The spilled register file does not contain a tagged value for the program counter capability. This is to prevent library functions that run with interrupts disabled or with access to secrets from accidentally leaking on faults. All other registers will be preserved exactly as they are in the register file.

Error handlers are somewhat similar to UNIX signal handlers, but with some important differences. They are invoked for synchronous faults, not arbitrary event notification. Importantly, they are required only to handle the current compartment's errors. You cannot, for example, call `malloc` in a signal handler because it would deadlock (or corrupt state) if the signal arrives during a call to `malloc` or `free`. In contrast, if a call to `heap_allocate` fails then that error will be handled in the allocator compartment. Your error handler will never be invoked in the middle of a call to the allocator and so it is fine to use error handlers to release

locks and free memory.

At the end of your error handler, you have two choices. You can either ask the switcher to resume, installing your modified register file (rederiving the PCC from the compartment's code capability), or you can ask it to continue unwinding.

Error handling functions are used for resource cleanup. For example, you may wish to drop locks when an error occurs, or you may wish to reset the compartment entirely. The heap_free_all function, discussed in Chapter 6 helps with the latter.

4.10. Conventions for cross-compartment calls

If a compartment faults and force unwinds to the caller then the return registers will be set to `-1`. This makes it easy to use the UNIX convention of returning negative numbers to indicate error codes. The value `-1` is `-ECOMPARTMENTFAIL` and other numbers from `errno.h` can be used to indicate other failures.

A CHERIoT capability is effectively a tagged union of a pointer and 64 bits of data. You can take advantage of this in functions that return pointers to return either an integer or, if the result is not tagged, an error code.

4.11. Building software capabilities with sealing

The CHERI capability mechanism can be used to express arbitrary software-defined capabilities. Recall that a capability, in the abstract, is an unforgeable token of authority that can be presented to allow some action. In UNIX systems, for example, file descriptors are capabilities. A userspace process cannot directly talk to the disk or the network, but if it presents a valid file descriptor to system calls such as `read` and `write` then the kernel will perform those operations on its behalf.

CHERIoT provides a mechanism to create arbitrary software-defined capabilities using the *sealing* mechanism (see Section 1.4). CHERIoT provides almost a few billion sealing types for use with software-defined capabilities. You can allocate one of these dynamically by calling token_key_new.

There is no mechanism to reuse sealing capabilities. As such, once you have allocated 4,278,190,079, you will be unable to create new ones. A 20 MHz core doing nothing other than allocating new sealing capabilities could exhaust this space in around a day. If untrusted code is allowed to allocate dynamic sealing capabilities then you may wish to restrict its access to this API and instead give it access to a wrapper that limits the number that it may allocate.

```
SKey token_key_new(void)
```

Create a new sealing key.

This function is guaranteed to complete unless the allocator has exhausted the total number of sealing keys possible (2^32 - 2^24). After this point, it will never succeed. A compartment that is granted access to this entry point is trusted not to exhaust this resource. If you wish to allow a compartment to seal objects, but do not wish to allow it to allocate new sealing keys, then you should insert a proxy compartment that guarantees that it will call this API once and return a single key to the caller.

The return value from this is a capability with the permit-seal and permit-unseal permissions. Callers may remove one or both of these permissions and delegate the resulting capability to allow other compartments to either seal or unseal the capabilities with this key.

If the sealing keys have been exhausted then this will return `INVALID_SKEY`. This API is guaranteed never to block.

You can also statically register a sealing type with the `STATIC_SEALING_TYPE()` macro. This takes a single argument, the name that you wish to give the type. This name is used both to refer to the static sealing capability is the name that will show up in auditing reports.

You can access the sealing capability within the compartment that exported it using the `STATIC_SEALING_VALUE()` macro. You can also refer to it in other compartments, but *only* when constructing *static sealed objects*. A static sealed object is like a global defined in a compartment, but that compartment can access it only via a sealed capability.

Static sealed objects are declared with `DECLARE_STATIC_SEALED_VALUE` and defined with `DEFINE_STATIC_SEALED_VALUE`. These macros take both the name of the sealing type and the compartment that exposes it as arguments. This ensures that there is no ambiguity and that accidental name collisions don't lead to security vulnerabilities.

Any object created in this way shows up in the audit log. The exports section for the compartment that exposes the sealing key will will contain an entry like this:

```
{
  "export_symbol": "__export.sealing_type.alloc.MallocKey",
  "exported": true,
  "kind": "SealingKey"
},
```

This is cross-referenced with a section like this:

```
{
  "contents": "00100000 00000000 00000000 00000000 00000000 00000000",
  "kind": "SealedObject",
```

```
  "sealing_type": {
    "compartment": "alloc",
    "key": "MallocKey",
    "provided_by": "build/cheriot/cheriot/release/cheriot.allocator.compartment",
    "symbol": "__export.sealing_type.alloc.MallocKey"
  }
},
```

This contains the full contents of the sealed object. You can audit these in a firmware image to ensure that they are valid.

Auditing a hex string is not easy. A future version of CHERIoT RTOS will include tools to map these back to useful types.

This gives a building block that can be used to define arbitrary software-defined capabilities at system start. A compartment that performs some action exposes a sealing type and a structure layout that it expects. Static instances of this structure can be baked into the firmware image and then passed as sealed capabilities into the compartment that wishes to use them as capabilities. They can be unsealed using the token APIs described in Section 6.7.

The token APIs look as if they're provided by the allocator, but `token_obj_unseal` is a fast path implemented as a library. This makes it fast to unseal objects (no cross-compartment call). It also removes any dependency on the allocator from things that rely on static sealing.

The allocator uses the static sealing mechanism to define allocation capabilities. These contain a quota that is decreased on allocation and increased on deallocation. A compartment can allocate memory only if it has an allocation capability and any allocation capability that it holds shows up in the audit report when linking a firmware image.

Chapter 5. Communicating between threads

CHERIoT RTOS provides threads as a core abstraction. Threads run until they either yield or are preempted via an interrupt, and then later resume from the same point. There are a small number of scheduler APIs that allow threads to block; higher-level APIs from other compartments may block by indirectly invoking them.

Remember that, in most respects, the scheduler is just another compartment. It doesn't run with elevated privileges, it makes a decision about which thread to run next but it is not able to see the stacks or register states associated with threads.

5.1. Defining threads

Threads in CHERIoT RTOS cannot be dynamically created. Creating threads at run time would require allocating stacks at run time. The no-capture guarantees that CHERIoT RTOS enforces are based on the guarantee that no code (after the loader) has access to two stacks' memory at a time and so the switcher can zero stacks and avoid leaks. The only store-local capabilities that a thread ever has access to are derived from its current stack. Allowing stack creation would violate that: at least the memory allocator would have access to multiple stacks at once. It would be possible to allocate stacks from a separate pool, but that's not really different from allocating stacks up front and having a small compartment that switches them from one use to another or implements a thread pool. There is an example thread pool in `lib/thread_pool` and `thread_pool.h` that you can either use directly or use as inspiration for your own design, if you want to create new thread-like contexts.

Threads in CHERIoT RTOS are constructed with four properties:

- The size of their stack.
- The size of their *trusted stack*.
- Their priority.
- The entry point where they start executing.

The stack size means the same as on any other platform. Specifically on CHEIRoT, the stack-pointer capability will be bounded to this size (rounded up if necessary for alignment) and any overflow of the stack, even by a single byte, will trap. The trusted stack size is the maximum number of cross-compartment calls that this thread can do. Each cross-compartment call invokes the switcher, which pushes a new frame on the trusted stack describing where to return.

In the current version, each trusted stack frame is three capabilities (24 bytes). A larger trusted stack does not make much difference to total memory consumption.

The priority of threads matters only in relative terms. Like FreeRTOS (and unlike UNIX), higher numbers mean higher priorities. The scheduler has some data structures whose size depends on the number of priorities, so compiling with fewer priorities can make the scheduler smaller.

The entry point is a compartment entry point. It must be exposed as described in Chapter 4. Thread entry points take no arguments and return no arguments.

On most other systems, thread creation functions take a pointer. This does not make sense for threads that are not dynamically created because there is no context for their creation.

5.2. Identifying the current thread

You will sometimes need to know which thread is currently running. This can be for something as simple as debugging but may also be needed for maintaining per-thread data structures. The ID of each thread is stored in the register save area for that thread and the switcher exposes a library call (thread_id_get) to read it.

```
uint16_t thread_id_get(void)
```

Return the thread ID of the current running thread.

This is mostly useful where one compartment can run under different threads and it matters which thread entered this compartment.

This is implemented in the switcher.

Thread IDs start at one (not zero!) because zero is used to indicate the idle thread and so is never visible. The thread_count function returns the number of threads that have been created in the system. This is not decremented when threads exit and so provides the upper bound on the number of threads that may exist. This can be used to size data structures that are indexed by thread ID.

```
uint16_t thread_count()
```

Returns the number of threads, including threads that have exited.

This API never fails, but if the trusted stack is exhausted and it cannot be called then it will return -1. Callers that have not probed the trusted stack should check for this value.

The result of this is safe to cache: it will never change over time.

5.3. Using the `Timeout` structure

Several RTOS APIs have timeouts. These are expressed as a pointer to a `Timeout` structure. This design is intended to allow a single timeout to be passed down a chain of operations.

Timeouts represent time spent blocking (yielding waiting to be runnable), not time spent running (doing useful work).

Timeouts measure time in scheduler `Ticks`. A tick is a single scheduling quantum, which depends on the board configuration. This is the minimum amount of time for which it is plausible for a

thread to sleep. If a thread sleeps then another thread becomes runnable and is then allowed to run (unless it also yields) for one tick.

At the end of each tick, the scheduler receives the timer interrupt and chooses the next thread to run. Threads may only run at all if no higher-priority thread is runnable. Threads at the same priority level are round-robin scheduled.

The timeout structure captures the amount of time that is allowed to block and the number of ticks for which it has blocked. Each subsequent call that is passed the same timeout structure may increase the amount of slept time and decrease the remaining time.

A thread may block for more than the permitted limit if it is sleeping while a higher-priority thread runs. Only the highest-priority thread can make strong realtime guarantees in the presence of other runnable threads.

Functions that take a timeout should always expect it as the first argument. This allows it to be forwarded to subsequent calls trivially.

Timeouts may not be stored on the heap. Any function checking timeouts may refuse to accept a heap-allocated timeout. It is difficult to work with heap-allocated timeouts because they may be deallocated while the thread is sleeping, which would then cause it to crash on updating the timeout structure.

5.4. Sleeping

Sleeping for a bounded number of ticks is the simplest form of blocking available. The thread_sleep call causes the caller to yield until a certain number of ticks have run.

```
int thread_sleep(struct Timeout * timeout)
```

Sleep for at most the specified timeout (see `timeout.h`).

The thread becomes runnable once the timeout has expired but a higher-priority thread may prevent it from actually being scheduled. The return value is a saturating count of the number of ticks that have elapsed.

A call of `thread_sleep` is with a timeout of zero is equivalent to `yield`, but reports the time spent sleeping. This requires a cross-domain call and return in addition to the overheads of `yield` and so `yield` should be preferred in contexts where the elapsed time is not required.

As with other calls that take a `Timeout`, the number of ticks that have elapsed during the call can be checked by reading the `elapsed` field of the timeout structure.

5.5. Building locks with futexes

The scheduler exposes a set of futex APIs as a building block for various notification and locking mechanisms. Futex is a contraction of 'fast userspace mutex'. This does not quite apply on a CHERIoT system, where there is no userspace, but the core concept of avoiding a privilege transition on fast paths still applies.

A CHERIoT RTOS futex is a 32-bit word where the scheduler provides compare-and-sleep (futex_timed_wait) and notify (futex_wake) operations.

```
int futex_timed_wait(Timeout * ticks,
                     const uint32_t * address,
                     uint32_t expected,
                     enum FutexWaitFlags flags)
```

Compare the value at `address` to `expected` and, if they match, sleep the thread until a wake event is sent with `futex_wake` or until this the thread has slept for `ticks` ticks.

The value of `ticks` specifies the permitted timeout. See `timeout.h` for details.

The `address` argument must permit loading four bytes of data after the address.

The `flags` argument contains flags that may control the behaviour of the call.

This returns:

- 0 on success: either `*address` and `expected` differ or a wake is received.
- `-EINVAL` if the arguments are invalid.
- `-ETIMEOUT` if the timeout expires.

```
int futex_wake(uint32_t * address,
               uint32_t count)
```

Wakes up to `count` threads that are sleeping with `futex[_timed]_wait` on `address`.

The `address` argument must permit storing four bytes of data after the address. This call does not store to the address but requiring store permission prevents a thread from waking up a futex that it cannot possibly have moved to a different state.

The return value for a successful call is the number of threads that were woken. `-EINVAL` is returned for invalid arguments.

In C++, `std::atomic<uint32_t>` provides `wait`, `notify_all`, and `notify_one` methods that expose futex functionality and may be more convenient to call than the raw

> futex APIs. These include some additional (non-standard) overloads that expose more of the underlying futex functionality.

A futex allows you to use atomic operations on a 32-bit word for fast paths but then sleep and wake threads when they are blocked, rather than spinning. Anything that can be implemented with a spin-wait loop can usually be made more efficient with a futex.

For example, consider the simplest possible spinlock, which uses a single word containing a one to indicate locked and a zero to indicate unlocked. When you encounter a one, you sit in a loop doing an atomic compare-and-swap trying to replace a zero with a one. When this succeeds, you've acquired the lock.

On most operating systems with single-core systems, you will sit in this loop until you exhaust your quantum, then a timer will fire and another thread will run. Your thread may be scheduled before the thread that owns the lock finishes, so you'll then spin for another quantum.

The first simple optimisation on this design is to yield in the spin loop. This will allow other threads to run but the waiting thread remains runnable and so may be rescheduled early. With an RTOS priority scheduler, if the thread that's waiting is a higher priority than thread that owns the lock then the thread that owns the lock may never be scheduled.

A futex lets the waiting thread sleep. The `futex_timed_wait` call will compare the value in the futex word to the expected value (one, indicating locked, in this case) and, if they match, will send the thread to sleep and remain asleep until the thread owning the lock will then do a `futex_wake` call when unlocking.

A more complete futex-based lock uses three values in the lock word to differentiate between locked states with and without waiters. This allows the uncontended case to avoid any cross-compartment calls.

The `locks` library provides a set of futex-based locks. The `locks.h` header exposes the interface to this library.

Ticket locks

provide guaranteed FIFO semantics for waiters.

Flag locks

are simple locks that wake waiters in the order of their thread priorities. These can optionally provide priority inheritance (see Section 5.6).

Recursive mutexes

wrap a priority-inheriting flag lock and allow the same thread to acquire a lock multiple times.

Semaphores

provide a counting semaphore abstraction.

C++ users may prefer to use the wrappers provided in `locks.hh`, which implement a uniform interface for different lock types. This header also defines a `NoLock` class that provides the same interface but does not do any locking so generic data structures can be implemented with and

without locking.

Futexes can be used to build other waiting mechanisms beyond locks. For example, a ring buffer with producer and consumer counters can have the sender wait while the ring is full by using a futex wait on the consumer counter and the receiver can do likewise with the producer counter. This allows a ring buffer design that is mostly lockless, yet allows the producer to sleep when the ring is full or the consumer to sleep when it is empty.

5.6. Inheriting priorities

Simple futex-based locks are vulnerable to *priority inversion*. Consider a case with three threads. The first is a low-priority thread that acquires a lock. The second is a medium-priority thread that preempts the first. The third is a high-priority thread that waits for the lock.

The high-priority thread in this example cannot make progress until the low-priority thread releases the lock. The low-priority thread cannot make progress until the medium-priority thread blocks. This means that the medium-priority thread is preventing the high-priority thread from making progress, which is the opposite of the desired situation.

Priority inheritance is the solution to this kind of problem. The blocking high-priority thread loans its priority to the low-priority thread, allowing it to (temporarily) be scheduled in preference to the medium-priority thread.

The futex APIs implement this by storing the thread ID of the owning thread in the bottom 16 bits of the futex word and passing `FutexPriorityInheritance` to the `flags` argument in the wait call. The specified thread will have its priority set to the highest priority of any of the waiting threads. The priority boost lasts until the waiters time out or the boosted thread releases the lock, whichever happens first.

A single thread can hold multiple priority-inheriting locks and receive priority boosts from all of them.

The priority inheritance mechanism can also be used to build asymmetric locks. These have a fast path that doesn't do any cross-compartment calls and a slow path that does. You can find one example of this in the hazard pointer mechanism for short-lived claims. This must detect when a thread has tried to add a hazard pointer while the allocator is scanning the list, without slowing down the allocator. Before reading the list, the allocator increments the top 16 bits of the futex word and sets the low 16 to the thread ID performing the operation. Threads updating the hazard set check the futex word before and after updating the list. If the top 16 bits have changed, they know that the allocator has scanned the list and they must retry. If the top 16 bits contain an odd value, the allocator is currently scanning the list and they must wait. They can do a priority-inheriting wait with a one-tick timeout *even though the allocator will not ever call `futex_wake`*. They will yield for one tick, boosting the priority of the thread that's currently in the allocator, but then resume at the end of the tick.

5.7. Securing futexes

Most of the time you will want to use futexes (and the locks that wrap them) to synchronise

operations within a single compartment. Futex-based locks rely on the contents of the lock word to be valid. For example, if a flag lock is directly accessible by two mutually distrusting compartments, one can write an invalid value to the word and either prevent the other from waking waiters or cause it to spuriously believe that it has acquired the lock.

This is not normally a limitation because locks typically protect some data structure or other resource that should not be concurrently mutated by multiple threads. Providing mutable views of such a structure to multiple compartments is almost certainly a security vulnerability, even without attacks on the futex.

There is one situation where futexes are safe to share across compartment boundaries. If you have a component that others trust for availability, it can share read-only views of a futex to allow waiting for an out-of-band event. The scheduler does this for interrupts (see [drivers]), allowing threads to use the futex wait operation to block until an interrupt is ready.

5.8. Using event groups

The `event_group` library provides an event group API that is primarily intended for porting code written against FreeRTOS's event groups APIs. The `event.h` header exposes the interface to this library. These APIs do not have a clear trust model and so should be avoided in new code that is not ported from FreeRTOS. You can build more convenient interfaces atop futexes for most synchronisation operations. You may also simply use multiple futexes and the `multiwaiter` API (see Section 5.10) to wait for multiple events.

An event group is a set of up to 24 values that can be set or cleared independently. Waiters can wait for any or all of an arbitrary subset of these.

Event groups are created with the eventgroup_create function. This returns an opaque handle to the event group, which can be used for setting, clearing, or waiting on events.

```
int eventgroup_create(struct Timeout * timeout,
                      struct SObjStruct * heapCapability,
                      struct EventGroup ** outGroup)
```

Create a new event group, allocated using `heapCapability`.

The event group is returned via `outGroup`.

This returns zero on success. Otherwise it returns a negative error code. If the timeout expires then this returns `-ETIMEDOUT`, if memory cannot be allocated it returns `-ENOMEM`.

Note that, because this allocates memory, it requires an *allocation capability*. See Chapter 6 for more information about what this means.

You can then use eventgroup_set and eventgroup_clear to set and clear some or all of the event flags in this group. Both of these calls return the old values of the bits.

```
int eventgroup_set(Timeout * timeout,
                   struct EventGroup * group,
                   uint32_t * outBits,
                   uint32_t bitsToSet)
```

Set one or more bits in an event group.

The `bitsToSet` argument contains the bits to set. Any thread waiting with `eventgroup_wait` will be woken if the bits that it is waiting for are set.

This returns zero on success. If the timeout expires before this returns then it returns `-ETIMEDOUT`.

Independent of success or failure, `outBits` will be used to return the set of currently set bits in this event group.

```
int eventgroup_clear(Timeout * timeout,
                     struct EventGroup * group,
                     uint32_t * outBits,
                     uint32_t bitsToClear)
```

Clear one or more bits in an event group.

The `bitsToClear` argument contains the set of bits to clear. This does not wake any threads.

This returns zero on success. If the timeout expires before this returns then it returns `-ETIMEDOUT`.

Independent of success or failure, `outBits` will be used to return the set of currently set bits in this event group.

You can then subsequently wait for some of the events to be set with the eventgroup_wait function. This takes a set of events to wait for and can wait until either any or all of them are set.

```
int eventgroup_wait(Timeout * timeout,
                    struct EventGroup * group,
                    uint32_t * outBits,
                    uint32_t bitsWanted,
                    _Bool waitForAll,
                    _Bool clearOnExit)
```

Wait for events in an event group.

The `bitsWanted` argument must contain at least one bit set in the low 24 bits (and none in the high bits). This indicates the specific events to wait for. If `waitForAll` is true then all of the bits in `bitsWanted` must be set in the event group before this returns. If `waitForAll` is false then any of the bits in `bitsWanted` being set in the event group will cause this to return.

If this returns zero then `outBits` will contain the bits that were set at the time that the condition became true. If this returns `-ETIMEDOUT` then `outBits` will contain the bits that were set at the time that the timeout expired.

Note: `waitForAll` requires all bits to be set *at the same time*. This makes it trivial to introduce race conditions if used with multiple waiters and `clearOnExit`, or if different threads clear bits different bits in the waited set.

If `clearOnExit` is true and this returns successfully then the bits in `bitsWanted` will be cleared in the event group before this returns.

This call can also atomically clear the bits that you've waited on, giving them edge-triggered behaviour.

5.9. Sending messages

A message queue is a FIFO capable of storing a fixed number of fixed-sized entries. There are two distinct use cases for message queues:

- Communicating between two threads in the same compartment.
- Communicating between different compartments.

In the first case, the endpoints are in the same trust domain. The `message_queue_library` library provides a simple message queue API that is intended for this use case. When the endpoints are in different trust domains, the endpoints must be protected from tampering. The `message_queue` compartment wraps the library in a compartment that exposes an almost identical interface to the library but with the endpoints exposed as (tamper-proof) sealed capabilities.

Queues for use within a single compartment are created with queue_create, which allocates the buffer and returns a handle that can be used for sending and receiving messages. There is no explicit `queue_destroy` function. The memory allocated can simply be freed when the queue is no longer needed. The pointer returned via the `outAllocation` parameter refers to the entire allocation used for the queue and so can be passed to `heap_free`, along with the heap capability used to

allocate the queue.

```
int queue_create(Timeout * timeout,
                 struct SObjStruct * heapCapability,
                 struct QueueHandle * outQueue,
                 void ** outAllocation,
                 size_t elementSize,
                 size_t elementCount)
```

Allocates space for a queue using `heapCapability` and stores a handle to it via `outQueue`.

The underlying allocation (which is necessary to free the queue) is returned via `outAllocation`.

The queue is has space for `elementCount` entries. Each entry is a fixed size, `elementSize` bytes.

Messages are then sent with queue_send and received with queue_receive. These are blocking (if allowed to by with a non-zero timeout) calls that send or receive a single message.

```
int queue_send(Timeout * timeout,
               struct QueueHandle * handle,
               const void * src)
```

Send a message to the queue specified by `handle`.

This expects to be able to copy the number of bytes specified by `elementSize` when the queue was created from `src`.

Returns 0 on success. On failure, returns `-ETIMEOUT` if the timeout was exhausted, `-EINVAL` on invalid arguments.

This expected to be called with a valid queue handle. It does not validate that this is correct. It uses `safe_memcpy` and so will check the buffer.

```
int queue_receive(Timeout * timeout,
                  struct QueueHandle * handle,
                  void * dst)
```

Receive a message over a queue specified by `handle`.

This expects to be able to copy the number of bytes specified by `elementSize`. The message is copied to `dst`, which must have sufficient permissions and space to hold the message.

Returns 0 on success, `-ETIMEOUT` if the timeout was exhausted, `-EINVAL` on invalid arguments.

For defence in depth, you can use queue_make_receive_handle or queue_make_send_handle to create a handle that can only be used for receiving or sending messages, respectively.

```
struct QueueHandle queue_make_receive_handle(struct QueueHandle handle)
```

Convert a queue handle returned from `queue_create` into one that can be used *only* for receiving.

Note: This is primarily defence in depth. A malicious holder of this queue handle can still set the consumer counter to invalid values.

```
struct QueueHandle queue_make_send_handle(struct QueueHandle handle)
```

Convert a queue handle returned from `queue_create` into one that can be used *only* for sending.

Note: This is primarily defence in depth. A malicious holder of this queue handle can still set the producer counter to invalid values and overwrite arbitrary queue locations.

The library interfaces to queues are not intended to be robust in the presence of malicious callers. They run in the same security context as the caller and so a caller may abuse them to corrupt its own state. They do aim to be robust with respect to the source or destination buffer for sending and receiving messages being invalid or concurrently deallocated.

You can probe the number of messages in a queue with queue_items_remaining.

```
int queue_items_remaining(struct QueueHandle * handle,
                          size_t * items)
```

Returns the number of items in the queue specified by `handle` via `items`.

Returns 0 on success. This has no failure mechanisms, but is intended to have the same interface as the version that operates on a sealed queue handle.

Note: This interface is inherently racy. The number of items in the queue may change in between the return of this function and the caller acting on the result.

If you are passing messages between compartments, you should use the versions of these functions with the `_sealed` suffix. The queue_create_sealed function creates a queue and returns separate send and receive handles, which can be passed to separate compartments. This queue can be destroyed by calling queue_destroy_sealed with the send and receive handles. The queue is not destroyed until both handles have been passed to this function.

```
int queue_create_sealed(Timeout * timeout,
                        struct SObjStruct * heapCapability,
                        struct SObjStruct ** outQueueSend,
                        struct SObjStruct ** outQueReceive,
                        size_t elementSize,
                        size_t elementCount)
```

Allocate a new message queue that is managed by the message queue compartment.

This is returned as two sealed pointers to send and receive ends of the queue.

```
int queue_destroy_sealed(Timeout * timeout,
                         struct SObjStruct * heapCapability,
                         struct SObjStruct * queueHandle)
```

Destroy a queue using a sealed queue endpoint handle.

The queue is not actually freed until *both* endpoints are destroyed, which means that you can safely call this from the sending end without the receiving end losing access to messages held in the queue.

The corresponding send and receive functions are identical to their library counterparts, but take one of the queue handles returned from queue_destroy_sealed.

5.10. Waiting for multiple events

The multiwaiter API allows waiting for any of a set of independent events. It is conceptually similar to `select`, `poll`, `epoll`, and `kqueue` in *NIX operating systems or `WaitForMultipleObjects` in Windows. It is designed to bound the amount of time that the scheduler must spend checking multiwaiters and to minimise the amount of memory that multiwaiters consume. Memory is allocated only when a multiwaiter is created, with multiwaiter_create. This creates a multiwaiter with space for a fixed number of events.

```
int multiwaiter_create(Timeout * timeout,
                       struct SObjStruct * heapCapability,
                       struct MultiWaiter ** ret,
                       size_t maxItems)
```

Create a multiwaiter object.

This is a stateful object that can wait on at most `maxItems` event sources.

Each multiwaiter_wait call is a one-shot operation. The call is passed a set of things to wait for and the associated condition via the `events` array and returns the waited status via the same array. This is typically an on-stack array.

```
int multiwaiter_wait(Timeout * timeout,
                     struct MultiWaiter * waiter,
                     struct EventWaiterSource * events,
                     size_t newEventsCount)
```

Wait for events.

The first argument is the multiwaiter to wait on. New events can optionally be added by providing an array of `newEventsCount` elements as the `newEvents` argument.

Return values:

- On success, this function returns 0.
- If the arguments are invalid, this function returns -EINVAL.
- If the timeout is reached without any events being triggered then this returns -ETIMEOUT.

The multiwaiter can natively wait only for futex notifications but higher-level mechanisms are built out of futexes. For example, if you wish to wait for a message queue (see Section 5.9) to be ready to send, you can call multiwaiter_queue_receive_init to initialise a multiwaiter event with the queue's receive counter and expected value. This event will then fire if the queue becomes non-full. The normal caveats about race conditions apply: the queue may become full again if another thread sends messages in between your receiving the notification and sending a message.

Chapter 6. Memory management in CHERIoT RTOS

It is common for embedded systems to avoid heap allocation entirely and preallocate all memory that they will need. This means that the total amount of memory that a system requires is the sum of the peak memory usage of all components.

The CHERIoT platform is designed to enable safe reuse of memory. The shared heap allows memory to be dynamically allocated for individual uses and then reused. This means that the total memory requirement for a system becomes the peak combined usage of all components. If two components use a lot of memory at different times, they can safely share the same memory.

6.1. Understanding allocation capabilities

The memory allocator uses a capability model. Every caller of a memory allocation or deallocation function must present a capability that authorises allocation. This is a *sealed* capability to an `AllocatorCapabilityState` structure. Sealed capabilities were introduced in Section 1.4.

This uses the static sealing mechanism described in Section 4.11. There is no limit to the number of allocator capabilities that a compartment can hold. Each allocation capability holds an independent quota.

There is no requirement that the sum of all allocation quotas is less than the total available heap space. You can over-commit memory if you know that it will not all be needed at the same time. The quota mechanism gives you a way of limiting the total memory consumption of individual compartments (or groups of compartments) and of cleaning up after failure.

6.2. Creating custom allocation capabilities

A compartment may hold different allocation capabilities for different purposes. The heap_free_all function allows you to free all memory allocated with a specified capability and so using multiple allocation quotas can be useful for error recovery.

You can forward-declare an allocator capability with the `DECLARE_ALLOCATOR_CAPABILITY` macro. This takes a single argument: the name of the allocator capability. You can then define the allocator capability with the `DEFINE_ALLOCATOR_CAPABILITY` macro, which takes the name and the quota size as arguments. These can be combined with the `DECLARE_AND_DEFINE_ALLOCATOR_CAPABILITY` macro.

The allocator capabilities are exposed as COMDATs in C++. This allows them to be defined in a header and used in multiple translation units. C does not expose a similar mechanism and so you must use the separate declare and define macros in C if your compartment has more than one compilation unit that wish to share an allocator capability and define the capability in a single compilation unit.

In future versions of CHERIoT RTOS, allocator capabilities are likely to gain additional restrictions (for example, separating the ability to allocate from the ability to claim).

6.3. Recalling the memory safety guarantees

Every pointer to a new allocation provided by memory allocator is derived from a capability to a large heap region and bounded. The *monotonicity* guarantees in a CHERI system ensure that a caller cannot expand the bounds of the returned pointer.

The CHERIoT platform provides two additional features for temporal safety. These both depend on a revocation bitmap, a shadow memory space that stores one bit per eight bytes of heap memory. When an object is freed, the allocator paints the bits associated with it.

The *load filter* then ensures that any pointer to the object will have its tag cleared when it is loaded. This gives deterministic use-after-free protection; any attempt to use a pointer to a deallocated object will trap. The object is then placed in *quarantine*.

The *revoker* periodically scans all memory and invalidates any pointers whose base address points to a deallocated object. The monotonicity of bounds ensures that the base of a capability always points either somewhere within the allocation or, if the length is zero, to the word immediately after it.

The allocator marks the metadata between allocations as freed. This means that a zero-length capability to the end of an object is likely to be untagged.

The load filter ensures that no new pointers to deallocated objects can appear in memory and so the revocation sweep can proceed asynchronously. Any object that is in quarantine at the start of a sweep is safe to remove from quarantine at the end.

This combination of features allows the allocator to provide complete spatial and temporal safety for heap objects.

6.4. Allocating with an explicit capability

```
void * heap_allocate(Timeout * timeout,
                     struct SObjStruct * heapCapability,
                     size_t size)
```

Non-standard allocation API.

Allocates `size` bytes. Blocking behaviour is controlled by the `timeout` parameter.

The non-blocking mode will return a successful allocation if one can be created immediately, or `nullptr` otherwise. The blocking versions of this may return `nullptr` if the timeout has expired or if the allocation cannot be satisfied under any circumstances (for example if `size` is larger than the total heap size).

Memory returned from this interface is guaranteed to be zeroed.

```
int heap_free(struct SObjStruct * heapCapability,
              void * ptr)
```

Free a heap allocation.

Returns 0 on success, or `-EINVAL` if `ptr` is not a valid pointer to the start of a live heap allocation.

The heap_allocate and heap_free functions take a capability, as described above, that authorises allocation and deallocation. When an object is allocated with an explicit capability, it may be freed only by presenting the same capability. This means that, if you pass a heap-allocated buffer to another compartment, that compartment cannot free it unless you also pass the authorising capability.

The allocation uses a timeout because the allocation API is able to block if insufficient memory is available. In contrast the deallocation API will always make progress. The allocator uses a priority-inheriting lock, which is dropped while blocking. If a high-priority thread frees memory while a lower-priority thread owns the lock then the lower-priority thread will wake up, complete its allocation or deallocation, release the lock, and allow the higher-priority thread to resume.

If you need to clean up all memory allocated by a particular capability, heap_free_all will walk the heap and deallocate everything owned by that capability. This is useful when a compartment has crashed, to reclaim all of its heap memory.

```
ssize_t heap_free_all(struct SObjStruct * heapCapability)
```

Free all allocations owned by this capability.

Returns the number of bytes freed or `-EPERM` if this is not a valid heap capability.

6.5. Using C/C++ default allocators

If you are porting existing C/C++ code then it is likely that it uses `malloc` / `free` or the C++ `new` / `delete` operators. These are implemented as wrappers around heap_allocate and heap_free that pass `MALLOC_CAPABILITY` as the authorising capability. You can also pass this capability explicitly to allocate things from the same quota as the standard allocation routines.

`MALLOC_CAPABILITY` is a macro referring to the default allocation capability *in the current compartment*. It refers to a different capability in every compartment.

You can control the amount of memory provided by this capability by defining the `MALLOC_QUOTA` for your compartment. If a compartment is not supposed to allocate memory on its own behalf, you

can define `CHERIOT_NO_AMBIENT_MALLOC`. This will disable C's `malloc` and `free` and C++'s global `new` and `delete` operators. Defining `CHERIOT_NO_NEW_DELETE` will disable the global C++ operator `new` and `delete`, but leave `malloc` and `free` available.

Defining these does not prevent memory allocation, you can still define non-default allocator capabilities and use them directly, but it prevents accidental allocation.

6.6. Defining custom allocation capabilities for `malloc` and `free`

If you simply wish to change the quota that is available to `malloc` and `free` then you can define `MALLOC_QUOTA` when compiling your compartment. If you require more control, such as controlling the compilation unit that contains the definition of the allocator capability, then you can define `CHERIOT_CUSTOM_DEFAULT_MALLOC_CAPABILITY`. This macro will cause `stdlib.h` to provide a forward declaration of the default allocator capability, but not to define it. You must define it as described in Section 6.2.

This is most useful for C compartments with multiple compilation units. These will need to define the malloc capability in a single compilation unit.

This limitation will be removed in a future toolchain iteration.

6.7. Allocating on behalf of a caller

Sometimes a compartment needs to be able to allocate memory but that memory is not logically owned by the compartment. This pattern appears even in the core of the RTOS. The compartment that provides message queues, for example, allocates memory on behalf of a caller, it does not hold the right to allocate memory on its own behalf. It does this by taking an allocator capability as an argument and forwarding it to the allocator.

Often, if a compartment is allocating on behalf of a caller, it needs to ensure that the caller doesn't tamper with the object. The token APIs provide a lightweight mechanism for doing this.

```
SObj token_sealed_unsealed_alloc(Timeout * timeout,
                                 struct SObjStruct * heapCapability,
                                 SKey key,
                                 size_t sz,
                                 void ** unsealed)
```

Allocate a new object with size `sz`.

An unsealed pointer to the newly allocated object is returned in `*unsealed`, the sealed pointer is returned as the return value.

The `key` parameter must have both the permit-seal and permit-unseal permissions.

On error, this returns `INVALID_SOBJ`.

```
void * token_obj_unseal(SKey,
                        SObj)
```

Unseal the obj given the key.

The key must have the permit-unseal permission.

```
int token_obj_destroy(struct SObjStruct * heapCapability,
                      SKey,
                      SObj)
```

Destroy the obj given its key, freeing memory.

The key must have the permit-unseal permission.

When you call token_sealed_unsealed_alloc, you must provide two capabilities:

- An allocator capability.
- A permit-seal sealing capability.

The first of these authorises memory allocation, the second authorises sealing. The CHERIoT ISA includes only three bits of object type space in the capability encoding and so the allocator provides a virtualised sealing mechanism. This allocates an object with a small header containing the sealing type and returns a sealed capability to the entire allocation and an unsealed capability to all except the header.

The unsealed capability can be used just like any other pointer to heap memory. The sealed

capability can be used with token_obj_unseal to retrieve a copy of the unsealed capability. The `token_obj_unseal` function requires a permit-unseal capability whose value matches the permit-seal capability passed to `token_sealed_unsealed_alloc`.

The virtualised sealing mechanism must be able to derive an accurate capability for the object excluding the header. This means that the size is currently restricted to a little under 4 KiB. Attempting to allocate larger sealed objects will fail. If you need larger sealed objects, allocate them as unsealed objects and store a pointer to them in a sealed object.

An object allocated in this way can be deallocated only by presenting *both* the allocator capability and the sealing capability that match the original allocation. This is very convenient for compartments that expose services because the memory cannot go away while they are using it and can be reclaimed only when the same caller (or something acting on its behalf) authorises the deallocation.

6.8. Ensuring that heap objects are not deallocated

If malicious caller passes a compartment a buffer and then frees it, then the callee can be induced to trap. There are some situations where this is acceptable. In some cases, compartments exist in a hierarchical trust relationship and it's fine for a more-trusted compartment to be able to crash a less-trusted one. In other cases, the compartment is fault tolerant. For example, the scheduler ensures that its data structures are in a consistent state before performing any operations on user-provided data that may trap. As such, it can unwind to the caller and, at worst, leak memory owned by the caller.

In situations involving mutual distrust, the callee needs to *claim* the memory to prevent its deallocation. The heap_claim function allows you to place a claim on an object. The claim is dropped by calling heap_free.

While you have a claim on an object, that object counts towards your quota. You can claim the same object multiple times, each time adds a new claim to the object but (if it is already claimed with that quota) does not consume quota.

You can pass a capability with bounds that do not cover an entire object to `heap_claim` but your claim will cover the entire object because you cannot

Chapter 7. Features for debug builds

CHERIoT provides a small set of APIs for use in debug builds in `debug.hh`. These include:

- Rich log messages
- Assertions with error messages
- Invariants that are checked in release builds but provide debugging help only in release builds

All of the message-producing aspects of these APIs use direct access to the UART. This can cause the messages to be interleaved but ensures that they are generated even if part of the system has crashed or deadlocked.

Access to the UART will show up in the linker report. You should ensure that your auditing checks ensure that you have not left debug access to the UART enabled in release builds.

7.1. Enabling per-component debugging

Debug builds can often be significantly larger than release builds. They contain more code and potentially large strings for debug messages. CHERIoT RTOS is designed to allow debugging features to be turned on on a per-compartment basis to help mitigate this. You can see this in the core components. If you run `xmake config --help` in a firmware build, you will see this at the end of the output:

```
--debug-token_library=[y|n] Enable verbose output and assertions in the token_library
--debug-allocator=[y|n]     Enable verbose output and assertions in the allocator
--debug-loader=[y|n]        Enable verbose output and assertions in the loader
--debug-scheduler=[y|n]     Enable verbose output and assertions in the scheduler
```

Each of the core components allows extra debugging modes to be enabled independently, rather than via a global debug-mode switch. Adding something similar requires two changes in your `xmake.lua` file. The first line, at top-level scope, declares the option:

```
debugOption("myComponent")
```

With this, you will get a message in `xmake config --help` like the one above, but it won't actually do anything. You must also opt your compartment or library into debugging support by adding the corresponding rule in the description of your compartment or library:

```
compartment("myComponent")
    add_rules("cheriot.component-debug")
```

By default, this assumes that the `debugOption` that you've provided has the same name as the target. Sometimes, it's useful to have a single debug option that enables or disables debugging for multiple components. You can set the `cheriot.debug-name` target property in your component to the name

that you expect, with a line like this:

```
compartment("myComponent")
    add_rules("cheriot.component-debug")
    on_load(function (target)
        target:set('cheriot.debug-name', "nameOfDebugOption")
    end)
```

Now, your compartment will be compiled with a macro that starts with `DEBUG_` and ends with the name of the debug option in all capitals. In the first example above, this would be `DEBUG_MYCOMPONENT`.

This can then be used with the `ConditionalDebug` class from `debug.hh`. This is typically used as follows:

```
using Debug = ConditionalDebug<DEBUG_MYCOMPONENT, "My component">;
```

The first template parameter is a boolean value that indicates whether this component is being debugged. The second is a free-form string literal that will be prepended (in magenta) to any debug line.

The rest of this chapter will assume that the `Debug` type has been defined in this way.

7.2. Generating log messages

Printing log messages is the simplest use of the debug APIs. The `Debug::log()` function takes a format string and then a set of arguments. This is similar to `printf` or `std::format`, inserting the arguments into the output, replacing placeholders. The syntax here is modelled on `std::format`, but does not currently accept any format modifiers. The `{}` syntax for placeholders makes it possible to add modifiers in the future. This class is designed to avoid needing heap allocator or large amounts of stack space and so is intentionally less flexible than a general-purpose formatting library.

Unsigned integers are printed as hex. Signed integers are printed as decimal. Floating point numbers are not supported. Individual characters are printed as characters, strings (either `const char*` or `std::string_view`) are printed as strings.

Enumerated types are converted to strings using the Magic Enum library and printed with their numeric value in brackets. This has some limitations (in particular, by default, it does not work with very large enumeration values). It also requires capability relocations because it generates tables of strings. If you compile a compartment with `CHERIOT_AVOID_CAPRELOCS` defined then enumerations will be printed as numeric values.

Two other types have rich formatted output. `PermissionSet` objects (see Section 3.9) are printed using the characters from the tables in Section 1.2. Capabilities (either as raw pointers or instances of the `CHERI::Capability` class) are printed in full detail. Printing a capability will give a block that looks something like this:

```
0x2004cc8c (v:1 0x2004cc8c-0x2004cc90 l:0x4 o:0x0 p: G RWcgm- -- ---)
```

This starts with the address and then has the metadata in brackets. The metadata includes the tag (valid) bit, then the range, then the length, object type, and permissions.

7.3. Asserting invariants

Assertions and invariants use the same formatting infrastructure as lthe log message code. In debug mode (for this component), the following two are equivalent:

```
Debug::Invariant(theAnswer == 42, "The answer was {}, expected 42", theAnswer);
Debug::Assert(theAnswer == 42, "The answer was {}, expected 42", theAnswer);
```

They will both check whether then answer is 42 and, if not, print a message to the UART telling you what the real value was. They will then issue an invalid instruction. If your compartment does not have an error handler (see Section 4.9, then this will unwind to the compartment that called you. If it does, then you can handle this just like any other error.

In release builds, assertions are removed entirely. Invariants are still checked, but no longer log a message on failure, they just trigger an illegal instruction.

In some cases, you may find that the expression that calculates the assertion condition is expensive and the compiler does not successfully optimise it away in release builds. In this case, you can use the version that takes a lambda instead:

```
Debug::Assert([]() { return someExpensiveCheck(); }, "An expensive check failed");
```

The lambda is never executed in release builds and so the compiler will strip it away. You can also use this form if you have multiple steps (which may have side effects) leading up to the assertion condition.

Chapter 8. Writing a device driver

CHERIoT aims to be small and easy to customize. It does not have a generic device driver interface but it does have a number of tools that make it possible to write modular device drivers.

8.1. What is a device?

From the perspective of the CPU, a device is something that you communicate with via a memory-mapped I/O interface, which may (optionally) generate interrupts. There are several devices that the core parts of the RTOS interact with:

- The UART, which is used for writing debug output during development.
- The core-local interrupt controller, which is used for managing timer interrupts.
- The platform interrupt controller, which is used for managing external interrupts.
- The revoker, which scans memory for dangling capabilities (pointers) and invalidates them.

Most embedded systems on chip will include additional devices. These range from very simple interfaces, such as general-purpose I/O (GPIO) pins that are mapped to a bit in a register, up to entire wireless network interfaces with rich sets of functionality.

8.2. Specifying a device's locations

Devices are specified in the board description file. The two relevant parts are the `devices` node, which specifies the memory-mapped I/O devices and the `interrupts` section that describes how external interrupts should be configured. For example, our initial FPGA prototyping platform had sections like this describing its Ethernet device:

```
    "devices" : {
        "ethernet" : {
            "start" : 0x98000000,
            "length": 0x204
        },
        ...
    },
    "interrupts": [
        {
            "name": "Ethernet",
            "number": 16,
            "priority": 3
        }
    ],
```

The first part says that the ethernet device's MMIO space is 0x204 bytes long and starts at address 0x98000000. The second says that interrupt number 16 is used for the ethernet device.

8.3. Accessing the memory-mapped I/O region

The `MMIO_CAPABILITY` macro is used to get a pointer to memory-mapped I/O devices. This takes two arguments. The first is the C/C++ type of the pointer, the second is the name from the board configuration file. For example, to get a pointer to the memory-mapped I/O space for the ethernet device above, we might do something like:

```
struct EthernetMMIO
{
    // Control register layout here:
    ...
};

__always_inline volatile struct EthernetMMIO *ethernet_device()
{
    return MMIO_CAPABILITY(struct EthernetMMIO, ethernet);
}
```

This macro must be used in code, it cannot be used for static initialisation. The macro expands to a load from the compartment's import table. Assigning the result of it to a global is an antipattern: you will get smaller code using it directly. The helper shown here will be inlined and expand to a single load capability load.

Now that you have a pointer to a `volatile` object representing the device's MMIO region, you can access its control registers directly. Any device can be accessed from any compartment in this way, but that access will appear in the linker audit report.

Any compartment that accesses this device will have an entry in the audit report that looks like this:

```
        {
          "kind": "MMIO",
          "length": 516,
          "start": 2550136832
        },
```

There is no generic policy for device access because the right policy depends on device and the SoC. Consider a device has two GPIO pins, one connected to an LED used to indicate a fault in the device and the other to trigger the sprinkler system for the building. You would probably write a policy that allows most compartments to indicate a fault, but restricts access to the sprinkler control to a single compartment. From the perspective of both the SoC and the RTOS, the two devices are identical.

You can then audit whether a firmware image enforces whatever policy you want (for example, no compartment other than a device driver may access the device directly). Note that the linker

reports will always provide the addresses and lengths in decimal, because they are standard JSON. We support a small number of extensions to JSON in the files that we consume, to improve usability, but don't use these in files that we produce, to improve interoperability.

There is no requirement to expose a device as a single MMIO region. You may wish to define multiple regions, which can be as small as a single byte, so that you can privilege separate your device driver.

Some devices have a very large control structure. For example, the platform-local interrupt controller is many KiBs. We don't define a C structure that covers every single field for this and instead just use `uint32_t` as the type for `MMIO_CAPABILITY`, which lets us treat the space as an array of 32-bit control registers.

8.4. Handling interrupts

To be able to handle interrupts, you must have a software capability (see Section 4.11) that authorises access to the interrupt. For the ethernet device that we've been using as an example, you would typically request one with this macro invocation:

```
DECLARE_AND_DEFINE_INTERRUPT_CAPABILITY(ethernetInterruptCapability, Ethernet, true,
true);
```

If you wish to share this between multiple compilation units, you can use the separate `DECLARE_` and `DEFINE_` forms (see `interrupt.h`) but the combined form is normally most convenient. This macro takes four arguments:

1. The name that we're going to use to refer to this capability. The name `ethernetInterruptCapability` is arbitrary, you can use whatever makes sense to you.
2. The name of the interrupt, from the board description file (`Ethernet`, in this case).
3. Whether this capability authorises waiting for this interrupt (this will almost always be `true`).
4. Whether this capability authorises acknowledging the interrupt so that it can fire again. This will almost always be true in device drivers but should generally be true for only one compartment (for each interrupt), whereas multiple compartments may wish to observe interrupts for monitoring.

As with the MMIO capabilities, sealed objects appear in compartment reports. For example, the above macro expands to this in the final report:

```
        {
          "contents": "10000101",
          "kind": "SealedObject",
          "sealing_type": {
            "compartment": "sched",
            "key": "InterruptKey",
            "provided_by": "build/cheriot/cheriot/release/example-
firmware.scheduler.compartment",
```

```
            "symbol": "__export.sealing_type.sched.InterruptKey"
          }
```

The sealing type tells you that this is an interrupt capability (it's sealed with the `InterruptKey` type, provided by the scheduler). The contents lets you audit what this authorises. The first two bytes are a 16-bit (little-endian on all currently supported targets) integer containing the interrupt number, so 1000 means 16 (our Ethernet interrupt number). The next two bytes are boolean values reflecting the last two arguments to the macro, so this authorises both waiting and clearing the macro. Again, this can form part of your firmware auditing.

8.5. Waiting for an interrupt

Now that you're authorised to handle interrupts, you will need something that you can wait on. Most real-time operating systems allow you to register interrupt-service routines (ISRs) directly. CHERIoT RTOS does not allow this because ISRs run with access to the state of the interrupted thread. On Arm M-profile, some registers are banked but the others are visible, on RISC-V all registers of the interrupted thread are visible. This means that an ISR runs with access to the thread and compartment that are interrupted. Not only would this potentially break compartment isolation, it would be difficult to use safely because the ISR would inherit an (untrusted) stack from the interrupted thread and have access to the interrupted compartment's globals instead of its own.

Instead, CHERIoT RTOS maps interrupts onto events that threads can wait on. A single thread with the highest priority that blocks waiting on an interrupt will be run as soon as the switcher and scheduler finish handling the interrupt. The switcher will spill the interrupted thread's state, the scheduler will wake the sleeping thread and note that it is now the highest-priority runnable thread, and then the switcher will resume from that thread. This sequence takes around 1,000 cycles on Ibex, giving an interrupt latency of 50 µS at 20 MHz or 10 µS at 100 MHz.

A future version of the CHERIoT architecture is expected to include extensions to the interrupt controller to allow direct context switch to a high-priority thread.

Each interrupt is mapped to a futex word, which can be used with scheduler waiting primitives. Futexes are discussed in detail in Section 5.5 but, for the purpose of interrupt handling you can think of them as counters with a compare-and-wait operation. You can get the word associated with an interrupt by passing the authorising capability to the `interrupt_futex_get` function exported by the scheduler:

```
const uint32_t *ethernetFutex = ethernetFutex =
    interrupt_futex_get(STATIC_SEALED_VALUE(ethernetInterruptCapability));
```

The `ethernetFutex` pointer is now a read-only capability (attempting to store through it will trap) that contains a number that is incremented every time the ethernet interrupt fires. You can now query whether any interrupts have fired since you last checked by comparing it against a previous value and you can wait for an interrupt with `futex_wait`, for example:

```
do
```

```
{
    uint32_t last = *ethernetFutex;
    // Handle interrupt here
} while (futex_wait(ethernetFutex, last) == 0);
```

If you want to wait for multiple event sources, you can use the multiwaiter (see Section 5.10) API. This allows sleeping on multiple kinds of event source so you can, for example, have a single thread that blocks waiting for a message to send from another thread or a message to receive from the device.

8.6. Acknowledging interrupts

If you copy the last example into a real device driver then you might be surprised that the loop runs twice and then stops. It will run once on start, once when the first interrupt is delivered, and then never again. This is because external interrupts are not delivered on a particular channel unless the preceding one has been acknowledged. A more complete version of the loop above looks like this:

```
do
{
    uint32_t last = *ethernetFutex;
    // Handle interrupt here
    interrupt_complete(STATIC_SEALED_VALUE(ethernetInterruptCapability));
} while ((last != *ethernetFutex) || futex_wait(ethernetFutex, last) == 0);
```

This includes two changes. The first is the call to `interrupt_complete` once the interrupt has been handled. This tells the scheduler to mark the interrupt as completed in the interrupt handler. It is possible that the interrupt will then fire immediately, in which case there's no point trying to sleep. The second change checks whether the value of the futex word has changed - if it has, then we skip the `futex_wait` call and handle the next interrupt immediately.

8.7. Exposing device interfaces

CHERIoT device drivers often have two levels of abstraction. The lower level provides an abstraction across different devices that offer similar functionality. The higher level provides a security model atop this.

In most cases, the lower-level abstractions are provided as header-only libraries that can be included in whichever compartments need them. This allows drivers to be incorporated into another compartment that has full access to the device. For example, the scheduler is the only component that has direct access to the interrupt controller, the memory allocator is the only component that has full access to the revoker. In both cases, separating the driver into a compartment would not provide any security benefit because the component that uses the device is allowed to do anything that it wants to the device and does not need to be protected from the device.

If a device has multiple consumers then it may need a compartment to handle multiplexing. For

example, our debug APIs use the UART directly, but safe use of the UART would involve locking to avoid interleaved messages. Implementing this model would use the header-only UART driver from a compartment and writing a simple interface for reading and writing (possibly with an authorising capability).

8.8. Using layered platform includes

Each board description contains a set of include paths. For example, our Ibex simulator has this:

```
    "driver_includes" : [
        "../include/platform/ibex",
        "../include/platform/generic-riscv"
    ],
```

These are added **in this order**, which makes it possible for code in the more specialised directories to `#include_next` versions of the files in the more generic versions or to add files that are found in preference to the generic versions.

For example, the UART device in the `generic-riscv` directory defines a basic 16550 interface. This is templated with the size of the register because the original 16550 used 8-bit registers, whereas newer versions typically use the low 8 bits of a 32-bit register. This implementation is sufficient for simulated environments but real UARTs with higher-speed cores often require more control over their frequency to get the right baud rate. We can support the Synopsis extended 16550 by creating a `platform/synopsis` directory containing a `platform-uart.hh` that uses `#include_next <platform-uart.hh>` to get the generic version. This can be inserted in the include path before `platform/generic-riscv`. A specific configuration can use this by not providing anything at a higher level, replace it entirely by providing a custom `platform-uart.hh`, or provide a modified version of it by using `#include_next`.

8.9. Conditionally compiling driver code

The `DEVICE_EXISTS` macro can be used with `#if` to conditionally compile code depending on whether the current board provides a definition of the device. This is keyed on the existence of an MMIO region in the board description file with the specified name. For example, the ethernet device that we've been using as an example could be protected with:

```
#if DEVICE_EXISTS(ethernet)
// Driver for the ethernet device here.
#endif
```

Note: This highlights why "ethernet" is not a great name for the device: ideally the name should be specific to the hardware interface, not the high-level functionality, so that you can conditionally compile specific drivers. We have used a generic name in this tutorial to avoid introducing device-specific complications.

Chapter 9. Adding a new board

CHERIoT RTOS uses a JSON file to describe the target. At a first glance, this looks similar to a flattened device tree (FDT) source file. Both contain a layout of memory and locations of devices but the CHERIoT RTOS board description file also contains a lot of information that is useful only at build time, such as the locations of header files and preprocessor defines for the platform.

When you want to create a board support package (BSP) for a new CHERIoT configuration, this is the first place to start. The CHERIoT RTOS build system allows board description files to be specified either as names in the `sdk/boards` directory or as file paths. Anything that has been contributed to the CHERIoT RTOS repository will use the former mechanism, anything distributed separately will use the latter.

9.1. Memory layout

CHERIoT RTOS has to differentiate three kinds of memory in the configuration

- Code and read-only global data, which cannot contain pointers to revokable heap memory.
- Globals and stacks, which may contain pointers to revokable heap memory.
- The heap, which may contain pointers to revokable heap memory and may itself be revoked.

The allocator is given a capability to the revocation bitmap covering the last range. The revoker must be configured to include (at least) the latter two in its scans. The first category is safe to ignore.

A future version of CHERIoT RTOS will differentiate between code (and non-capability read-only data) and read-only data so that the former can be run from memory that does not support tags and the latter from tag-carrying memory.

The memory layout will put code then globals into memory and then the heap afterwards. In most systems, there is more code than heap and so, to reduce costs, not all memory needs to support tags.

Our security guarantees for the shared heap depend on the mechanism that allows the allocator to mark memory as quarantined. Any pointer to memory in this region is subject to a check (by the hardware) on load: if it points to deallocated memory then it will be invalidated on load. This mechanism is necessary only for memory that can be reused by different trust domains during a single boot. Memory used to hold globals and code does not require it and so an implementation may save some hardware and power costs by supporting these temporal safety features for only a subset of memory. As such, we require a range of memory that is used for static code and data ('instruction memory') that is not required to support this mechanism and an additional range that **must** support this for use as the shared heap ('heap memory'). Implementations may choose not to make this separation and provide a single memory region. At some point, we expect to further separate the mutable and immutable portions of instruction memory so that we can support execute in place.

Instruction memory is described by the `instruction_memory` property. This must be an object with a `start` and `end` property, each of which is an address.

The region available for the heap is described in the `heap` property. This must describe the region over which the load filter is defined. If its `start` property is omitted, then it is assumed to start in the same place as instruction memory.

The Sail board description has a simple layout:

```
    "instruction_memory": \{
        "start": 0x80000000,
        "end": 0x80040000
    },
    "heap": \{
        "end": 0x80040000
    },
```

This starts instruction memory at the default RISC-V memory address and has a single 256 KiB region that is used for both kinds of memory.

9.2. MMIO Devices

Each memory-mapped I/O device is listed as an object within the `devices` field. The name of the field is the name of the device and must be an object that contains a `start` and either a `length` or `end` property that, between them, describe the memory range for the device. Software can then use the `MMIO_CAPABILITY` macro with the name of the device to get a capability to that device's MMIO range and can use `#if DEVICE_EXISTS(device_name)` to conditionally compile code if that device exists.

The Sail model is very simple and so provides only three devices:

```
    "devices": \{
        "clint": \{
            "start": 0x2000000,
            "length": 0x10000
        },
        "uart": \{
            "start": 0x10000000,
            "end":   0x10000100
        },
        "shadow" : \{
            "start" : 0x83000000,
            "end"   : 0x83001000
        }
    },
```

This describes the core-local interrupt controller (`clint`), a UART, and the shadow memory used for the temporal safety mechanism (`shadow`). The UART, for example, is referred to in source using `MMIO_CAPABILITY(struct Uart, uart)`, which evaluates to a `volatile struct Uart *`, giving a capability to this device.

9.3. Interrupts

External interrupts should be defined in an array in the `interrupts` property. Each element has a `name`, a `number` and a `priority`. The name is used to refer to this in software and must be a valid C identifier. The number is the interrupt number. The priority is the priority with which this interrupt will be configured in the interrupt controller.

Interrupts may optionally have an `edge_triggered` property (if this is omitted, it is assumed to be false). If this exists and is set to true then the interrupt is assumed to fire when a condition first holds, rather than to remain raised as long as a condition holds. Interrupts that are edge triggered are automatically completed by the scheduler; they do not require a call to `interrupt_complete`.

9.4. Hardware features

Some properties define base parts of hardware support. The `revoker` property is either absent (no temporal safety support), `"software"` (revocation is implemented via a software sweep) or `"hardware"` (there is a hardware revoker). We expect this to be `"hardware"` on all real implementations, the software revoker exists primarily for the Sail model and the no temporal safety mode only for benchmarking the overhead of revocation.

If the `stack_high_water_mark` property is set to true, then we assume the CPU provides CSRs for tracking stack usage. This property is primarily present for benchmarking as all of our targets currently implement this feature.

9.5. Clock configuration

The clock rate is configured by two properties. The `timer_hz` field is the number of timer increments per second, typically the clock speed of the chip (the RISC-V timer is defined in terms of cycles). The `tickrate_hz` specifies how many scheduler ticks should happen per second. See the [timeout documentation](Timeout.md) for more discussion about ticks.

9.6. Conditional compilation

The `defines` property specifies any pre-defined macros that should be set when building for this board. The `driver_includes` property contains an array (in priority order) of include directories that should be added for this target. Each of the paths in `driver_includes` is, by default, relative to the location of the board file (which allows the board file and drivers to be distributed together). Optionally, it may include the string `$(sdk)`, which will be replaced by the full path of the SDK directory. For example, `"$(sdk)/include/platform/generic-riscv"` will expand to the generic RISC-V directory in the SDK.

The driver headers use `#include_next` to include more generic files and so it is important to list the directories containing your overrides first.

9.7. Simulation support

There are two properties for defining simulation platforms. If `simulation` is set to `true` then this

board is assumed to be a simulation platform. This will make the `simulation_exit` function attempt to exit the simulator in case of catastrophic failures.

In addition, the `simulator` property can be the name of a program (or script) that can simulate images compiled for this board. This will be run from the build directory and will be passed the absolute path of the firmware image when `xmake run` is used. The build system will look for the simulator in the SDK directory and, failing that, in the path. Exact paths can be provided by using `${sdk}` or `${board}` in the name of the simulator. These will be expanded to the full path of the SDK or the directory containing the board description file, respectively.

Chapter 10. Porting from bare metal

If you have existing code that runs happily bare metal, you may consider CHERIoT for a variety of reasons, for example:

- You want to add network connectivity and so need to isolate network communication.
- You are consolidating multiple functions from different microcontrollers onto a single device.
- You really love memory safety.

These reasons are often some variation on needing to do two more more things in different security contexts on a single device. This means that your workloads are going to now run with their privileges reduced enough that they cannot interfere (beyond permitted amounts) with each other.

10.1. Replacing a real-time control loop

Control systems often run with a single loop that polls or some input and manages a (potentially very complex) state machine and sets some output state. You can get precisely this model by running code in CHERIoT RTOS with interrupts disabled.

A function that has the `[[cheri::interrupt_state(disabled)]]` attribute will run with interrupts disabled and so has exclusive use of the core until it yields. You can add this attribute to the entry point for the thread running your control loop to start with interrupts disabled.

Not true. See issue 129, should be fixed soon!

The scheduler will always schedule the highest-priority runnable thread (or round-robin schedule threads if more than one is runnable at the same priority). If your thread is the highest priority, it won't be preempted, but interrupts may still fire and cause the scheduler to perform some book-keeping work. Disabling interrupts and running with the highest priority ensures that a thread is scheduled first and continues to run for as long as it wants to.

This is a direct replacement of a real-time control loop, but somewhat misses the point of running an RTOS: no other threads will run.

10.2. Yielding

If it makes sense for a control loop to run on a multitasking operating system then there will be times when it able to safely yield. Just yielding from a high-priority thread is not normally sufficient because it remains the highest-priority thread and so will be the next to run.

Chapter 5 discusses the various ways for a thread to block. This can be as simple as sleeping. If a realtime thread sleeps for one tick then another thread can run, but the next timer interrupt will return control to the realtime thread (unless another thread is running with interrupts disabled - this can be prevented via a policy on the linker report).

More commonly, a realtime control loop will want to block until some external event occurs and triggers an interrupt. Section 8.5 describes how to wait for an interrupt to fire.

When an interrupt fires, the thread waiting for it will become runnable and, if it is higher priority than any other thread, will be scheduled immediately. If the code that yielded had interrupts disabled then interrupts will be disabled once again on return.

10.3. Replacing direct device access

In bare-metal code for non-CHERI systems, it is common to construct pointers to memory-mapped devices by either casting an integer to a pointer or by creating a global that is placed in the correct location via a linker script.

Neither of these works in the CHERIoT model. Instead, you must use the macros described in Section 8.3 to construct valid capabilities to devices. This mechanism allows auditing, with a link-time record of which compartments can access each device.

If your code is using `volatile` pointers to access device memory then you should be able to port your code to CHERIoT RTOS by simply changing how you first construct those pointers.

10.4. Replacing interrupt service routines

Some bare-metal environments have special attributes for declaring interrupt-service routines and associating them with different channels. As discussed in Section 8.5, this kind of mechanism would violate the CHERIoT security model and so is not provided. You can implement your own dispatcher in a CHERIoT environment by waiting on multiple interrupts with the multiwaiter APIs (see Section 5.10) and then calling the interrupt routines yourself.

If interrupts are marked as edge-triggered in the board description then they are implicitly acknowledged in the interrupt controller by the scheduler. If not, then you must explicitly acknowledge them before they can fire again. This model is closer to the implicit masking during an ISR.

Simply waiting for multiple interrupts and handling them as they arrive does not allow interrupt handlers to be preempted. You can wait for different-priority interrupts on different-priority threads, but the threads that handle the lower-priority interrupts must run with interrupts enabled to allow preemption.

Chapter 11. Porting from FreeRTOS

FreeRTOS is an established real-time operating system with a large deployed base. It runs on tiny microcontrollers up to large systems with MMU-based isolation. The CHERIoT platform aims to provide, on small microcontrollers, stronger security guarantees than FreeRTOS is able to provide on large systems.

This chapter describes how several concepts in FreeRTOS map to equivalents in CHERIoT RTOS.

The `FreeRTOS-Compat` directory in `include` contains a set of headers (including `FreeRTOS.h`) that expose FreeRTOS-compatible wrappers around various CHERIoT RTOS services. These allow you to port existing FreeRTOS code to CHERIoT RTOS with minimal changes. These are not complete, but are expected to evolve over time.

11.1. Contrasting design philosophies

FreeRTOS is primarily designed around a model with a single trust domain. The initial targets did not provide any memory protection. You, the author of an embedded system, were assumed to have control over all components that you're integrating. Later, MPU support was added, building on top of the task model. When using an MPU, some tasks can be marked as unprivileged. These have access to their own stack and up to three memory regions, which must be configured explicitly.

Even when an MPU exists, the trust model is limited to hierarchical trust. The system integrator may mark certain tasks as unprivileged, but individual tasks cannot define more complex trust relationships. Memory safety is limited to the granularity of an MPU region. For example, the scheduler can expose message queues as privileged functions, which protects the queue's internal state from being tampered with by untrusted tasks, but may still overwrite the bounds of an object in an untrusted tasks if passed a pointer to an object that is not large enough to store a complete message.

As a fundamental design principle, FreeRTOS aims to run on many different platforms and provide portable abstractions. This limits the security abstractions that are possible to implement.

In contrast, the CHERIoT platform was created as a whole-system hardware-software co-design project. The hardware is required to provide properties that the software stack can use to build security policies. The core design of CHERIoT is motivated by a world in which a developer of an embedded system may not have full control over components provided by third parties, yet must integrate them. It is intended to provide auditing support that allows the integrator to make security claims even when integrating binary-only components.

This difference manifests most obviously in the fact that FreeRTOS provides imperative APIs for a number of things that CHERIoT RTOS prefers to create via declarative descriptions. Auditing a declarative description is easier than auditing arbitrary Turing-complete imperative code calling privileged APIs.

FreeRTOS starts from a position of sharing by default and has added MPU support to provide isolation. CHERIoT RTOS starts from a default position of isolation and provides object-granularity sharing.

The design of FreeRTOS was designed to support adding features to systems that did not originally use any kind of OS. This is apparent, for example, in how the programmer interacts with the scheduler. The scheduler is just another service that the system integrator may choose to use. User code chooses when the scheduler starts and may choose to stop it for arbitrary periods.

In contrast, CHERIoT RTOS provides a model mode familiar to users of desktop or server systems. The core parts of the RTOS are always available and provide strong isolation guarantees.

11.2. Replacing tasks with threads and compartments

The FreeRTOS task abstraction is similar to the traditional UNIX process abstraction. A task owns a thread and is independently scheduled. It is intended to be isolated from the rest of the system, though on systems without memory protection it has access to everything in the address space.

A task in FreeRTOS is roughly the equivalent of a combination of a thread and a compartment in CHERIoT RTOS. The compartment defines the code and global data associated with the task. The thread provides the stack and allows the task to be created.

CHERIoT RTOS threads have one key limitation in comparison to FreeRTOS tasks: They cannot be dynamically created. The security model requires a static guarantee that no memory moves between being stack memory (which is permitted to hold non-global capabilities) and non-stack (global or heap) memory. The trusted stack memory and save area memory should never be visible outside of the switcher. Without these static properties, the allocator would be in the TCB for thread and compartment isolation.

As such, there is no equivalent of the FreeRTOS `xTaskCreate` function. Threads (and their associated stacks and trusted stacks) must be described up front in the build system (see Section 5.1). In some cases, dynamically created threads can be replaced with thread pools, in the same way that coroutines can.

The compatibility later exposes `xTaskCreate` and `xTaskCreateStatic` as macros that generate a warning and evaluate to an invalid thread handle. This is intended to ease porting of code that conditionally uses these APIs.

The best way to replace dynamic thread creation is usually to create the threads declaratively in the build system. If they need to be started only after a certain event, then you can wait on a futex (see Section 5.5) and notify that futex at the point where the original code called `xTaskCreate`.

11.3. Using thread pools to replace coroutines

The CHERIoT RTOS thread pool (see `lib/thread_pool`) allows a small number of threads to be reused. This provides a compartment that has two entry points. One is a thread entry point that sits and waits for messages from other threads, the other is exposed for calls by other compartments and sends a message to one of the threads in the pool.

This is most commonly used with C++ lambdas via the `async` wrapper in `thread_pool.h`:

```
async([]() {
```

```
    // This runs in the caller's compartment but in another thread.
})
```

This can be used for cooperatively-scheduled work in a similar manner to stackless coroutines. Each task dispatched to a thread pool will run until completion on one of the threads allocated to the thread pool. When it returns, the thread-pool thread will block until another task is available in the queue.

Some of the use cases for dynamic FreeRTOS task creation can be implemented the same way. On memory-constrained systems, dynamic thread creation can easily exhaust memory for stacks and so most systems that depend on dynamic thread creation do so at different phases of computation to allow the stack space to be reused. Pushing these as thread-pool tasks provides similar behaviour, with each task taking ownership of the (safely zeroed) stack after the previous one has finished.

The RTOS-provided thread pool is very simple. You may wish to implement something similar using it as an example, rather than using it as an off-the-shelf component.

11.4. Porting code that uses message buffers

The CHERIoT RTOS message queue APIs (see Section 5.9) are modelled after the FreeRTOS message queue. In most cases, there is a direct mapping between the FreeRTOS APIs and the CHERIoT RTOS ones, as shown in Table 4

Table 4. CHERIoT equivalents of FreeRTOS queue operations

FreeRTOS API	CHERIoT RTOS API
xQueueCreate	queue_create
vQueueDelete	free
xQueueReceive	queue_receive
xQueueSendToBack	queue_send
uxQueueMessagesWaiting	queue_items_remaining

The `FreeRTOS-Compat/queue.h` header provides wrappers that respect this mapping. The CHERIoT RTOS APIs provide some additional functionality that is not present in FreeRTOS and so code that does not need to be maintained working in both environments may benefit from being moved to the native APIs.

This mapping uses the queue *library*, which is intended for communication between threads in the same compartment. FreeRTOS code typically assumes a single trust domain and so this is usually what you want when porting. In some cases, you will split multiple FreeRTOS components into separate compartments. In this case, you will most likely want to use the queue *compartment* (see Section 5.9), which isolates the queue state from callers.

For C++ code, the ring buffer in `ring_buffer.hh` may be more interesting. This provides a generic

ring buffer that can be specialised with different locks on the producer and consumer end.

11.5. Porting code that uses event groups

As with message queues, the CHERIoT RTOS event queue API was modelled on that of FreeRTOS. As such, there is direct correspondence between the FreeRTOS APIs and the equivalent CHERIoT RTOS versions, shown in Table 5.

Table 5. CHERIoT equivalents of FreeRTOS event group operations

FreeRTOS API	CHERIoT RTOS API
xEventGroupCreate	eventgroup_create
vEventGroupDelete	eventgroup_destroy
xEventGroupWaitBits	eventgroup_wait
xEventGroupClearBits	eventgroup_clear
xEventGroupSetBits	eventgroup_set

The `FreeRTOS-Compat/event_groups.h` header performs this translation.

The FreeRTOS event queue structure provides a rich set of operations. In contrast, CHERIoT RTOS aims to provide a small set of core abstractions that can be assembled into complex systems. A lot of users of the event groups API could use simpler wrappers around a futex, rather than an event group.

11.6. Adopting CHERIoT RTOS locks

CHERIoT RTOS provides futexes as the building block for most locks. This can be used to build counting semaphores, ticket locks, mutexes, priority-inheriting mutexes, and so on. Several of these are implemented in the locks library and exposed via `locks.h` (and `locks.hh` for C++ wrappers).

The `FreeRTOS-Compat/semphr.h` exposes FreeRTOS-compatible wrappers for counting semaphores. In FreeRTOS, these are implemented as message queues with zero-sized messages. In CHERIoT RTOS, they are simply futexes that store a count. This means semaphore get and put operations are usually simple atomic operations. The scheduler is not involved unless a thread needs to block (the semaphore count is zero and a thread tries to do a semaphore-get operation) or needs to wake waiters (the semaphore value is increased from zero and there were waiting threads).

Unlike FreeRTOS, CHERIoT RTOS exposes different types for different locking primitives if they are incompatible. This catches some API misuse errors at compile time. For example, FreeRTOS uses `SemaphoreHandle_t` to represent semaphores and recursive mutexes. These must be created with different functions and then locked and unlocked with different functions, but creating something as a semaphore and then trying to lock it as a recursive mutex will compile. In contrast, CHERIoT RTOS exposes these as distinct types and will fail to compile if you try to pass a semaphore to, for example, recursivemutex_trylock.

The `FreeRTOS-Compat/semphr.h` header provides wrappers that for the various types. These expose the FreeRTOS APIs and wrap all of the relevant CHERIoT RTOS types in a union with a

discriminator. This adds a small amount of overhead for dynamic dispatch and so code that uses only one type of semaphore can avoid this. Each of the underlying types can be exposed by defining one of the following macros before including `FreeRTOS-Compat/semphr.h` (directly, or indirectly via `FreeRTOS.h`):

`CHERIOT_FREERTOS_SEMAPHORE`

: Expose counting and binary semaphores.

`CHERIOT_FREERTOS_MUTEX`

: Expose non-recursive (priority-inheriting) mutexes.

`CHERIOT_FREERTOS_RECURSIVE_MUTEX`

: Expose recursive mutexes.

Enabling only the subset that you use (which can be done on a per-file basis) will reduce code size and improve performance.

11.7. Building software timers

FreeRTOS provides a timer callback API. This is implemented on top of existing functionality in the FreeRTOS kernel. CHERIoT RTOS does not yet provide such an API, but building one is fairly simple.

The structure of such a service is similar to that of the thread pool in `lib/thread_pool`, except that each callback has an associated timer. These should be added to a data structure that keeps them sorted. The thread that runs the callbacks should wait on a message queue, with the timeout set to the shortest time timer. If this wakes with timeout, it should invoke the first (`__cheri_callback`, see Section 3.3) callback function in its queue. If it wakes receiving a message, it should add the new callback into the set that it has ready.

There is no generic version of this in CHERIoT RTOS because it is impossible to implement securely in the general case for a system with mutual distrust. Callbacks may run for an unbounded amount of time (preventing others from firing) or untrusted code may allocate unbounded numbers of timers and exhaust memory. As such, it is generally better to build a bespoke mechanism for the specific requirements of a given workload.

11.8. Timing out blocking operations

FreeRTOS uses the combination of `vTaskSetTimeOutState` and `xTaskCheckForTimeOut` to implement timeouts. These are implemented in the FreeRTOS compatibility layer. In CHERIoT RTOS, these are subsumed in the `Timeout` structure, which contains both the elapsed and remaining number of ticks for a timeout.

The CHERIoT RTOS design is intended to be trivially composed. Most operations simply forward the timeout structure to a blocking operation in the scheduler (a sleep of a futex wait). They can query whether the timeout has expired without needing to query the scheduler, simply by checking whether the `remaining` field of the structure is zero.

11.9. Dynamically allocating memory

FreeRTOS provides a number of different heap implementations, not all of which are thread safe. In contrast, CHERIoT RTOS design assumes a safe, secure, shared heap. Various uses of statically pre-allocated memory in a FreeRTOS system can move to using the heap allocation mechanisms in CHERIoT RTOS, reducing total memory consumption.

FreeRTOS prior to 9.0 allocated kernel objects from a private heap. Later versions allow the user to provide memory. The latter approach has the benefit of accounting these objects to the caller, but the disadvantage of breaking encapsulation.

CHERIoT RTOS has an approach (described in Chapter 6) that combines the advantages of both. Rather than providing memory for creating objects such as message queues, multiwaiters, semaphores, and so on, the caller provides an *allocation capability*. This is a token that permits the callee to allocate memory on behalf of the callee. The scheduler is not able to allocate memory on its own behalf, it can allocate memory only when explicitly passed an allocation capability. It then uses the sealing mechanism to ensure that the caller cannot break encapsulation for scheduler-owned objects.

11.10. Disabling interrupts

FreeRTOS code often uses critical sections to disable interrupts. This may require some source-code modifications. Critical sections in FreeRTOS are used for two things:

- Atomicity
- Mutual exclusion

Disabling interrupts is the simplest way of guaranteeing both on a single-core system. FreeRTOS provides two APIs for critical sections: `taskENTER_CRITICAL` and `taskEXIT_CRITICAL`, which disable interrupts, and `vTaskSuspendAll` and `xTaskResumeAll`, which disable the scheduler. CHERIoT RTOS is designed to provide availability guarantees across mutually distrusting components and so does not permit either unbounded disabling of interrupts or turning the scheduler off. If mutual exclusion is the only requirement then you can implement these function as acquiring and releasing a lock that is private to your component. This is how that are implemented in the compatibility layer. They use distinct locks and these must be defined in your compartment, as shown below:

```
struct RecursiveMutexState __CriticalSectionFlagLock;
struct RecursiveMutexState __SuspendFlagLock;
```

A futex-based lock is very cheap to acquire in the uncontended case, it requires a single atomic compare-and-swap instruction (this may be a function call to a library routine that runs with interrupts disabled if the hardware does not support atomics). If possible, this approach is preferred for two reasons. First, it ensures that your component's critical sections do not impede progress of higher-priority threads. Second, it removes a burden on auditing.

The second use case, atomicity with respect to the rest of the system, requires disabling interrupts.

The CHERIoT platform requires a structured-programming model for disabling interrupts. Interrupt control can be done only at a function granularity. Hopefully, the code that runs with interrupts disabled is already a lexically scoped block. In C++, you can simply wrap this in a lambda and pass it to `CHERI::with_interrupts_disabled`. In C, you will need to factor it into a separate function.

For auditing, you may prefer to move the code that runs with interrupts disabled into a separate library. This lets you separately audit the precise code that is allowed to run with interrupts disabled, but modify the rest of your component without constraints.

11.11. Strengthening compartment boundaries for FreeRTOS components

Microsoft did an internal port of the FreeRTOS network stack and MQTT library. This was not part of the open-source release, but involved very little code change. Most of the porting effort was done via a FreeRTOS compatibility header, which provided wrappers around the CHERIoT RTOS inter-thread communication APIs to make them look like the FreeRTOS equivalents.

FreeRTOS assumes, by default, that all code and globals are shared unless explicitly protected by an MPU region. When porting FreeRTOS components, this assumption is broken unless they are in the same compartment. This is not normally a problem for an initial port, because components are cleanly encapsulated and do not directly modify the state of other components.

This property does not hold on all RTOS implementations. For example, several ThreadX components directly manipulate the internal state of the scheduler, rather than acting via well-defined APIs.

Using compartments give some defence in depth against accidental errors, but may not provide strong security guarantees. For example, the FreeRTOS TCP/IP stack provides a `FreeRTOS_socket` call that returns a pointer to a heap-allocated socket structure that encapsulates connection state. Simply compiling this in a CHERIoT compartment has a few limitations.

First, the structure is allocated out of the network stack's quota. This means that a caller can perform a denial of service by opening a load of connections. Fixing this requires an API change to pass an allocation capability (and possibly a timeout) into the network-stack compartment so that it can allocate this space on behalf of the caller.

Second, the structure is unprotected. The caller can load and store via the returned pointer and so can corrupt connection state. This may allow it to leak state of connections owned by other components or cause arbitrary failures.

Finally, there is no notion of access control. That might be fine: if you're allowing only one compartment to talk to the network stack then you don't need any kind of authorisation. For more complex uses, you may want to allow one component to talk to a command-and-control server and another component to talk to an update server. Neither of these components should be able to connect anywhere else and so you probably want to use the software capability model to define a static authorisation to make DNS lookups of a specific domain and then have that return a dynamic authorisation that allows connection to that host (or place both the lookup and connection behind a

single interface).

This is more work than is necessary to simply make FreeRTOS code work in a CHERIoT system, but is desirably if you want to take advantage of the security properties that CHERIoT RTOS provides over and above what is possible in FreeRTOS.