



Asynchronous Development

Lecture 5



Asynchronous Development

- Concurrency
- Asynchronous Executor
- Future s
- Communication between tasks



Concurrency

Preemptive and Cooperative



Bibliography

for this section

Brad Solomon, *Async IO in Python: A Complete Walkthrough*



Preemptive Concurrency

- MCUs are usually *single core*^[1]
- Tasks in parallel require an OS^[2]
- Tasks can be suspended at any time
- **Switching** the task is **expensive**
- Tasks that do a lot of I/O which makes the **switching time longer than the actual processing time**

1. RP2040 is a dual core MCU, we use only one core ↵
2. Running in an ISR is not considered a normal task ↵





Cooperative Concurrency

- tasks cannot be interrupted^[1]
- hardware** works in an **asynchronous** way
- tasks cooperate
 - give up the MCU for other tasks to use it while they wait for hardware
- there is **no need for an OS**, everything is done in **one single flow**
- no penalty** for saving and restoring the state

1. except for ISR ↔





Asynchronous Executor

of Embassy



Bibliography

for this section

Embassy Documentation, *Embassy executor*



Tasks

- `#[embassy_executor::main]`
 - starts the Embassy scheduler
 - defines the `main` task
- `#[embassy_executor::task]` - defines a new task
 - `pool_size` - is *optional* and defines how many identical tasks can be spawned
- the `main` task
 - initializes the `led`
 - spawns the `led_blink` task (adds to the scheduler)
 - uses `.await` to give up the MCU while waiting for the button

```
1  #[embassy_executor::task(pool_size = 2)]
2  async fn led_blink(mut led:Output<'static, PIN_X>) {
3      loop {
4          led.toggle();
5          Timer::after_secs(1).await;
6      }
7  }
8
9  #[embassy_executor::main]
10 async fn main(spawner: Spawner) {
11     // ...
12
13     // init led
14     spawner.spawn(led_blink(led)).unwrap();
15     info!("task started");
16
17     // init button
18     loop {
19         button.wait_for_rising_edge().await;
20         info!("button pressed");
21     }
22 }
```



Tasks can stop the executor

- unless awaited, `async` functions are not executed
- tasks have to use `.await` in loops, otherwise they block the scheduler

```
1  #[embassy_executor::task]
2  async fn led_blink(mut led:Output<'static, PIN_X>) {
3      loop {
4          led.toggle();
5          // this does not execute anything
6          Timer::after_secs(1);
7          // infinite loop without `.await`
8          // that never gives up the MCU
9      }
10 }
11
12 #[embassy_executor::main]
13 async fn main(spawner: Spawner) {
14     // ..
15     loop {
16         button.wait_for_rising_edge().await;
17         info!("button pressed");
18     }
19 }
```



How it works



- sleep when **all tasks wait for events**
- after an ISR is executed
 - if waiting for events, ask every task if it can execute (if the IRQ was what the task was `.await` ing for)
 - if a task is executing, continue the task until it `.await` s
- if a task never `.await` s, the executor does not run and never executes another task



Priority Tasks



```
#[interrupt]
unsafe fn SWI_IRQ_1() {
    EXECUTOR_HIGH.on_interrupt()
}
#[interrupt]
unsafe fn SWI_IRQ_0() {
    EXECUTOR_MED.on_interrupt()
}
```

```
1 static EXECUTOR_HIGH: InterruptExecutor = InterruptExecutor::new();
2 static EXECUTOR_MED: InterruptExecutor = InterruptExecutor::new();
3 static EXECUTOR_LOW: StaticCell<Executor> = StaticCell::new();
4
5 #[entry]
6 fn main() -> ! {
7     // High-priority executor: SWI_IRQ_1, priority level 2
8     interrupt::SWI_IRQ_1.set_priority(Priority::P2);
9     let spawner = EXECUTOR_HIGH.start(interrupt::SWI_IRQ_1);
10    spawner.spawn(run_high()).unwrap();
11
12    // Medium-priority executor: SWI_IRQ_0, priority level 3
13    interrupt::SWI_IRQ_0.set_priority(Priority::P3);
14    let spawner = EXECUTOR_MED.start(interrupt::SWI_IRQ_0);
15    spawner.spawn(run_med()).unwrap();
16
17    // Low priority executor: runs in thread mode, using WFE/SEV
18    let executor = EXECUTOR_LOW.init(Executor::new());
19    executor.run(|spawner| {
20        unwrap!(spawner.spawn(run_low()));
21    });
22 }
```

priority executors run in ISRs, lower priority tasks are interrupted



The `Future` type

a.k.a `Promise` in other languages



Bibliography

for this section

Bert Peters, *How does async Rust work*



Future

```
enum Poll<T> {  
    Pending,  
    Ready(T),  
}  
  
trait Future {  
    type Output;  
    fn poll(&mut self) -> Poll<Self::Output>;  
}
```

```
fn execute<F>(mut f: F) -> F::Output  
where  
    F: Future  
{  
    loop {  
        match f.poll() {  
            Poll::Pending => wait_for_event(),  
            Poll::Ready(value) => break value  
        }  
    }  
}
```





Implementing a Future

```
1  enum SleepStatus {
2      SetAlarm,
3      WaitForAlarm,
4  }
5
6  struct Sleep {
7      timeout: usize,
8      status: SleepStatus,
9  }
10
11  impl Sleep {
12      pub fn new(timeout: usize) -> Sleep {
13          Sleep {
14              timeout,
15              status: SleepStatus::SetAlarm,
16          }
17      }
18  }
```

```
impl Future for Sleep {
    type Output = ();

    fn poll(&mut self) -> Poll<Self::Output> {
        match self.status {
            SleepStatus::SetAlarm => {
                ALARM.set_alarm(self.timeout);
                self.status = SleepStatus::WaitForAlarm;
                Poll::Pending
            }
            SleepStatus::WaitForAlarm => {
                if ALARM.expired() {
                    Poll::Ready(())
                } else {
                    Poll::Pending
                }
            }
        }
    }
}
```




Executing Sleep

```
impl Future for Sleep {  
    type Output = ();  
  
    fn poll(&mut self) -> Poll<Self::Output> {  
        match self.status {  
            SleepStatus::SetAlarm => {  
                ALARM.set_alarm(self.timeout);  
                self.status = SleepStatus::WaitForAlarm;  
                Poll::Pending  
            }  
            SleepStatus::WaitForAlarm => {  
                if ALARM.expired() {  
                    Poll::Ready(())  
                } else {  
                    Poll::Pending  
                }  
            }  
        }  
    }  
}
```





Async Rust

```
async fn blink(mut led: Output<'static, PIN_X>) {  
    led.on();  
    Timer::after_secs(1).await;  
    led.off();  
}
```

Rust rewrites

```
struct Blink {  
    // status  
    status: BlinkStatus,  
    // local variables  
    led: Output<'static, PIN_X>,  
    timer: Option<impl Future>,  
}  
  
impl Blink {  
    pub fn new(led: Output<'static, PIN_X>) -> Blink {  
        Blink { status: BlinkStatus::Part1, led, timer: None }  
    }  
}  
  
fn blink(led: Output<'static, PIN_X>) -> Blink {  
    Blink::new(led)  
}
```

```
impl Future for Blink {  
    type Output = ();  
    fn poll(&mut self) -> Poll<Self::Output> {  
        loop {  
            match self.status {  
                BlinkStatus::Part1 => {  
                    self.led.on();  
                    self.timer1 = Some(Timer::after_secs(1));  
                    self.status = BlinkStatus::Part2;  
                }  
                BlinkStatus::Part2 => {  
                    if self.timer.unwrap().poll() == Poll::Pending {  
                        return Poll::Pending;  
                    } else {  
                        self.status = BlinkStatus::Part3;  
                    }  
                }  
                BlinkStatus::Part3 => {  
                    self.led.off();  
                    return Poll::Ready(());  
                }  
            }  
        }  
    }  
}
```



Async Rust

- the Rust compiler rewrites `async` function into `Future`
- it does not know how to execute them
- executors are implemented into third party libraries

```
1  use engine::execute;
2
3  // Rust rewrites the function to a Future
4  async fn blink(mut led: Output<'static, PIN_X>) {
5      led.on();
6      Timer::after_secs(1).await;
7      led.off();
8  }
9
10 #[entry]
11 fn main() -> ! {
12     blink(); // this returns the Blink future, but does not execute it
13     blink().await; // does not work, as `main` is not an `async` function
14     execute(blink()); // this works, as `execute` executes the Blink future
15 }
```



Executor

```
1  static TASKS: [Option<impl Future>; N] = [None, N];
2
3  fn executor() {
4      loop {
5          // ask all tasks to continue if they have available data
6          for task in TASKS.iter_mut() {
7              if let Some(task) = task {
8                  if Poll::Ready(_) = task.poll() {
9                      *task = None
10                 }
11             }
12         }
13
14         // wait for interrupts
15         cortex_m::asm::wfi();
16     }
17 }
```

- this is a simplified version, `Option<impl Future>` does not work
- the executor is not able to use `TASKS` like this
- an efficient executor will not poll all the tasks, it uses a `waker` that tasks use to signal the executor



```
1 trait Future {
2     type Output;
3
4     fn poll(mut self: std::pin::Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
5 }
```

-
- The diagram illustrates the state transitions of a task scheduler. It features a horizontal bar at the top containing several states: 'Task' (with a red 'W' icon), 'Task' (with a red 'W' icon), 'Empty Task Slot', 'Empty Task Slot', an ellipsis, and 'Wait for Event'. Below this bar, an 'ISR' (Interrupt Service Routine) block is shown on the left, and an 'Execute' block is on the right. Solid black arrows indicate the flow: from an external input to the 'ISR', from 'ISR' to the first 'Task' state, from the first 'Task' state to the 'Execute' block, and from the 'Execute' block to the 'Wait for Event' state. Dashed red arrows show the return path: from the 'Wait for Event' state back to the 'Task' states, and from the 'Execute' block back to the 'Task' states.



Communication

between tasks



Bibliography

for this section

Omar Hiari, *Sharing Data Among Tasks in Rust Embassy: Synchronization Primitives*



Simultaneous Access

Rust forbids simultaneous writes access





Exclusive Access

we want to sequentially access the resource





Synchronization

safely share data between tasks

- `NoopMutex` - used for data shared between tasks within the **same executor**
- `CriticalSectionMutex` - used for data shared between multiple executors, ISRs and cores
- `ThreadModeMutex` - used for data shared between tasks within **low priority executors** (not running in **ISRs** mode) running on a **single core**



- ISRs are executed in parallel with tasks
- embassy allows registering priority executors, that run tasks in ISRs
- some MCUs have multiple cores



Blocking Mutex

no `.await` allowed while the mutex is held

```
1 use embassy_sync::blocking_mutex::Mutex;
2
3 struct Data { /* ... */ }
4
5 static SHARED_DATA: Mutex<ThreadModeRawMutex, RefCell<Data>> = Mutex::new(RefCell::new(Data::new(/* ... */)));
6
7 #[embassy_executor::task]
8 async fn task1() {
9     // Load value from global context, modify and store
10     SHARED_DATA.lock(|f| {
11         let data = f.borrow_mut();
12         // edit data
13         f.replace(data);
14     });
15 }
```



Async Mutex

`.await` is allowed while the Mutex is held, it will release the Mutex while `await` ing

```
1  use embassy_sync::mutex::Mutex;
2
3  struct Data { /* ... */ }
4
5  static SHARED: Mutex<ThreadModeRawMutex, Data> = Mutex::new(Data::new(/* ... */));
6
7  #[embassy_executor::task]
8  async fn task1() {
9      // Load value from global context, modify and store
10     {
11         let mut data = SHARED_DATA.lock().await;
12         // edit *data
13         Timer::after(Duration::from_millis(1000)).await;
14     }
15 }
```



Channels

send data from a task to another

Embassy provides four types of channels synchronized using `Mutex` s

Type	Description
<code>Channel</code>	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer.
<code>PriorityChannel</code>	A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer. Higher priority items are shifted to the front of the channel.
<code>Signal</code>	Signalling latest value to a single consumer.
<code>PubSubChannel</code>	A broadcast channel (publish-subscribe) channel. Each message is received by all consumers.



Channel and Signal

sends data from one task to another

Channel - A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer.

Signal - Signalling latest value to a single consumer.





PriorityChannel

sends data from one task to another with a priority

`PriorityChannel` - A Multiple Producer Multiple Consumer (MPMC) channel. Each message is only received by a single consumer. Higher priority items are shifted to the front of the channel.





PubSubChannel

sends data from one task to all receiver tasks

`PubSubChannel` - A broadcast channel (publish-subscribe) channel. Each message is received by all consumers.





Channel Example

```
1  enum LedState { On, Off }
2  static CHANNEL: Channel<ThreadModeRawMutex, LedState, 64> = Channel::new();
3
4  #[embassy_executor::main]
5  async fn main(spawner: Spawner) {
6      // init led
7      spawner.spawn(execute_led(CHANNEL.sender(), Duration::from_millis(500)));
8      loop {
9          match CHANNEL.receive().await {
10             LedState::On => led.on(),
11             LedState::Off => led.off()
12          }
13      }
14  }
15
16  #[embassy_executor::task]
17  async fn execute_led(control: Sender<'static, ThreadModeRawMutex, LedState, 64>, delay: Duration) {
18      let mut ticker = Ticker::every(delay);
19      loop {
20          control.send(LedState::On).await;
21          ticker.next().await;
22          control.send(LedState::Off).await;
23          ticker.next().await;
24      }
25  }
```



Conclusion

we discussed about

- Preemptive & Cooperative Concurrency
- Asynchronous Executor
- `Future` s and how Rust rewrites `async` function
- Communication between tasks