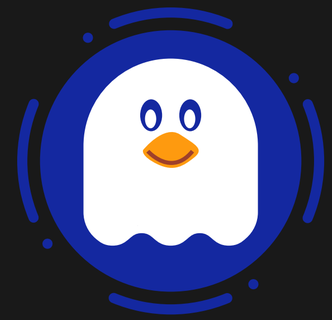
HRTIMER RUST ABSTRACTIONS

LPC 2024

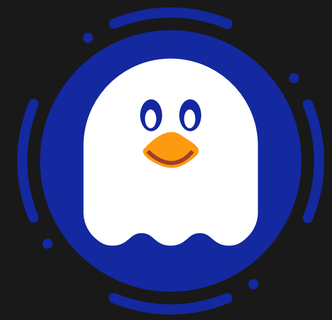
Andreas Hindborg

Samsung GOST



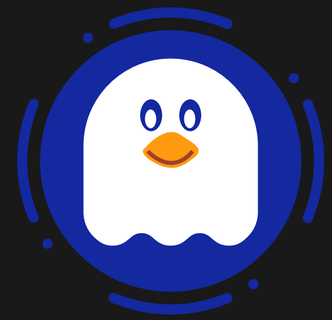
AGENDA 📄

- Rationale
- Prior work
- Current approach
- Stack allocated timers



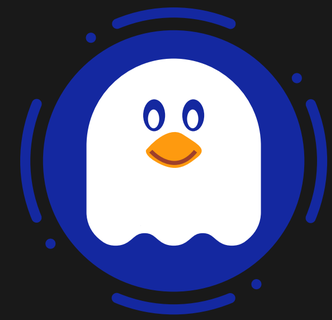
RATIONALE ❓

- First Rust block device driver patches merged in **6.11** 🎉
- C community asked for feature parity with C **null_blk**
  - We have a long list of things to upstream. Next in line are:
  - `module_params`
  - `configfs`
  - **`hrtimer`**
- `hrtimer` also a dependency for `rvkms`



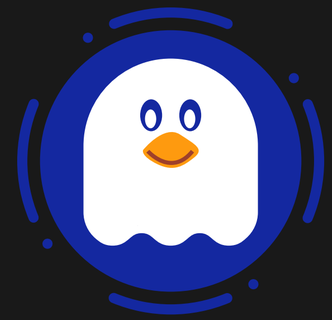
PRIOR WORK 🚧

- jitter-based timers by Boqun
- v1 hrtimer patches
  - tglx asked for a more complete API
  - issues with **drop** in (soft)irq context
- workqueue abstractions by Alice



C `hrtimer` API ⏰

- **Allocate** a `struct hrtimer` somewhere
- and **initialize** with `hrtimer_init` (and friends), setting `mode` and `clock_id`
- **Write** a function pointer into `hrtimer.funtion`
- **Schedule** timer with `hrtimer_start` and friends
- **Cancel** with `hrtimer_cancel`/`hrtimer_try_to_cancel`
- **Manipulate** timers: `hrtimer_forward` and friends
- `hrtimer_sleeper` and `hrtimer_nanosleep` for putting current task to **sleep**



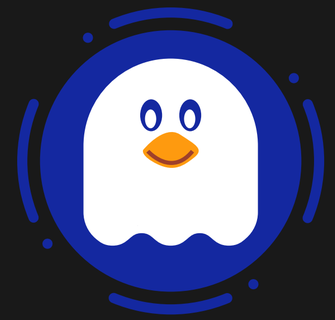
RUST TYPES 🧩

```
struct Timer<T> where T: TimerCallback;
trait HasTimer<T>;
trait TimerHandle;
trait TimerPointer { type TimerHandle: TimerHandle };
trait RawTimerCallback;
trait TimerCallback { type CallbackTarget<'a>: RawTimerCallback };
```

HasTimer 📜

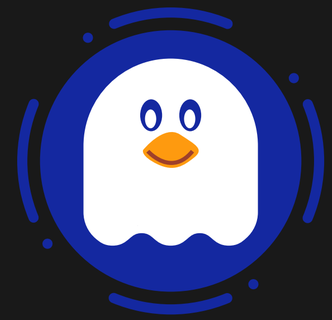

```rust
pub unsafe trait HasTimer<U> {
    const OFFSET: usize;

    unsafe fn raw_get_timer(ptr: *const Self) -> *const Timer<U> {
        // SAFETY: By the safety requirement of this trait, the trait
        // implementor will have a `Timer` field at the specified offset.
        unsafe { ptr.cast::<u8>().add(Self::OFFSET).cast::<Timer<U>>() }
    }

    unsafe fn timer_container_of(ptr: *mut Timer<U>) -> *mut Self
    where
        Self: Sized,
    {
        // SAFETY: By the safety requirement of this function and the `HasTimer`
        // trait, the following expression will yield a pointer to the `Self`
        // containing the timer addressed by `ptr`.
        unsafe { ptr.cast::<u8>().sub(Self::OFFSET).cast::<Self>() }
    }

    ...
}
```

impl_has_timer!




```
macro_rules! impl_has_timer {
    (
        impl$({$($generics:tt)*})?
            HasTimer<$timer_type:ty>
            for $self:ty
        { self.$field:ident }
        $($rest:tt)*
    ) => {
        // SAFETY: This implementation of `raw_get_timer` only compiles if the
        // field has the right type.
        unsafe impl$(<$($generics)*>)? $crate::hrtimer::HasTimer<$timer_type>  for $self {
            const OFFSET: usize = ::core::mem::offset_of!(Self, $field) as usize;

            #[inline]
            unsafe fn raw_get_timer(ptr: *const Self) -> *const $crate::hrtimer::Timer<$timer_type> {
                // SAFETY: The caller promises that the pointer is not dangling.
                unsafe {
                    ::core::ptr::addr_of!((*ptr).$field)
                }
            }
        }
    }
}
```

impl_has_timer!



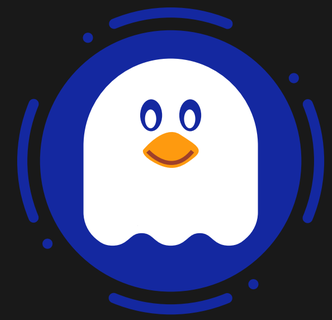

```rust
#[pin_data]
struct ArcIntrusiveTimer {
    #[pin]
    timer: Timer<Self>,
    flag: AtomicBool,
}

impl_has_timer! {
    impl HasTimer<Self> for ArcIntrusiveTimer { self.timer }
}
```

⬇️

```rust
unsafe impl $crate::hrtimer::HasTimer<Self> for ArcIntrusiveTimer {
    const OFFSET: usize = crate::hint::must_use({ builtin #offset_of(Self, timer) }) as usize;
    #[inline]
    unsafe fn raw_get_timer(ptr: *const Self) -> *const $crate::hrtimer::Timer<Self> {
        unsafe { &raw const ((*ptr).timer) }
    }
}
```

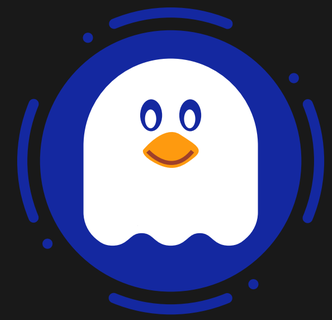
TimerHandle 👜

```rust
/// # Safety
///
/// When dropped, the timer represented by this handle must be cancelled, if it
/// is armed. If the timer handler is running when the handle is dropped, the
/// drop method must wait for the handler to finish before returning.
pub unsafe trait TimerHandle {
    fn cancel(&mut self) -> bool;
}
```



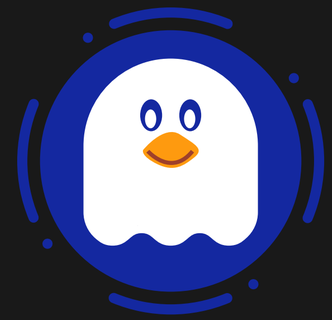
TimerPointer 👉

```rust
pub trait TimerPointer: Sync + Sized {
    /// ...
    /// If the timer is armed or if the timer callback is running when the
    /// handle is dropped, the drop method of `TimerHandle` should not return
    /// until the timer is unarmed and the callback has completed.
    ///
    /// Note: It must be safe to leak the handle.
    type TimerHandle: TimerHandle;

    /// Schedule the timer after `expires` time units. If the timer was already
    /// scheduled, it is rescheduled at the new expiry time.
    fn schedule(self, expires: u64) -> Self::TimerHandle;
}
```



RawTimerCallback ☎

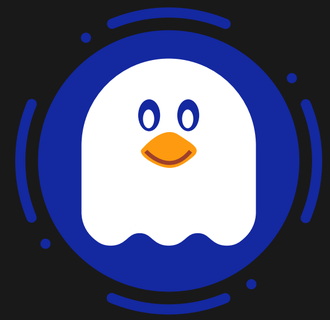
```rust
pub trait RawTimerCallback {
    /// Callback to be called from C when timer fires.
    ///
    /// # Safety
    ///
    /// Only to be called by C code in `hrtimer` subsystem. `ptr` must point to
    /// the `bindings::hrtimer` structure that was used to schedule the timer.
    unsafe extern "C" fn run(ptr: *mut bindings::hrtimer) -> bindings::hrtimer_restart;
}
```



TimerCallback 📱

```rust
pub trait TimerCallback {
    /// The type that was used for scheduling the timer.
    type CallbackTarget<'a>: RawTimerCallback;

    /// Called by the timer logic when the timer fires.
    fn run(this: Self::CallbackTarget<'_>) -> TimerRestart
    where
        Self: Sized;
}
```

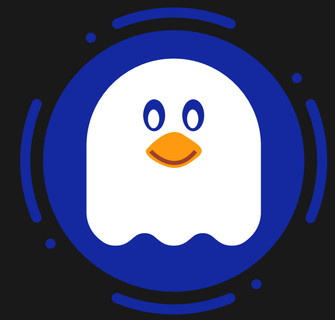
EXAMPLE 




```rust
#[pin_data]
struct ArcIntrusiveTimer {
    #[pin]
    timer: Timer<Self>,
    flag: AtomicBool,
}

impl ArcIntrusiveTimer {
    fn new() -> impl PinInit<Self, kernel::error::Error> {
        try_pin_init!(Self {
            timer <- Timer::new(),
            flag: AtomicBool::new(false),
        })
    }
}
```

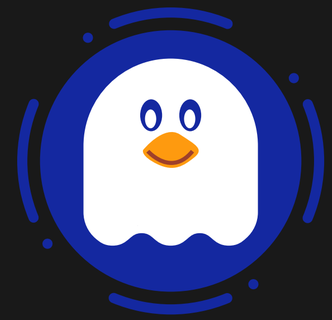
EXAMPLE 


```rust
impl TimerCallback for ArcIntrusiveTimer {
    type CallbackTarget<'a> = Arc<Self>;

    fn run(this: Self::CallbackTarget<'_>) -> TimerRestart {
        pr_info!("Timer called\n");
        this.flag.store(true, Ordering::Relaxed);
        TimerRestart::NoRestart
    }
}

impl_has_timer! {
    impl HasTimer<Self> for ArcIntrusiveTimer { self.timer }
}

fn arc_timer() -> Result<()> {
    let has_timer = Arc::pin_init(ArcIntrusiveTimer::new(), GFP_KERNEL)?;
    let _handle = has_timer.clone().schedule(some_time_in_the_future);
    while !has_timer.flag.load(Ordering::Relaxed) {
        core::hint::spin_loop()
    }

    pr_info!("Flag raised\n");
    Ok(())
}
```



STACK ALLOCATED TIMERS 

```rust
fn stack_timer() -> Result<()> {
    use kernel::stack_try_pin_init;

    stack_try_pin_init!( let has_timer =? PinIntrusiveTimer::new() );
    let _handle = has_timer.as_ref().schedule(some_time_in_the_future);

    while !has_timer.flag.load(Ordering::Relaxed) {
        core::hint::spin_loop()
    }

    pr_info!("Flag raised\n");
    Ok(())
}
```



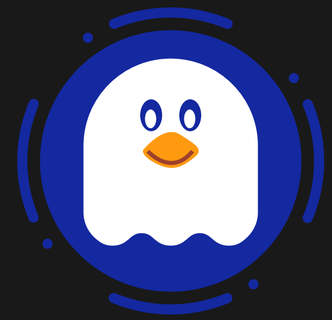
STACK ALLOCATED TIMERS 💥

```rust
fn stack_timer() -> Result<()> {
    use kernel::stack_try_pin_init;

    stack_try_pin_init!( let has_timer =? PinIntrusiveTimer::new() );
    let _handle = has_timer.as_ref().schedule(some_time_in_the_future);

    core::mem::forget(_handle); // BOOM

    Ok(())
}
```

UNSAFE STACK ALLOCATED TIMERS

SAMSUNG

```rust
/// # Safety
///
/// Implementers of this trait must ensure that instances of types implementing
/// [`UnsafeTimerPointer`] outlives any associated [`TimerPointer::TimerHandle`]
/// instances.
pub unsafe trait UnsafeTimerPointer: Sync + Sized {
    /// # Safety
    ///
    /// If the timer is armed or if the timer callback is running when the
    /// handle is dropped, the drop method of `TimerHandle` must not return
    /// before the timer is unarmed and the callback has completed.
    type TimerHandle: TimerHandle;

    /// # Safety
    ///
    /// Caller promises to either keep the timer structure alive until the timer
    /// is dead, or run the destructor of the returned `Self::TimerHandle`.
    unsafe fn schedule(self, expires: u64) -> Self::TimerHandle;
}
```



SAMSUNG

UNSAFE STACK ALLOCATED TIMERS

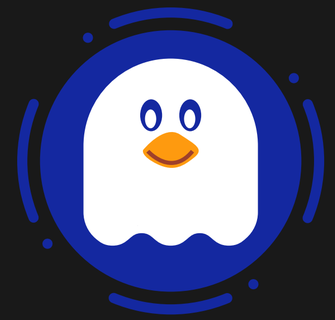
```rust
fn unsafe_stack_timer() -> Result<()> {
    use kernel::stack_try_pin_init;

    stack_try_pin_init!( let has_timer =? PinIntrusiveTimer::new() );

    // SAFETY: We do not unwind the stack before the timer has fired.
    let _handle = unsafe { has_timer.as_ref().schedule(some_time_in_the_future) };

    while !has_timer.flag.load(Ordering::Relaxed) {
        core::hint::spin_loop()
    }

    pr_info!("Flag raised\n");
    Ok(())
}
```

SAFE STACK ALLOCATED TIMERS


```rust
/// # Safety
///
/// Implementers must ensure that `schedule_scoped` does not return until the
/// timer is dead and the timer handler is not running.
pub unsafe trait ScopedTimerPointer {
    fn schedule_scoped<T, F>(self, expires: u64, f: F) -> T
    where
        F: FnOnce() -> T;
}
```

```rust
unsafe impl<U> ScopedTimerPointer for U
where
    U: UnsafeTimerPointer,
{

    fn schedule_scoped<T, F>(self, expires: u64, f: F) -> T
    where
        F: FnOnce() -> T,
    {
        let handle = unsafe { UnsafeTimerPointer::schedule(self, expires) };
        let t = f();
        drop(handle);
        t
    }
}
```

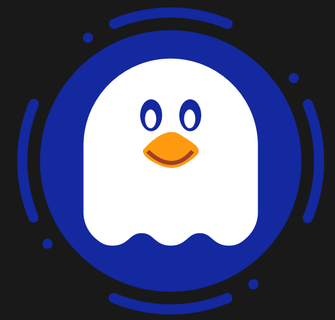
SAFE STACK ALLOCATED TIMERS




```rust
fn safe_stack_timer() -> Result<()> {
    use kernel::stack_try_pin_init;

    stack_try_pin_init!( let has_timer =? PinIntrusiveTimer::new() );

    has_timer.as_ref().schedule_scoped(some_time_in_the_future, || {
        while !has_timer.flag.load(Ordering::Relaxed) {
            core::hint::spin_loop()
        }

        pr_info!("Flag raised\n");
    });

    Ok(())
}
```

NEXT STEPS 👣

- Does this look good to you?
  - v2 is on list - reviews appreciated
- Still missing 🔧
  - sleeper - putting task to sleep for duration
  - clock source selection
  - timer introspection functions
  - accessing callback target through timer handle



THANK YOU!