# Panic Recovery in Rust-based Embedded Systems

Zhiyao Ma, Guojun Chen, Lin Zhong

Efficient Computing Lab
Yale University

YALE UNIVERSITY COMPUTER SCIENCE

# Tasks May Crash Due to (Language) Exception

- Language exception
  - Rust `panic`, C++ `std::runtime_error`, etc.
- Various sources of exception
  - Bugs in program code
  - Failed assertions
  - Transient hardware error
- Embedded systems pose greater challenges
  - Unattended
  - Mission critical


# Recover by Unwinding & Restarting

- Reclaim resources by unwinding a task's stack.
  - Force function returns out of `main()`.
  - Destruct initialized objects.

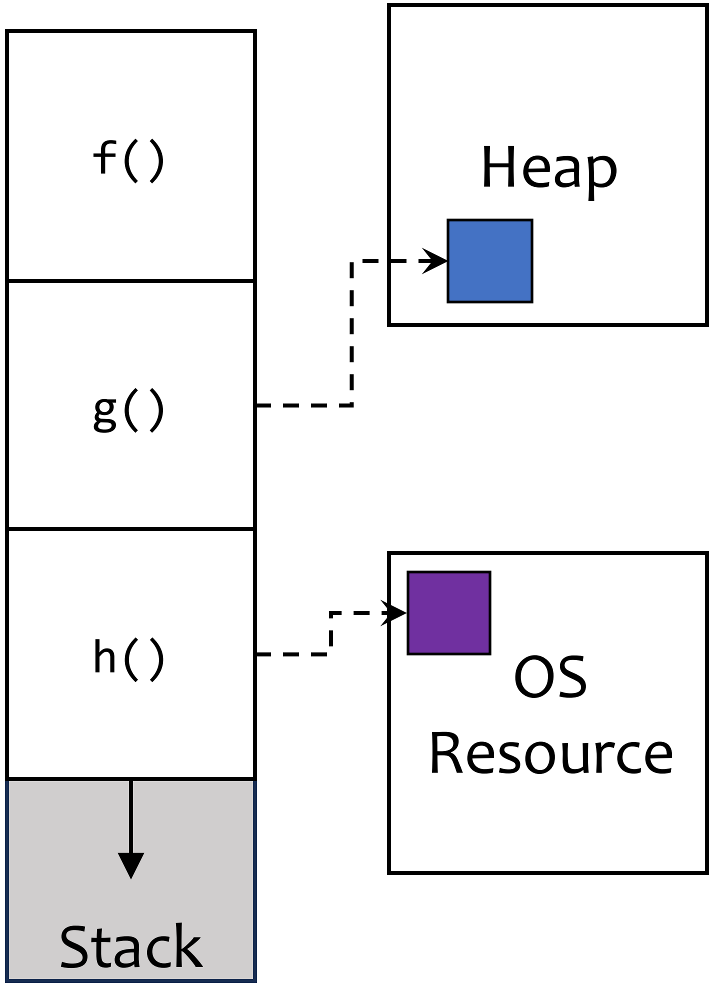
- Resume execution by restarting the task.
  - Run again from `main()`.

- Applicable to soft real-time systems.

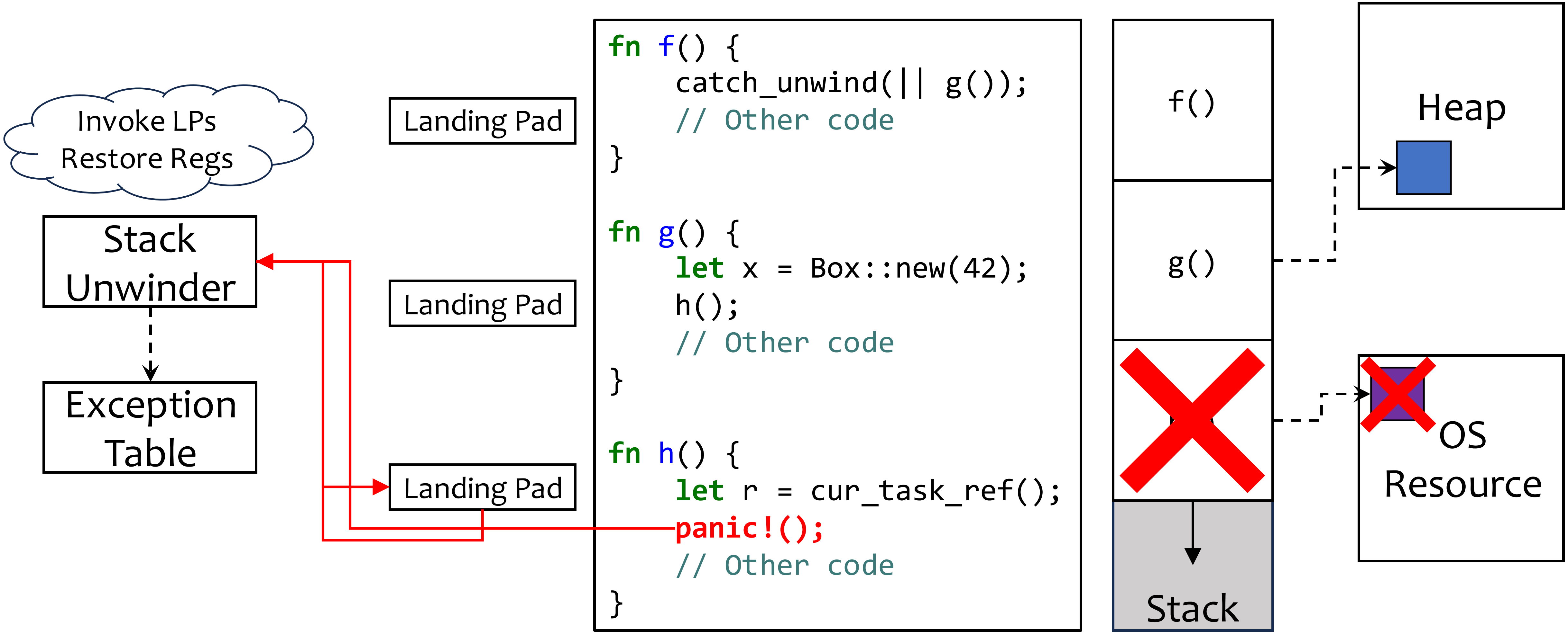
# How Stack Unwinding works

# Unwinder forces return until `catch_unwind`

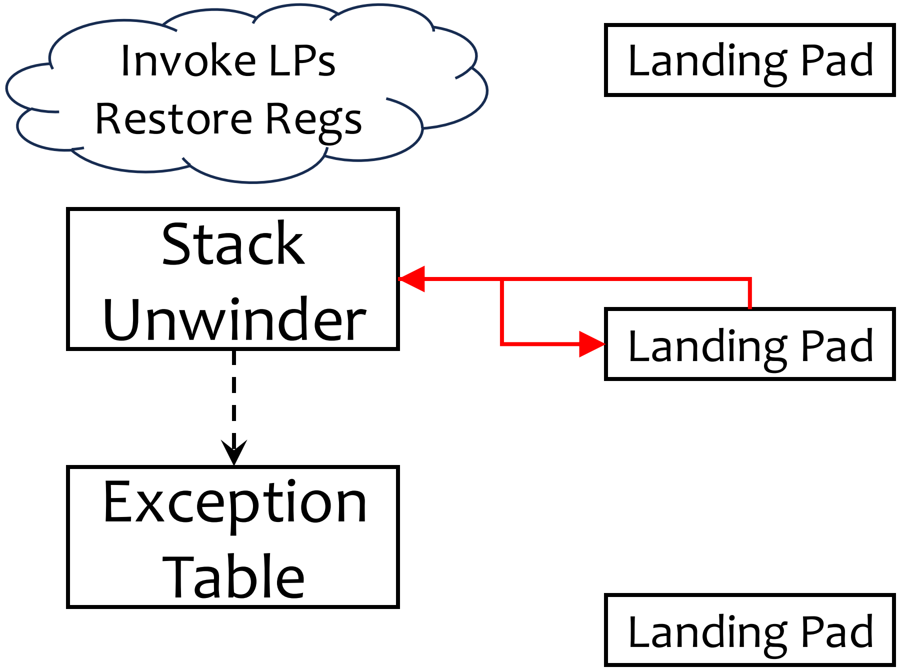
```
fn f() {
    catch_unwind(|| g());
    // Other code
}

fn g() {
    let x = Box::new(42);
    h();
    // Other code
}

fn h() {
    let r = cur_task_ref();
    panic!();
    // Other code
}
```



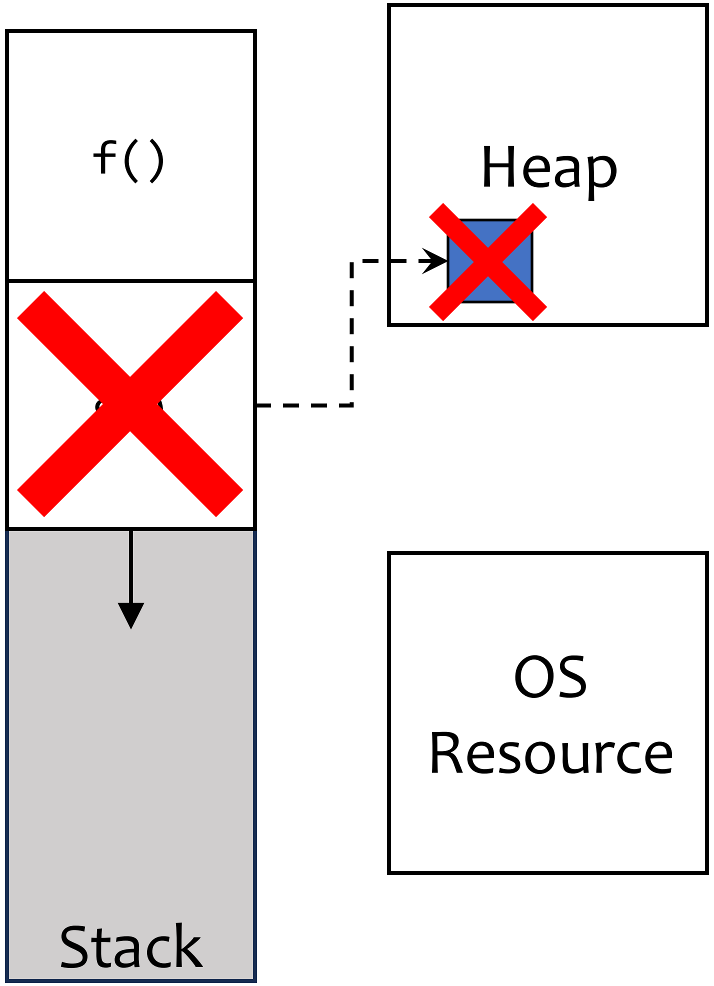
# Unwinder forces return until catch_unwind



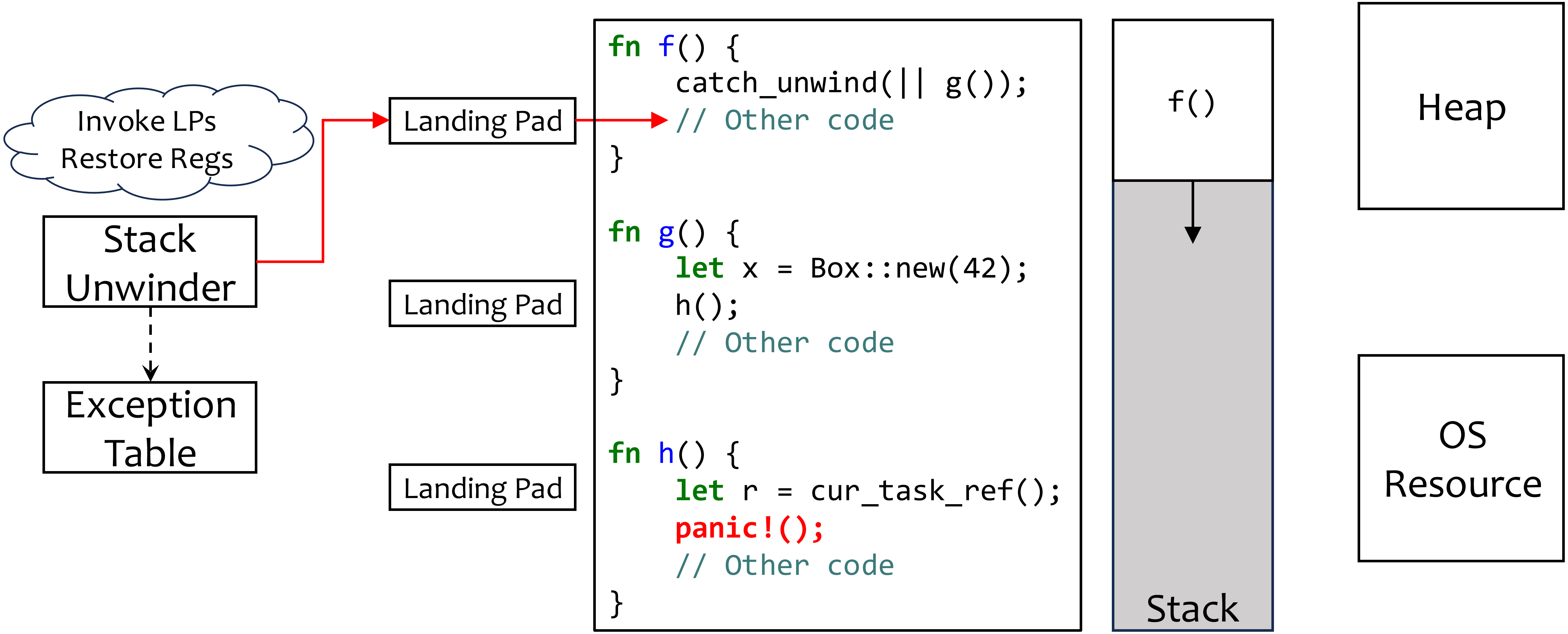
# Unwinder forces return until catch_unwind



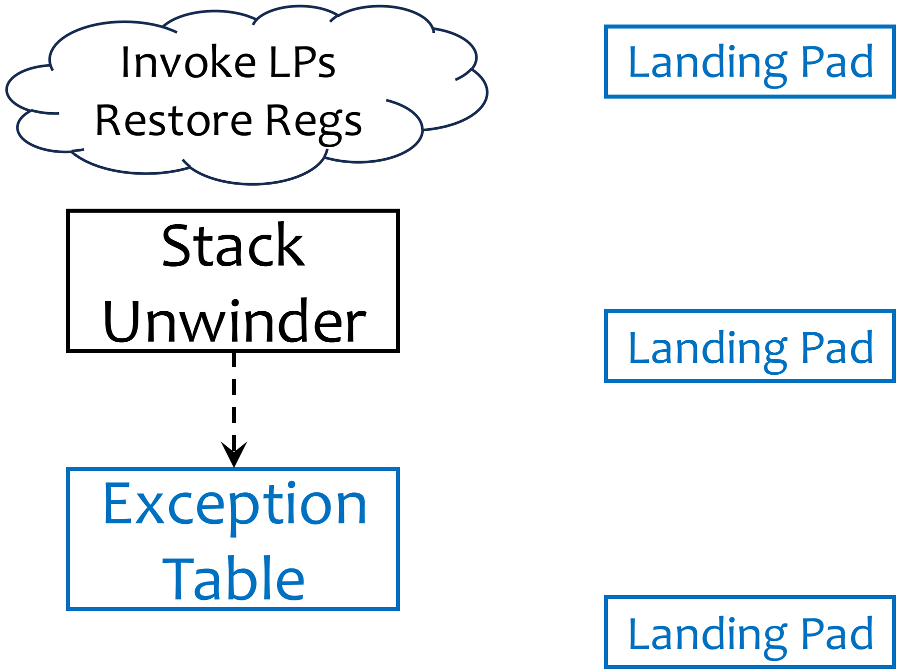

```
fn f() {
    catch_unwind(|| g());
    // Other code
}

fn g() {
    let x = Box::new(42);
    h();
    // Other code
}

fn h() {
    let r = cur_task_ref();
    panic!();
    // Other code
}
```



# Unwinder forces return until catch_unwind

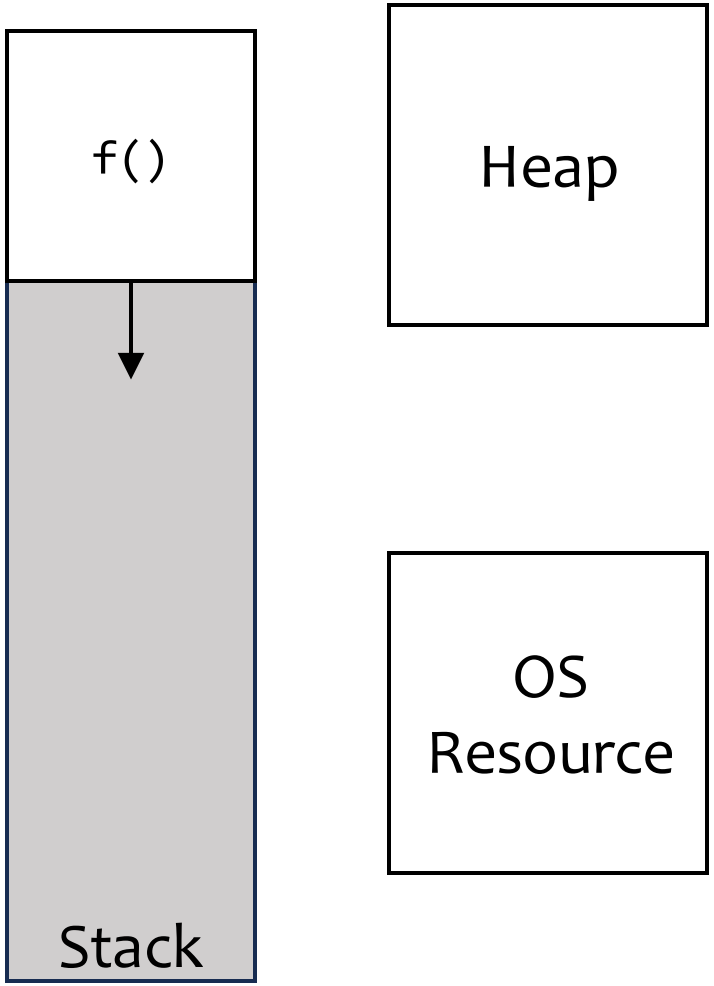


# Unwinder forces return until `catch_unwind`




Compiler Generated

```
fn f() {
    catch_unwind(|| g());
    // Other code
}

fn g() {
    let x = Box::new(42);
    h();
    // Other code
}

fn h() {
    let r = cur_task_ref();
    panic!();
    // Other code
}
```



# Challenges of Unwinding on Embedded Systems

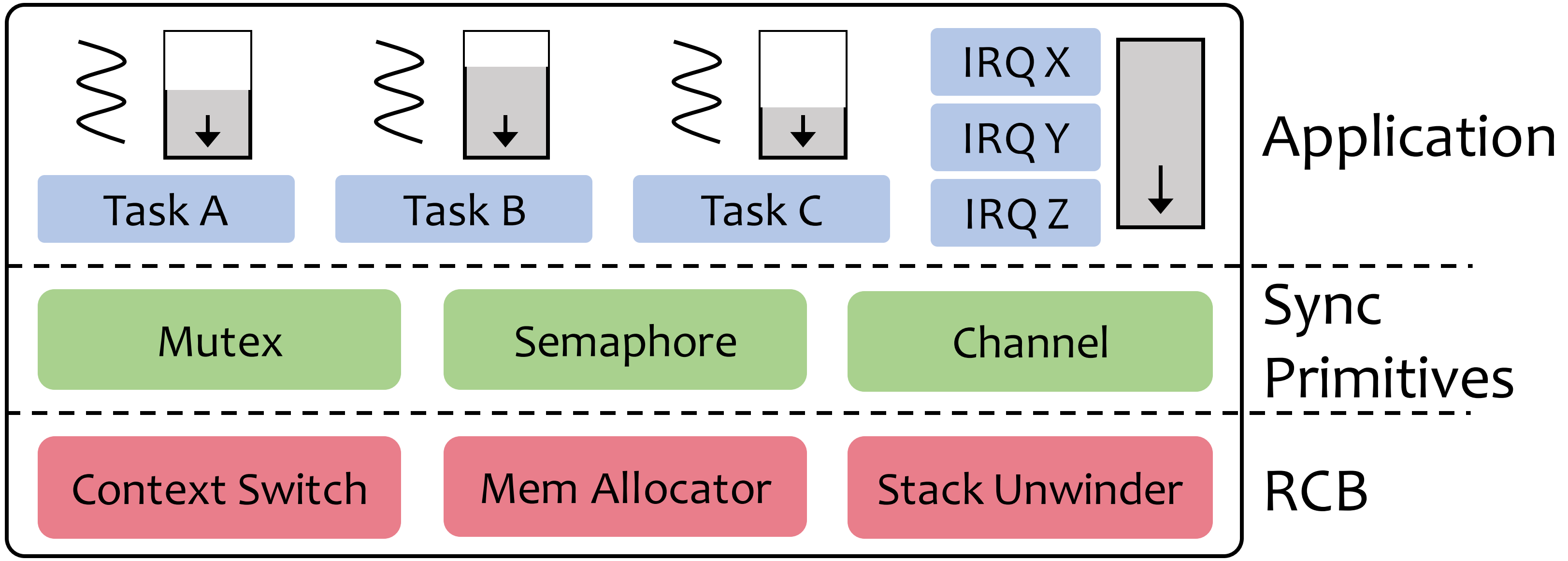
- Storage overhead
  - Unwinder logic, landing pads, exception table
  - Up to 5x size increase
  - Rust panic's simplicity: only `Exception` type, always fatal, no re-throw, etc.
  - Arm EHABI more compact exception table format

- Performance overhead
  - Unwinder interpreting exception table
  - Up to 1000x slow down
  - Concurrent restart-able task abstraction (facilitated by Rust)

# Hopter: An Embedded OS Capable of Recovering from Panic

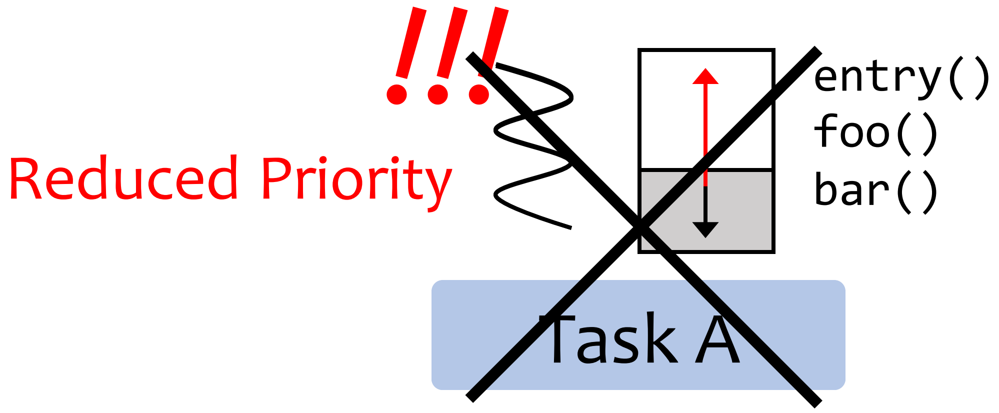
# RCB Supports Recovery from Panic

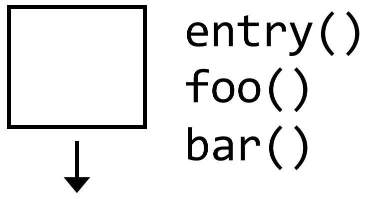
- Reliable Computing Base (RCB)
- Can recover panics from components above RCB



# Acceleration by Concurrent Task Restart

- Start a cloned task from `entry()` while unwinding the panicked one.
- Unwinding uses otherwise idle CPU time.



# Restartable Task Abstraction

```
pub fn create_restartable_task<F, A>(
    entry_closure: F, entry_argument: A,
    stack_size: usize, priority: u8) where
F: FnOnce(A) + Clone + Send + Sync + 'static,
A: Clone + Send + Sync + 'static, { ... }
```

Application Developer Supply

Clone: safe to duplicate

Send: safe to move across task context

Sync: safe to access from multiple task context

# Restartable Task Abstraction – Continued

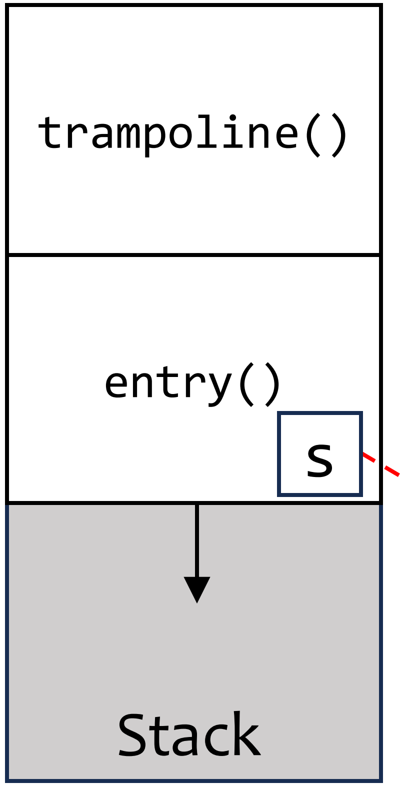
```
fn restartable_task_entry_trampoline<F, A>(
entry_closure: &F, entry_argument: &A)
where ..., {
    catch_unwind_with_arg(
        entry_closure.clone(),
        entry_argument.clone());
}
```

OS catches the panic outside of task's entry function.

# Priority Inheritance During Unwinding

Unwinding

Priority: Lowest

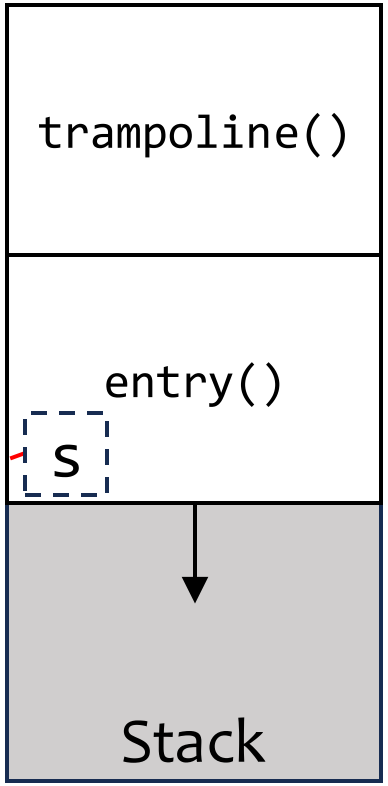



Task A

```
fn trampoline() {
    catch_unwind(|| entry());
    // Other code
}

static S: Mutex<T>
    = Mutex::new(T::new());

fn entry() {
    let s = S.lock();
    // Work on `s`
    panic!();
}
```

Running

Priority: Medium

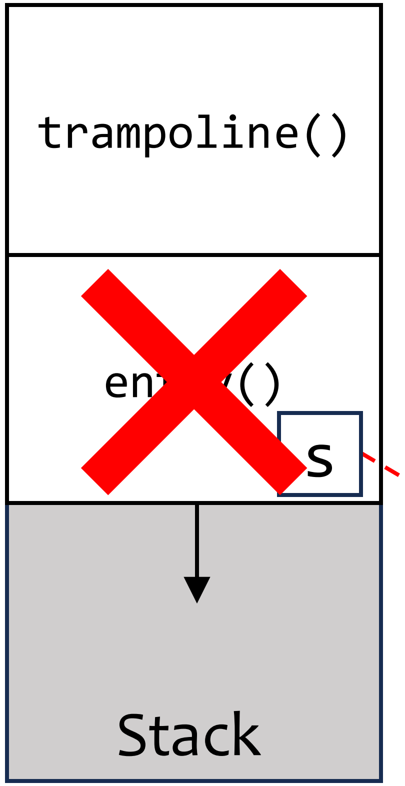



Task A’

# Priority Inheritance During Unwinding

Unwinding

Priority: Medium

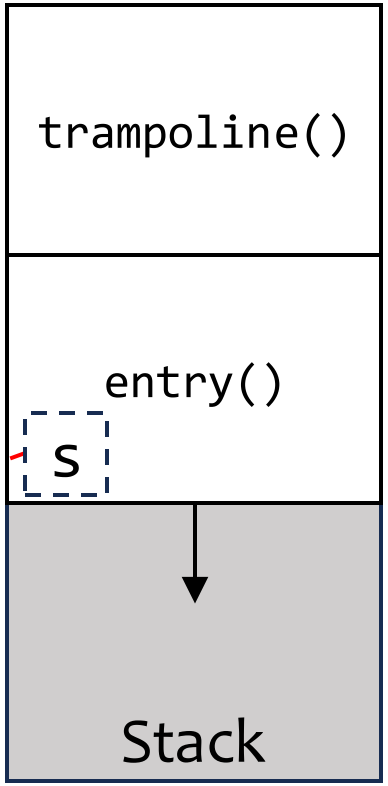



Task A

```
fn trampoline() {
    catch_unwind(|| entry());
    // Other code
}

static S: Mutex<T>
    = Mutex::new(T::new());

fn entry() {
    let s = S.lock();
    // Work on `s`
    panic!();
}
```

Running

Priority: Medium

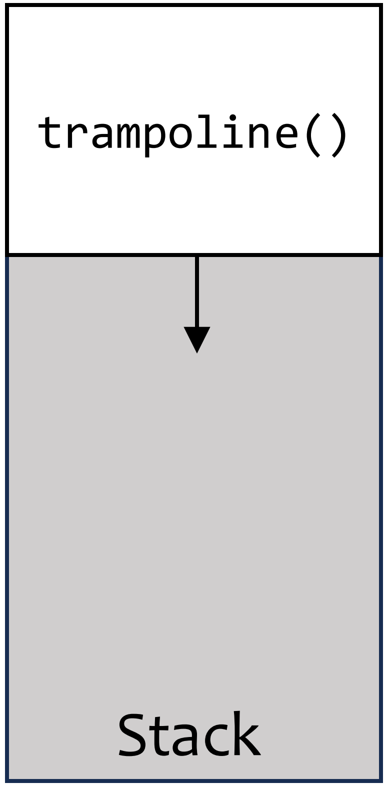



Task A’

# Priority Inheritance During Unwinding

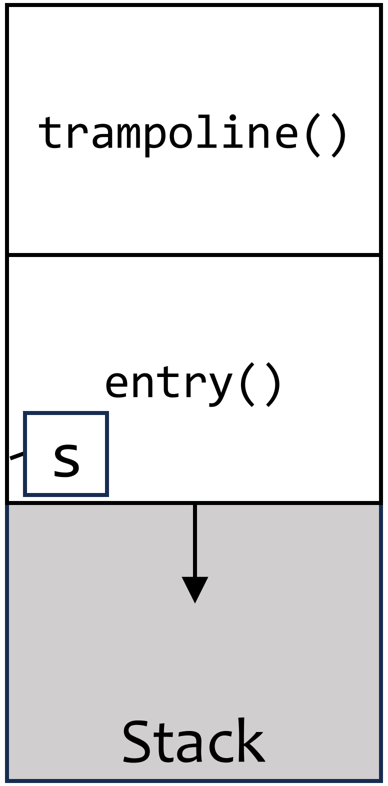
Unwinding

Priority: Lowest




Task A

```
fn trampoline() {
    catch_unwind(|| entry());
    // Other code
}

static S: Mutex<T>
    = Mutex::new(T::new());

fn entry() {
    let s = S.lock();
    // Work on `s`
    panic!();
}
```




Task A’

Running

Priority: Medium

# Rust Facilitates Concurrent Task Restart

- Rust precludes race conditions.
  - Between panicked and restarted tasks, accessing static variables.
  - Safe Rust disallows mutable static variables and requires `Sync` trait.
  - ➔ Must use `Mutex` around mutable static variables or atomic types
  - ➔ `Mutex` priority inheritance works as normal

# Rust Facilitates Concurrent Task Restart

- Rust precludes race conditions.
  - Between panicked and restarted tasks, accessing static variables.
  - Safe Rust disallows mutable static variables and requires `Sync` trait.
  - ➔ Must use `Mutex` around mutable static variables or atomic types
  - ➔ `Mutex` priority inheritance works as normal

- Rust disambiguates fatal exception.
  - Rust panic is always fatal.
  - C++ exception not always fatal: `std::stoi()` throws `std::invalid_argument()`.

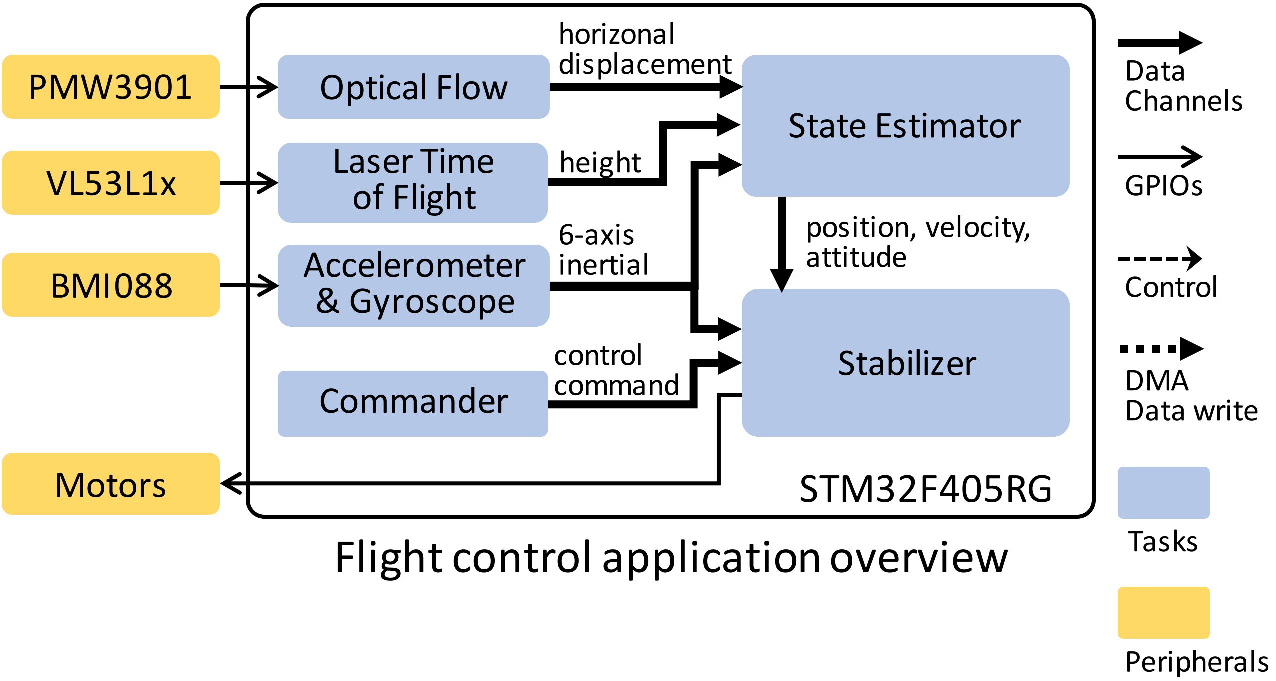
  ➔ Concurrent restart only makes sense for fatal exception.

# Evaluation with a Flying Drone

An Example of Soft Real-time System

# Running Flight Control Application on Hopter

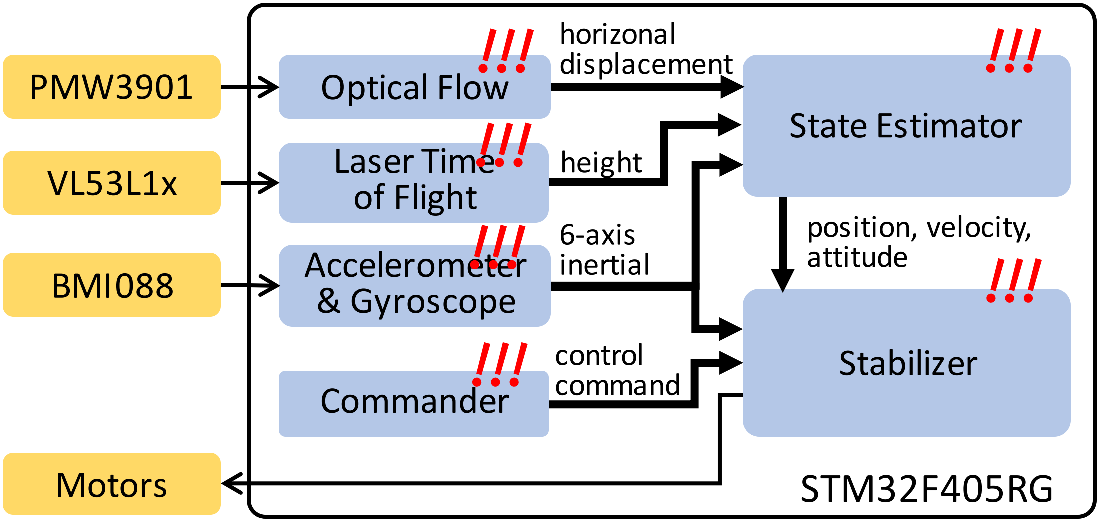
- Crazyflie 2.1
- COTS miniature drone
- Originally with FreeRTOS



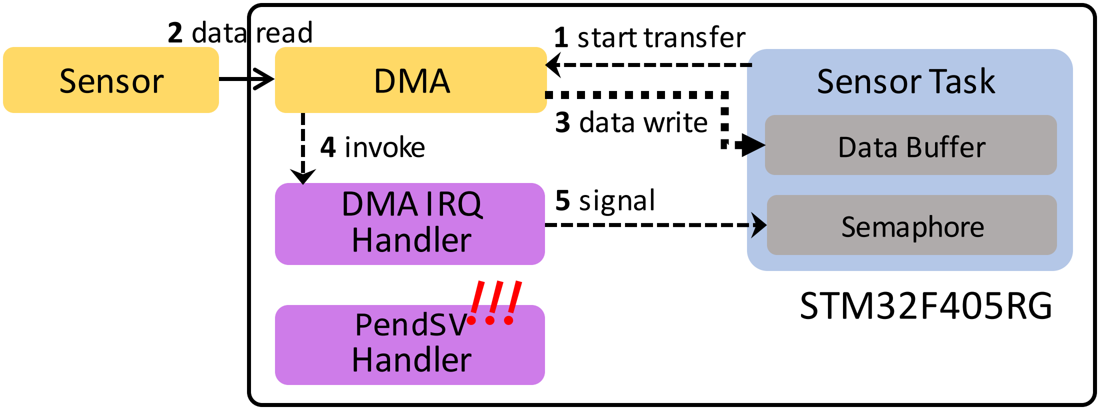

Flight control application overview

# Can Sustain Panics from Task & IRQ Handler

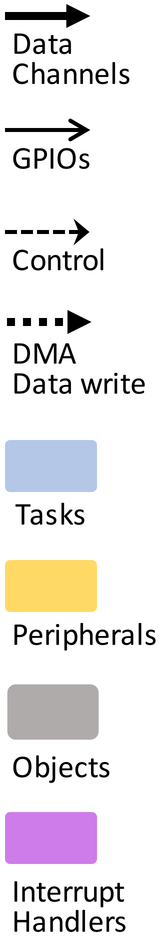
- Put `panic!()` in task's code
- Put `panic!()` in `PendSV` handler




Flight control application overview




Sensor task zoom in



# Without Panic

# With Panic

# Price for unwinding

- 2.6% more CPU usage after enabling unwinding
  - Precluded compiler optimizations
- 26.0% storage overhead
  - Unwinder logic, landing pads, exception table

# Conclusions

- Panic recovery via unwinding is feasible on embedded systems.
- Soft real-time constraint can be met.
- Rust reduces unwinder complexity.
- Rust facilitates concurrent task restarts.
- 2.6% CPU overhead, 26.0% storage overhead when flying a drone.