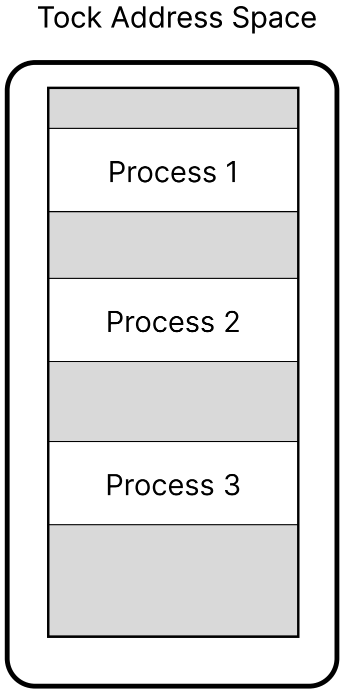
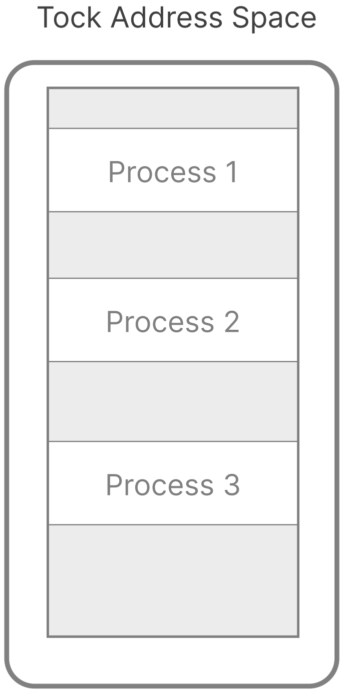
Verifying Memory Isolation in Tock

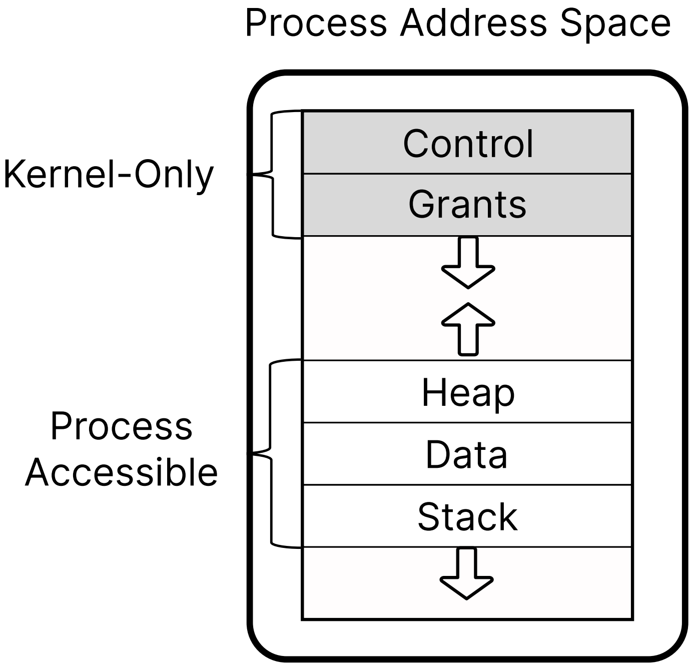
Vivien Rindisbacher, Evan Johnson, Nico Lehmann, Stefan Savage, Deian Stefan, Ranjit Jhala

Tock process memory layout

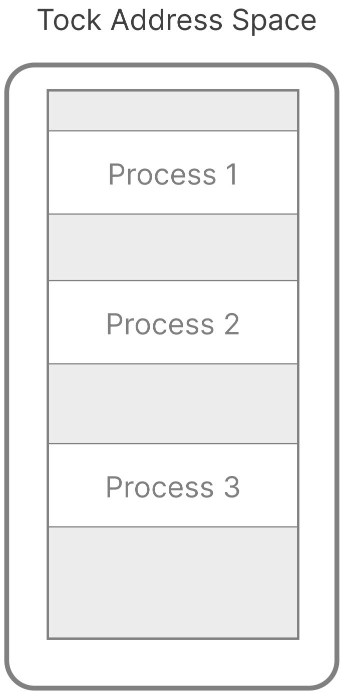


Tock process memory layout





Tock process memory layout

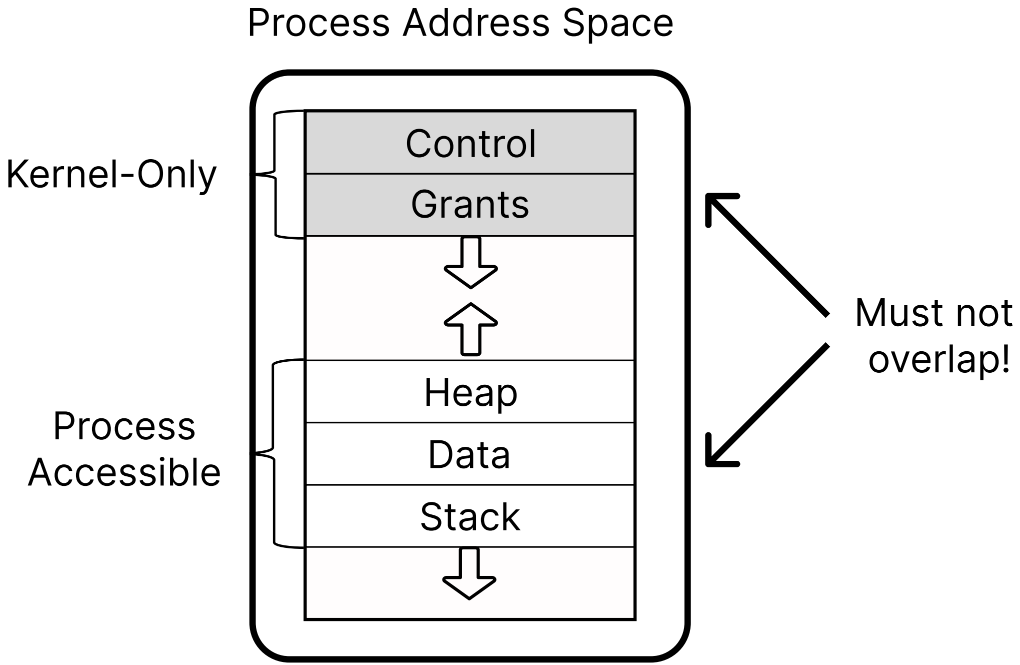


Tock Address Space

Process 1

Process 2

Process 3



Process Address Space

Kernel-Only

Control

Grants

Process Accessible

Heap

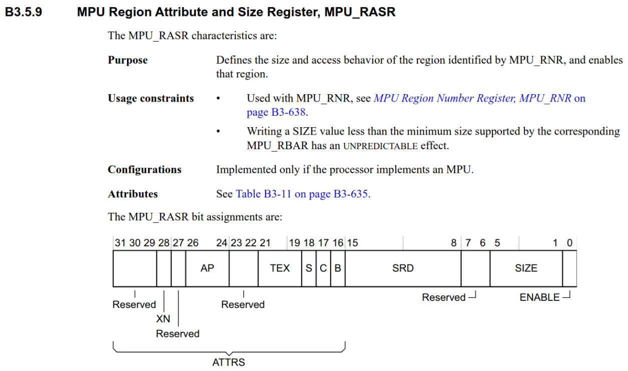
Data

Stack

Must not overlap!

Isolation is tricky on embedded systems

Must adhere to architectural constraints



B3.5.9 MPU Region Attribute and Size Register, MPU_RASR

The MPU_RASR characteristics are:

Purpose Defines the size and access behavior of the region identified by MPU_RNR, and enables that region.

Usage constraints
- Used with MPU_RNR, see *MPU Region Number Register, MPU_RNR* on page B3-638.
- Writing a SIZE value less than the minimum size supported by the corresponding MPU_RBAR has an UNPREDICTABLE effect.

Configurations Implemented only if the processor implements an MPU.

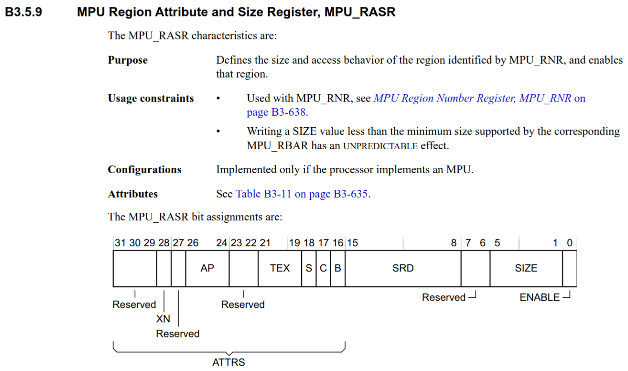
Attributes See Table B3-11 on page B3-635.

The MPU_RASR bit assignments are:

Isolation is tricky on embedded systems

Must adhere to architectural constraints

Must provide same safety across arches



B3.5.9 MPU Region Attribute and Size Register, MPU_RASR

The MPU_RASR characteristics are:

Purpose Defines the size and access behavior of the region identified by MPU_RNR, and enables that region.

Usage constraints

- Used with MPU_RNR, see *MPU Region Number Register, MPU_RNR* on page B3-638.
- Writing a SIZE value less than the minimum size supported by the corresponding MPU_RBAR has an UNPREDICTABLE effect.

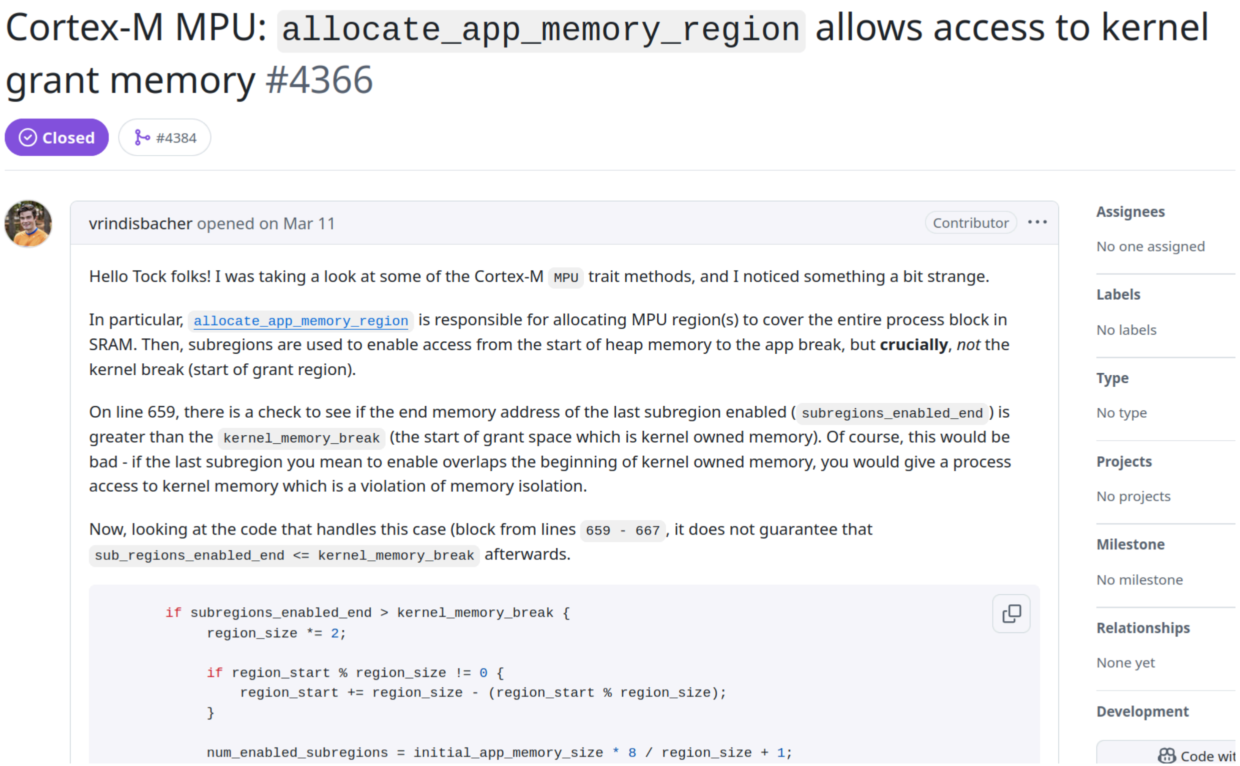
Configurations Implemented only if the processor implements an MPU.

Attributes See Table B3-11 on page B3-635.

The MPU_RASR bit assignments are:

31 30 29	28	27	26 24	23 22	21 19	18	17	16	15 8	7 6	5 1	0
Reserved	XN	Reserved	AP	Reserved	TEX	S	C	B	SRD	Reserved	SIZE	ENABLE

ATTRS



Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366

Closed · #4384

vrindisbacher opened on Mar 11 · Contributor · ···

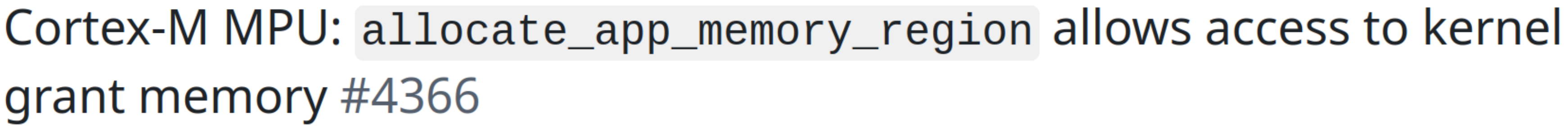
Hello Tock folks! I was taking a look at some of the Cortex-M `MPU` trait methods, and I noticed something a bit strange.

In particular, `allocate_app_memory_region` is responsible for allocating MPU region(s) to cover the entire process block in SRAM. Then, subregions are used to enable access from the start of heap memory to the app break, but **crucially**, *not* the kernel break (start of grant region).

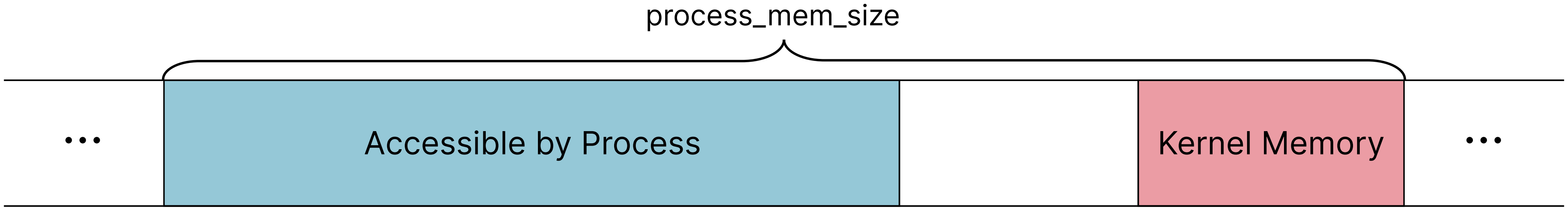
On line 659, there is a check to see if the end memory address of the last subregion enabled (`subregions_enabled_end`) is greater than the `kernel_memory_break` (the start of grant space which is kernel owned memory). Of course, this would be bad - if the last subregion you mean to enable overlaps the beginning of kernel owned memory, you would give a process access to kernel memory which is a violation of memory isolation.

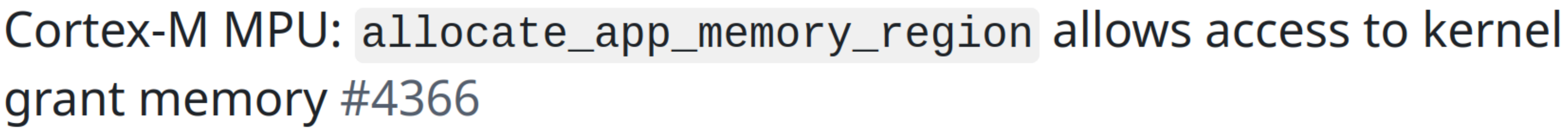
Now, looking at the code that handles this case (block from lines `659 - 667` , it does not guarantee that `sub_regions_enabled_end <= kernel_memory_break` afterwards.

```
        if subregions_enabled_end > kernel_memory_break {
            region_size *= 2;

            if region_start % region_size != 0 {
                region_start += region_size - (region_start % region_size);
            }

            num_enabled_subregions = initial_app_memory_size * 8 / region_size + 1;
```

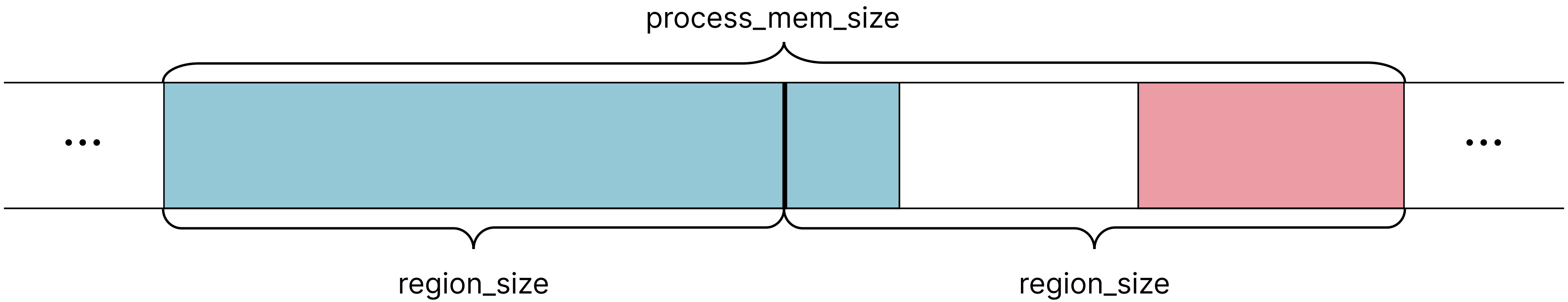


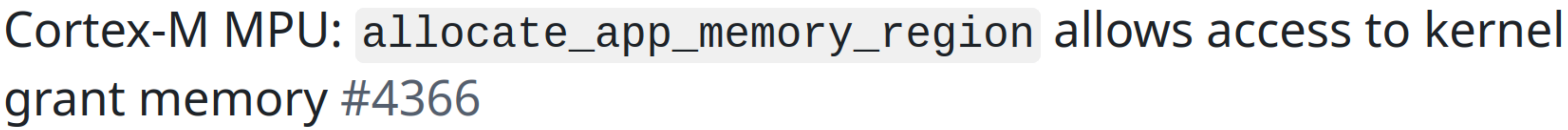
Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366



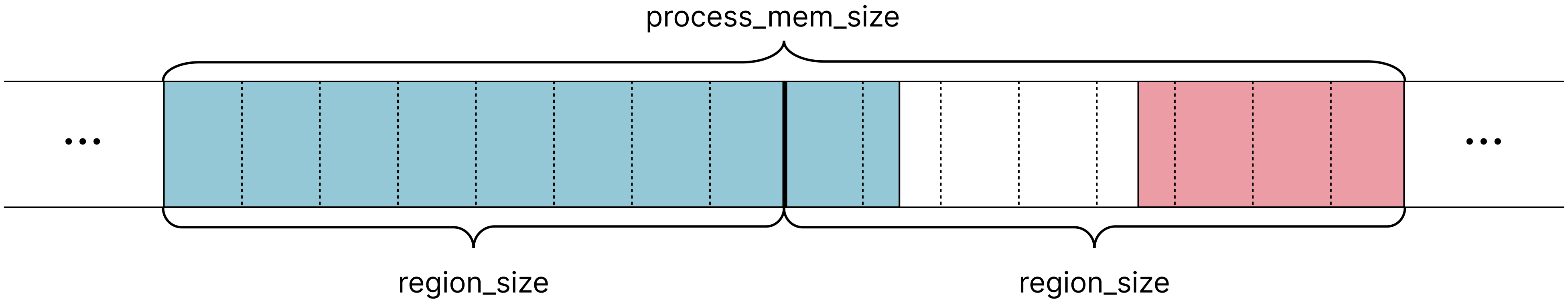


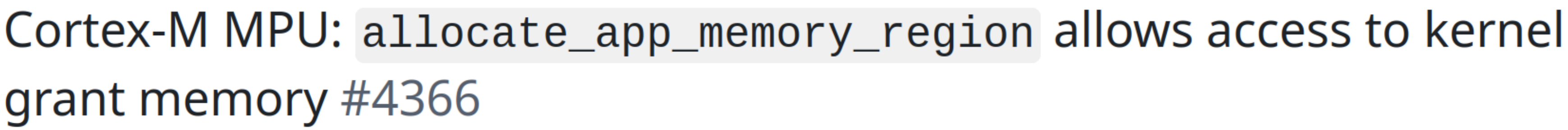
Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366



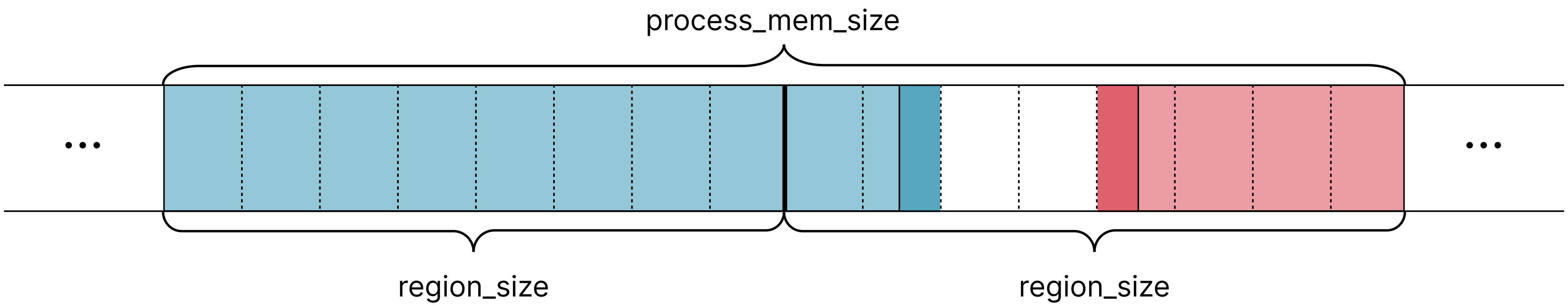


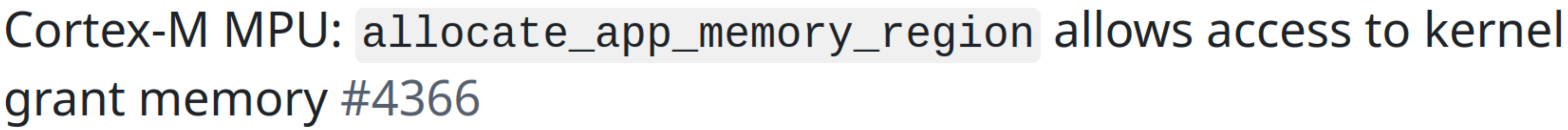
Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366



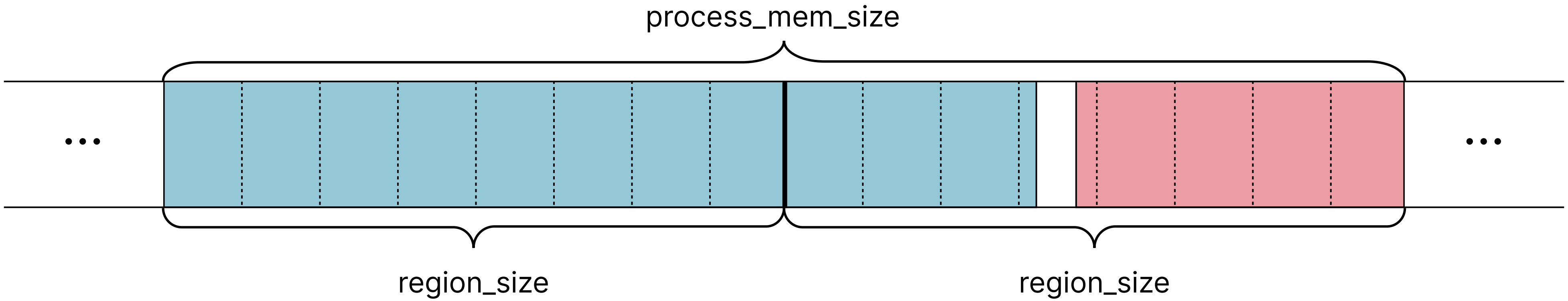


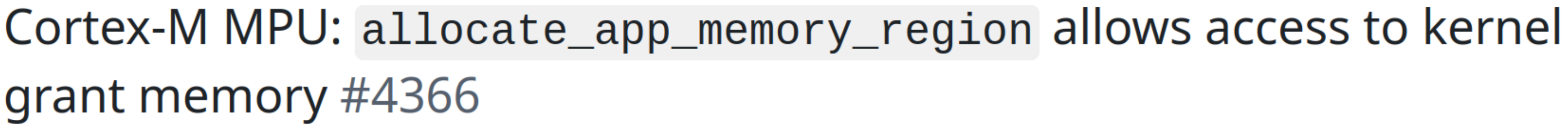
Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366



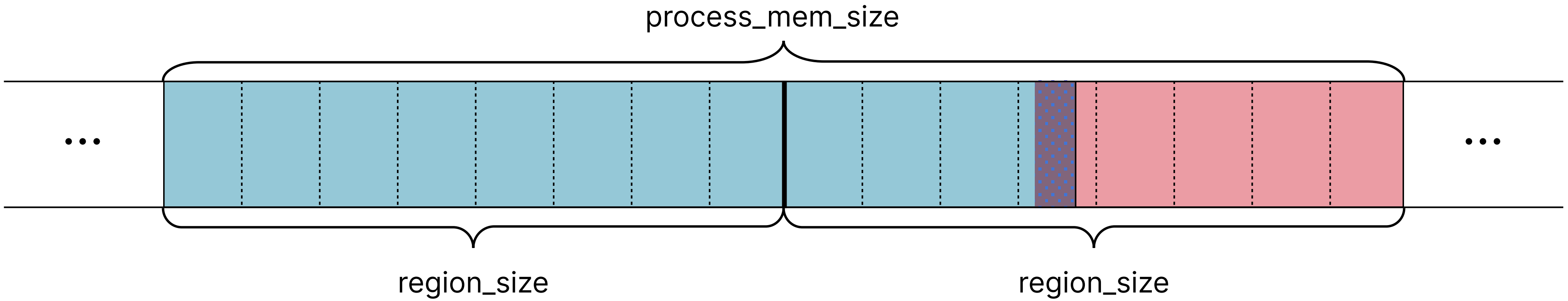


Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366





Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366



Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366

```
// If the last subregion covering app-owned memory overlaps the start of
// kernel-owned memory, we make the entire process memory block twice as
// big so there is plenty of space between app-owned and kernel-owned memory.
if subregions_enabled_end > kernel_memory_break {
    region_size *= 2;

    ...
}
```

Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366

```
// If the last subregion covering app-owned memory overlaps the start of
// kernel-owned memory, we make the entire process memory block twice as
// big so there is plenty of space between app-owned and kernel-owned memory.
if subregions_enabled_end > kernel_memory_break {
    region_size *= 2;
+    process_mem_size *= 2;
     ...
}
```

Rust

```rust
// When allocating memory for apps, we use two regions, each a power of two
// in size. By using two regions we halve their size, and also halve their
// alignment restrictions.
fn allocate_app_memory_region(
    &self,
    unallocated_memory_start: *const u8,
    unallocated_memory_size: usize,
    min_memory_size: usize,
    initial_app_memory_size: usize,
    initial_kernel_memory_size: usize,
    permissions: mpu::Permissions,
    config: &mut Self::MpuConfig,
) → Option<(*const u8, usize)> {
    // Check that no previously allocated regions overlap the unallocated
    // memory.
    for region in config.regions.iter() {
        if region.overlaps(unallocated_memory_start, unallocated_memory_size) {
            return None;
        }
    }

    // Make sure there is enough memory for app memory and kernel memory.
    let memory_size = cmp::max(
        min_memory_size,
        initial_app_memory_size + initial_kernel_memory_size,
    );

    // Size must be a power of two, so:
    // https://www.youtube.com/watch?v=ovo6zwv6DX4.
    let mut memory_size_po2 = math::closest_power_of_two(memory_size as u32) as usize;
    let exponent = math::log_base_two(memory_size_po2 as u32);

    // Check for compliance with the constraints of the MPU.
    if exponent < 9 {
        // Region sizes must be 256 bytes or larger to support subregions.
        // Since we are using two regions, and each must be at least 256
        // bytes, we need the entire memory region to be at least 512 bytes.
        memory_size_po2 = 512;
    } else if exponent > 32 {
        // Region sizes must be 4GB or smaller.
        return None;
    }

    // Region size is the actual size the MPU region will be set to, and is
    // half of the total power of two size we are allocating to the app.
    let mut region_size = memory_size_po2 / 2;

    // The region should start as close as possible to the start of the
    // unallocated memory.
    let mut region_start = unallocated_memory_start as usize;

    // If the start and length don't align, move region up until it does.
    if region_start % region_size ≠ 0 {
        region_start += region_size - (region_start % region_size);
    }

    // We allocate two MPU regions exactly over the process memory block,
    // and we disable subregions at the end of this region to disallow
    // access to the memory past the app break. As the app break later
    // increases, we will be able to linearly grow the logical region
    // covering app-owned memory by enabling more and more subregions. The
    // Cortex-M MPU supports 8 subregions per region, so the size of this
    // logical region is always a multiple of a sixteenth of the MPU region
    // length.

    // Determine the number of subregions to enable.
    // Want `round_up(app_memory_size / subregion_size)`.
    let mut num_enabled_subregions = initial_app_memory_size * 8 / region_size + 1;

    let subregion_size = region_size / 8;

    // Calculates the end address of the enabled subregions and the initial
    // kernel memory break.
    let subregions_enabled_end = region_start + num_enabled_subregions * subregion_size;
    let kernel_memory_break = region_start + memory_size_po2 - initial_kernel_memory_size;

    // If the last subregion covering app-owned memory overlaps the start of
    // kernel-owned memory, we make the entire process memory block twice as
    // big so there is plenty of space between app-owned and kernel-owned
    // memory.
    if subregions_enabled_end > kernel_memory_break {
+       memory_size_po2 *= 2;
        region_size *= 2;

        if region_start % region_size ≠ 0 {
            region_start += region_size - (region_start % region_size);
        }

        num_enabled_subregions = initial_app_memory_size * 8 / region_size + 1;
    }

    // Make sure the region fits in the unallocated memory.
    if region_start + memory_size_po2
        > (unallocated_memory_start as usize) + unallocated_memory_size
    {
        return None;
    }

    // Get the number of subregions enabled in each of the two MPU regions.
    let num_enabled_subregions0 = cmp::min(num_enabled_subregions, 8);
    let num_enabled_subregions1 = num_enabled_subregions.saturating_sub(8);

    let region0 = CortexMRegion::new(
        region_start as *const u8,
        region_size,
        region_start as *const u8,
        region_size,
        0,
        Some((0, num_enabled_subregions0 - 1)),
        permissions,
    )?;
```

Rust

```rust
    // We cannot have a completely unused MPU region
    let region1 = if num_enabled_subregions1 == 0 {
        CortexMRegion::empty(1)
    } else {
        CortexMRegion::new(
            (region_start + region_size) as *const u8,
            region_size,
            (region_start + region_size) as *const u8,
            region_size,
            1,
            Some((0, num_enabled_subregions1 - 1)),
            permissions,
        )?
    };

    config.regions[0] = region0;
    config.regions[1] = region1;
    config.is_dirty.set(true);

    Some((region_start as *const u8, memory_size_po2))
}
```

If only we could have the machine check this for us instead!

If only we could have the machine check this for us instead!

We can, with formal verification!

I. How Flux Works

II. Verifying Process Isolation in Tock

III. Future & Ongoing Work

Flux + Tock

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

$$B\{x : p\}$$

Base-type Value name Refinement

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

$$B\{x : p\}$$

Base-type Value name Refinement

“Set of values x of type B such that p is true”

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

```
i32{x : x % 2 == 0}
```

Base-type   Value name   Refinement

“Set of even integers“

A quick flux demo

```rust
#[flux_rs::sig(fn(num: i32{num % 2 == 0}) -> i32)]
fn divide_exact(num: i32) -> i32 {
    num / 2
}
```

A quick flux demo

```rust
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;


fn divide_exact(num: Even) -> i32 {
    num / 2
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;


fn divide_exact(num: Even) -> i32 {
    num / 2
}

fn main() {
    let x = 1;
    divide_exact(x);
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;



fn divide_exact(num: Even) -> i32 {
    num / 2
}

fn main() {
    let x = 1;
    divide_exact(x); // Compile-time error!
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;


fn divide_exact(num: Even) -> i32 {
    num / 2
}

fn main() {
    let x = 2;
    divide_exact(x);
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;

#[flux_rs::sig(fn(num: Even) -> i32{r: r < num})]
fn divide_exact(num: Even) -> i32 {
    num / 2
}

fn main() {
    let x = 2;
    divide_exact(x);
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;
// Compile-time error!
#[flux_rs::sig(fn(num: Even) -> i32{r: r < num})]
fn divide_exact(num: Even) → i32 {
    num / 2
}

fn main() {
    let x = 2;
    divide_exact(x);
}
```

A quick flux demo

```
#[flux::alias(type Even = i32{n: n % 2 == 0})]
type Even = i32;

#[flux_rs::sig(fn(num: Even) -> i32{r: r <= num})]
fn divide_exact(num: Even) -> i32 {
    num / 2
}

fn main() {
    let x = 2;
    divide_exact(x);
}
```

Refining RingBuffer

Tock

Rust

```rust
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T],
    head: usize,
    tail: usize,
}
```

Refined

Rust

Refining RingBuffer

Tock

Rust

```
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T],
    head: usize,
    tail: usize,
}
```

Refined

Rust

```
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T]{ring: ring.len() > 0},
    hd: usize,
    tl: usize,
}
```

Refining RingBuffer

Tock

Rust

```
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T],
    head: usize,
    tail: usize,
}
```

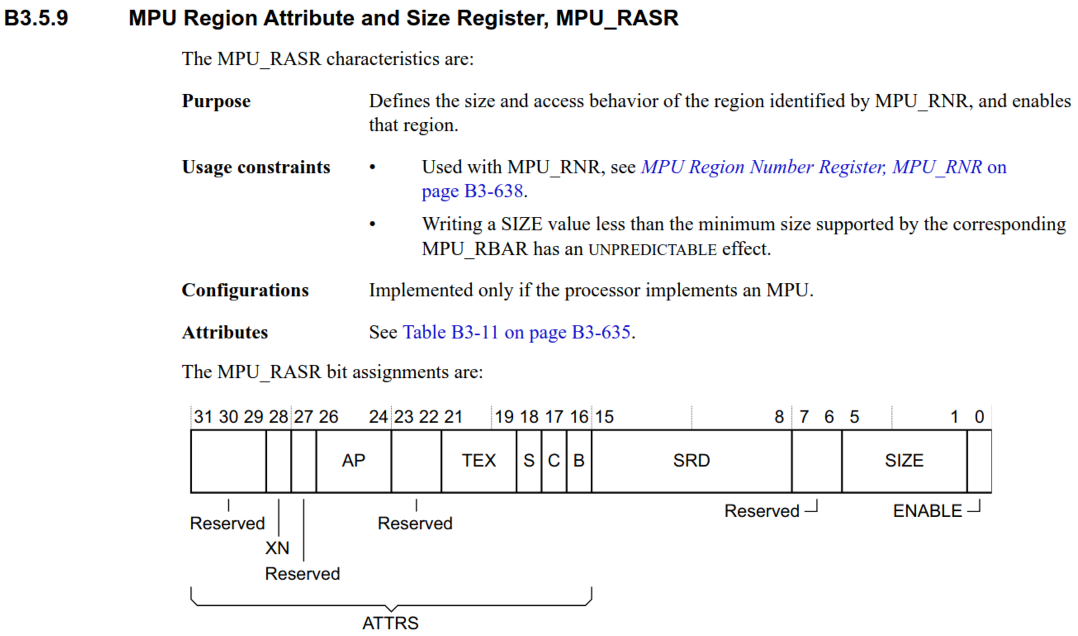
Refined

Rust

```
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T]{ring: ring.len() > 0},
    hd: usize{hd: hd < ring.len()},
    tl: usize{tl: tl < ring.len()},
}
```

Proof of Memory Isolation

```
MPUState{mpu : EnforcesTockIsolation(mpu)}
```



B3.5.9 MPU Region Attribute and Size Register, MPU_RASR

The MPU_RASR characteristics are:

Purpose Defines the size and access behavior of the region identified by MPU_RNR, and enables that region.

Usage constraints

- Used with MPU_RNR, see *MPU Region Number Register, MPU_RNR* on page B3-638.
- Writing a SIZE value less than the minimum size supported by the corresponding MPU_RBAR has an UNPREDICTABLE effect.

Configurations Implemented only if the processor implements an MPU.

Attributes See Table B3-11 on page B3-635.

The MPU_RASR bit assignments are:

31 30 29	28	27	26 24	23 22	21 19	18	17	16	15 8	7 6	5 1	0
			AP		TEX	S	C	B	SRD		SIZE	
Reserved	XN	Reserved		Reserved						Reserved		ENABLE

ATTRS

I. How Flux Works

II. Verifying Process Isolation in Tock

III. Future & Ongoing Work

Flux + Tock

What makes it hard

The kernel is relatively complex (~22K loc)

What makes it hard

The kernel is relatively complex (~22K loc)

Must track Process, MPUConfig, and HW State

What makes it hard

The kernel is relatively complex (~22K loc)

Must track Process, MPUConfig, and HW State

HW state does not work like normal data

What makes it hard

The kernel is relatively complex (~22K loc)

Must track Process, MPUConfig, and HW State

HW state does not work like normal data

Must work across architectures

What makes it hard

The kernel is relatively complex (~22K loc)

Must track Process, MPUConfig, and HW State

HW state does not work like normal data

Must work across architectures

Interrupts are hard :(

How much proof did this take?

Component	Source	Spec & Proof
Kernel	12,434	562
ARM MPU	2,486	506
Risc-V MPU	2,572	227
FluxARM	1,231	1900
Total	22,131	3,603

But how long does that take?

But how long does that take?

~3 seconds!

Ongoing & Future Work

- Support for x86?
- Verification-driven Optimization
- Verifying Core

Verification-driven optimization

Rust

```rust
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T]{ring: ring.len() > 0},
    hd: usize{hd: hd < ring.len()},
    tl: usize{tl: tl < ring.len()},
}
```

Verification-driven optimization

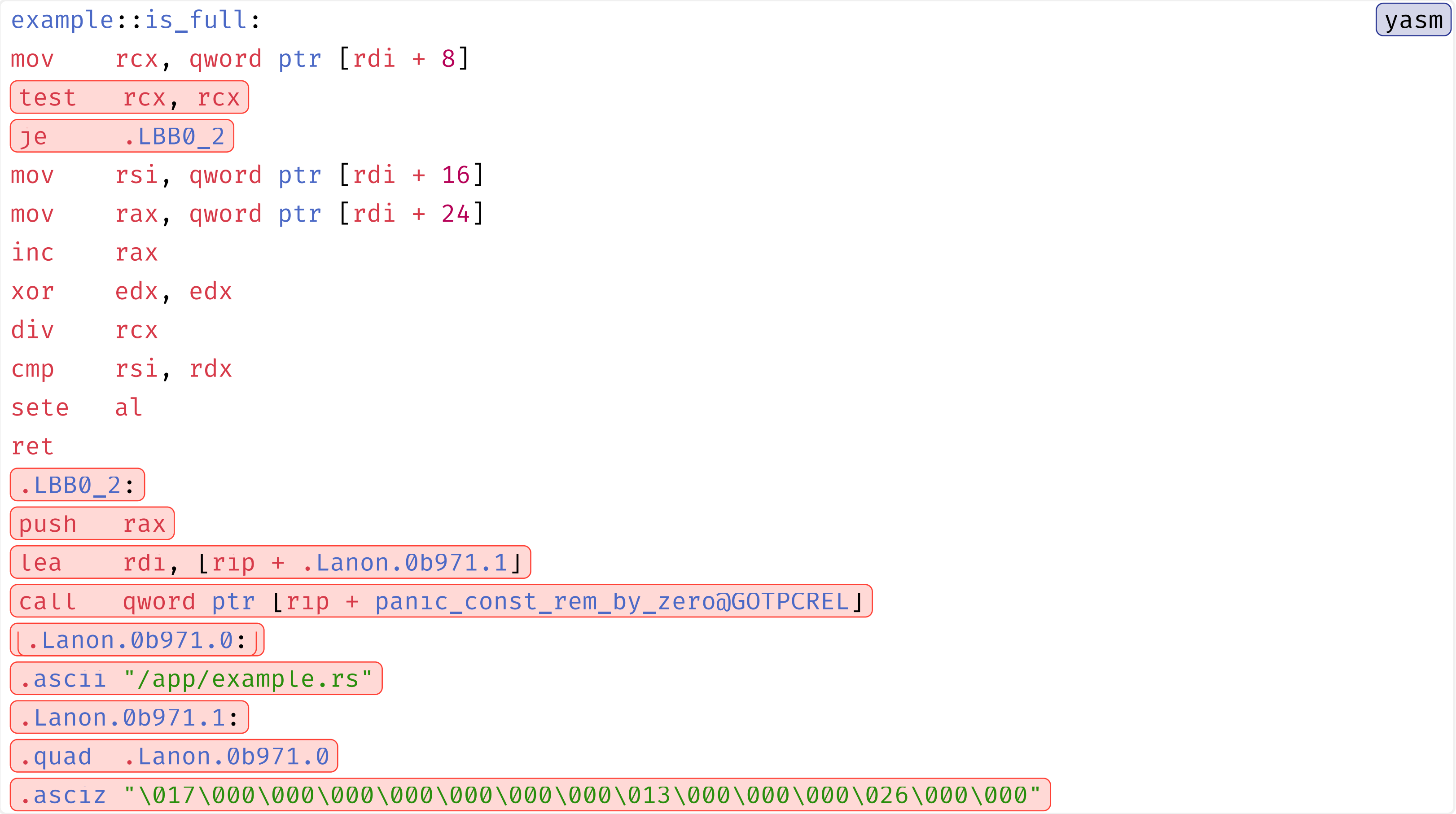
Rust

```rust
pub struct RingBuffer<'a, T: 'a> {
    ring: &'a mut [T]{ring: ring.len() > 0},
    hd: usize{hd: hd < ring.len()},
    tl: usize{tl: tl < ring.len()},
}
```

Rust

```rust
fn is_full(&self) -> bool {
 self.head == ((self.tail + 1) % self.ring.len)
}
```

```yasm
example::is_full:
mov     rcx, qword ptr [rdi + 8]
test    rcx, rcx
je      .LBB0_2
mov     rsi, qword ptr [rdi + 16]
mov     rax, qword ptr [rdi + 24]
inc     rax
xor     edx, edx
div     rcx
cmp     rsi, rdx
sete    al
ret
.LBB0_2:
push    rax
lea     rdi, [rip + .Lanon.0b971.1]
call    qword ptr [rip + panic_const_rem_by_zero@GOTPCREL]
.Lanon.0b971.0:
.ascii  "/app/example.rs"
.Lanon.0b971.1:
.quad   .Lanon.0b971.0
.asciz  "\017\000\000\000\000\000\000\000\013\000\000\000\026\000\000"
```



```yasm
example::is_full:
mov     rcx, qword ptr [rdi + 8]
test    rcx, rcx
je      .LBB0_2
mov     rsi, qword ptr [rdi + 16]
mov     rax, qword ptr [rdi + 24]
inc     rax
xor     edx, edx
div     rcx
cmp     rsi, rdx
sete    al
ret
.LBB0_2:
push    rax
lea     rdi, [rip + .Lanon.0b971.1]
call    qword ptr [rip + panic_const_rem_by_zero@GOTPCREL]
.Lanon.0b971.0:
.ascii  "/app/example.rs"
.Lanon.0b971.1:
.quad   .Lanon.0b971.0
.asciz  "\017\000\000\000\000\000\000\000\013\000\000\000\026\000\000"
```

Verified memory isolation in Tock

We used Flux to find a prevent bugs in Tock!

If you think Flux could help you, we should talk!

https://flux-rs.github.io