Using Devices on seL4

Stephen Sherratt | Research Engineer

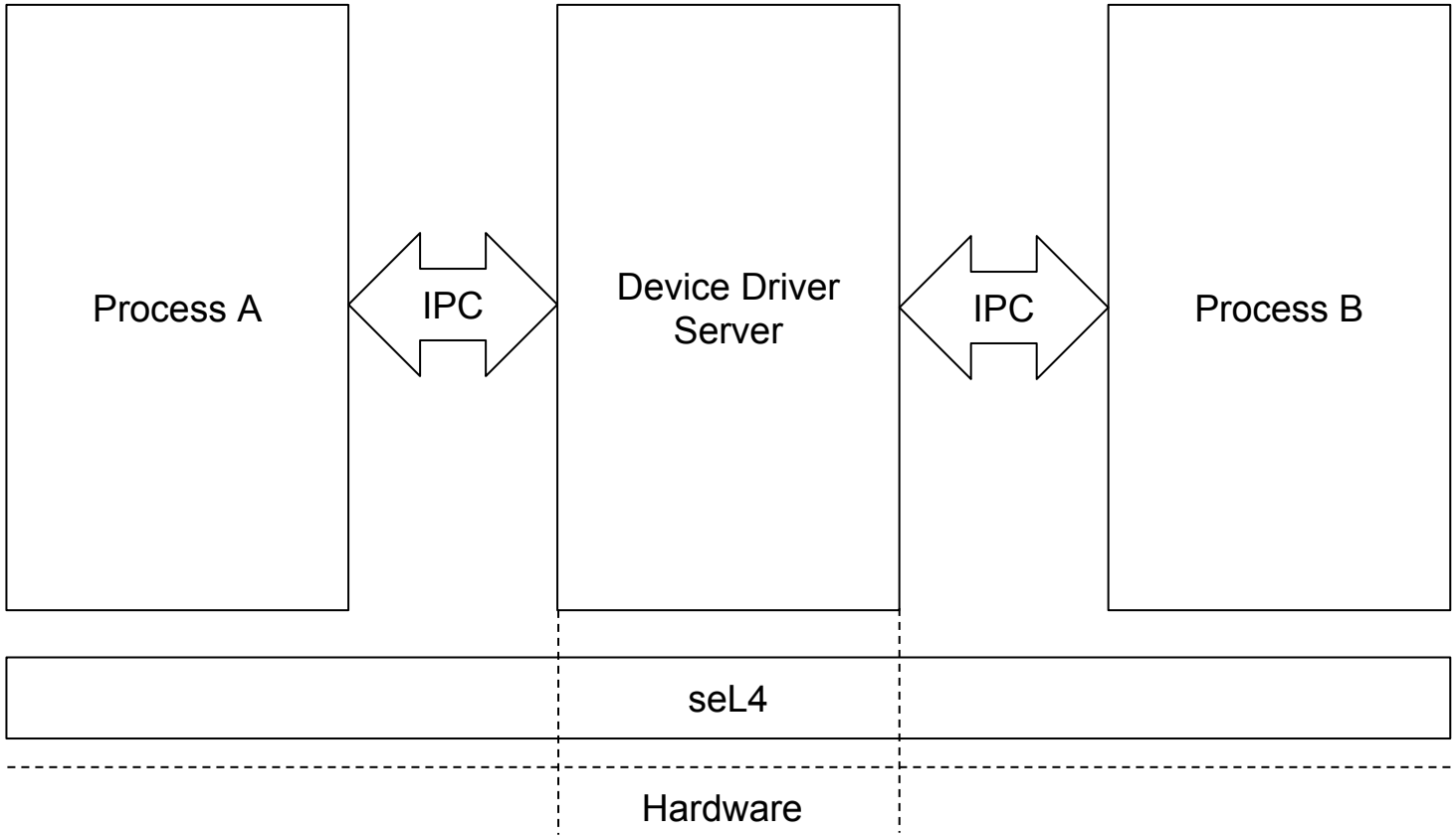
August 2016

www.data61.csiro.au

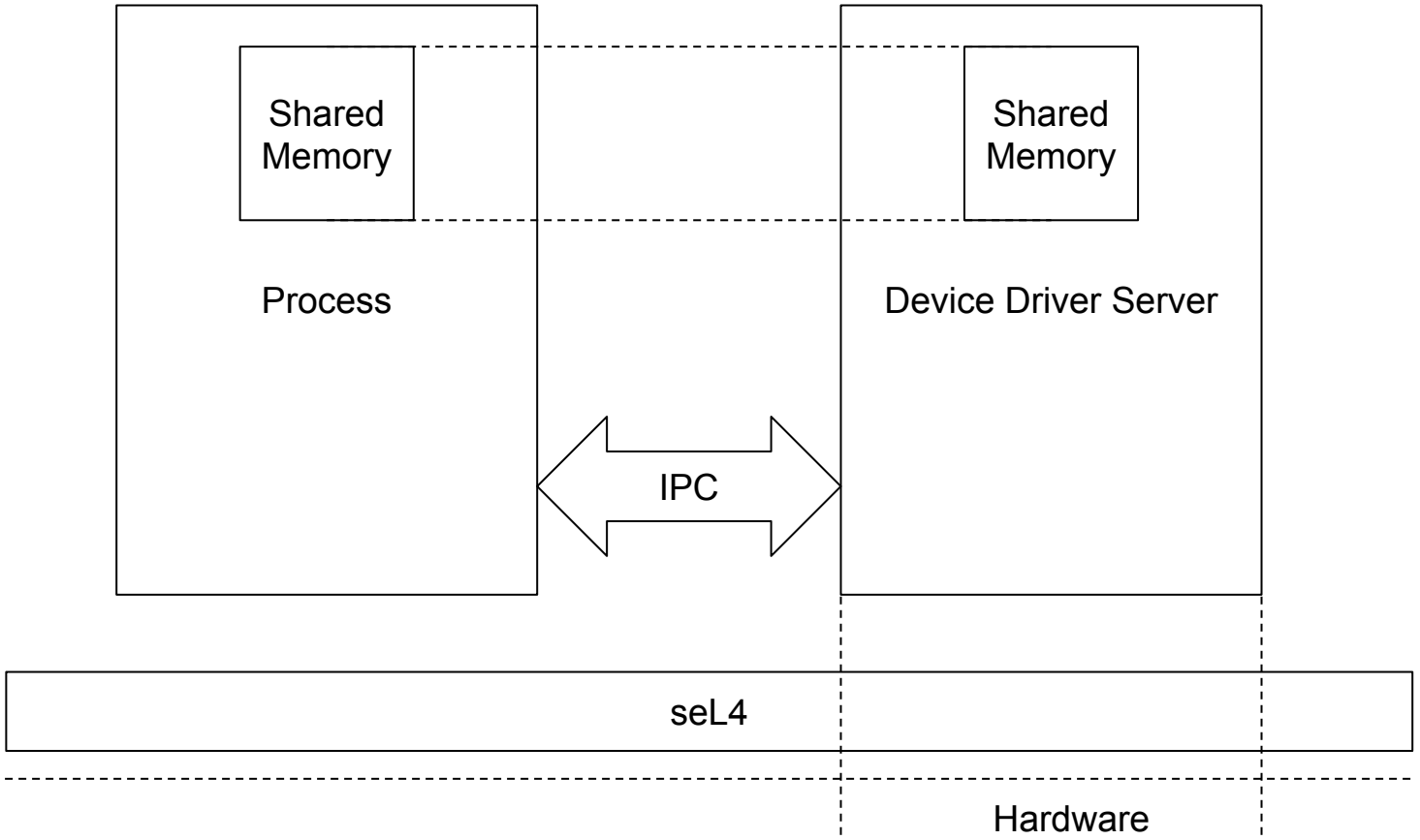
Overview

- Driver Model
- Interacting with Hardware
- Using Drivers
- Writing Drivers

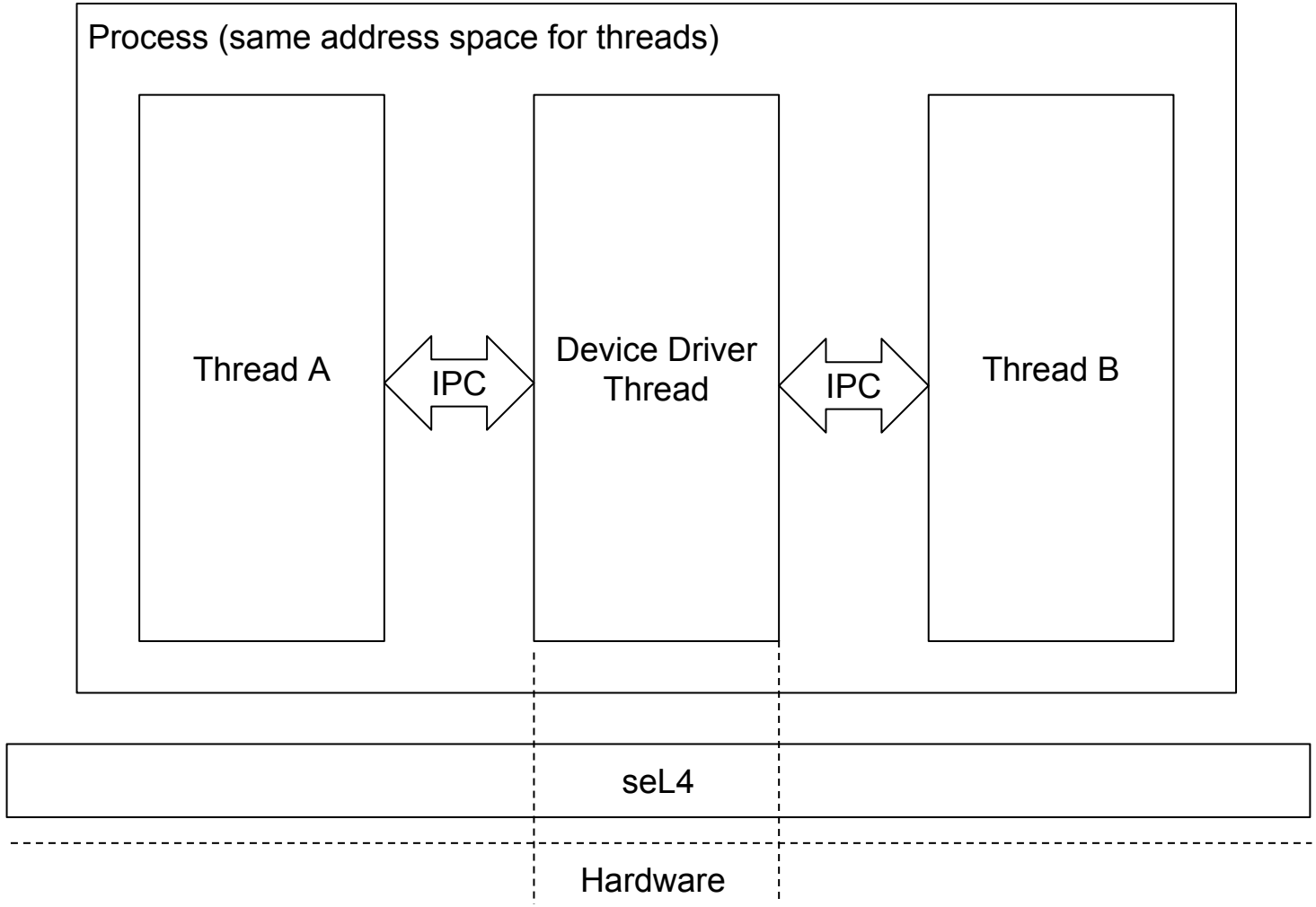
Driver Model



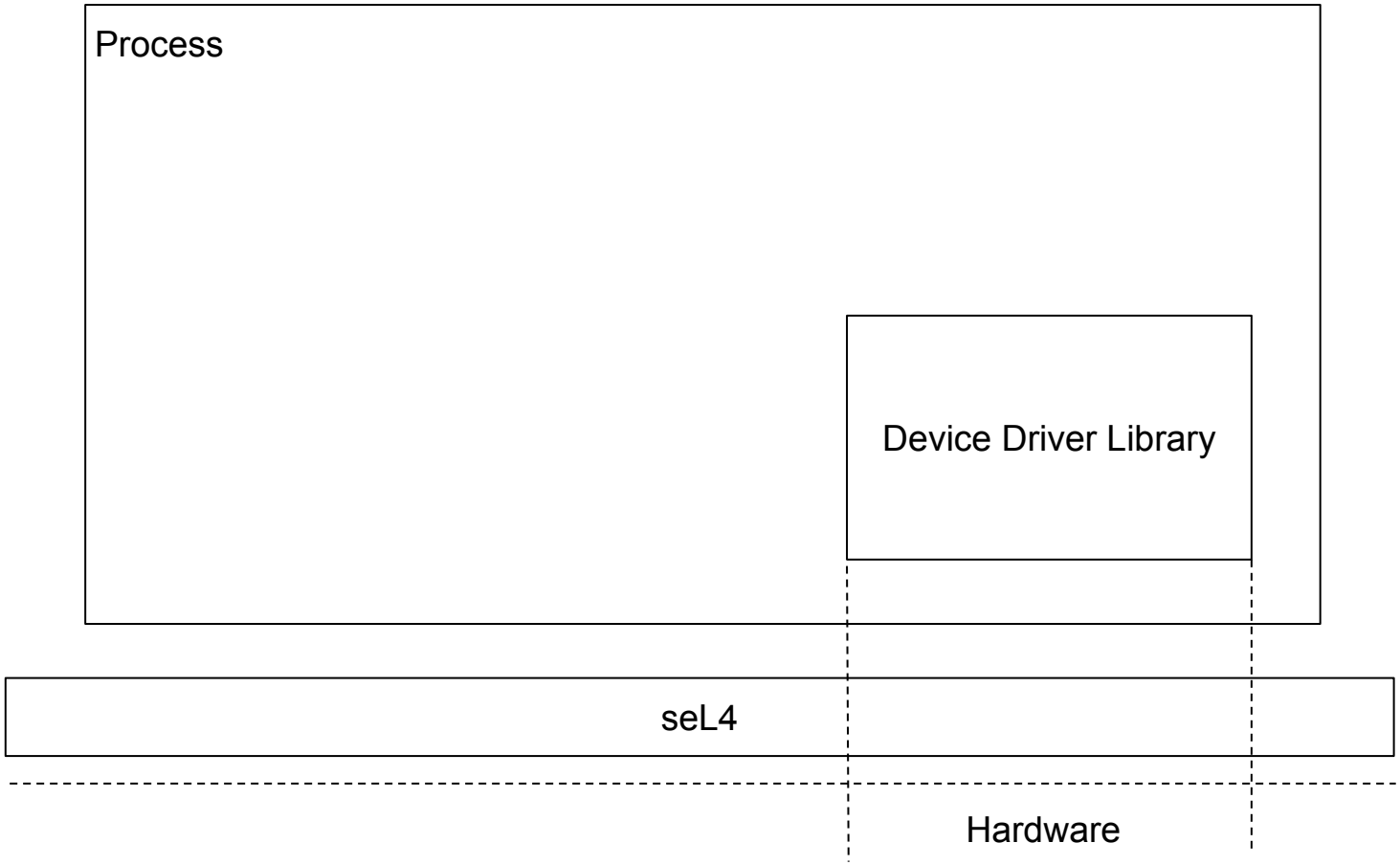
Driver Model



Driver Model



Driver Model



Driver Model

Driver Model

Summary

- Device drivers are at userlevel
- No policy for driver interfaces

Driver Model

Summary

- Device drivers are at userlevel
- No policy for driver interfaces

So how do we access hardware?

Interacting with Hardware

- Memory-Mapped IO
- Interrupts
- IO Ports (x86 only)

Interacting with Hardware

Memory-Mapped IO

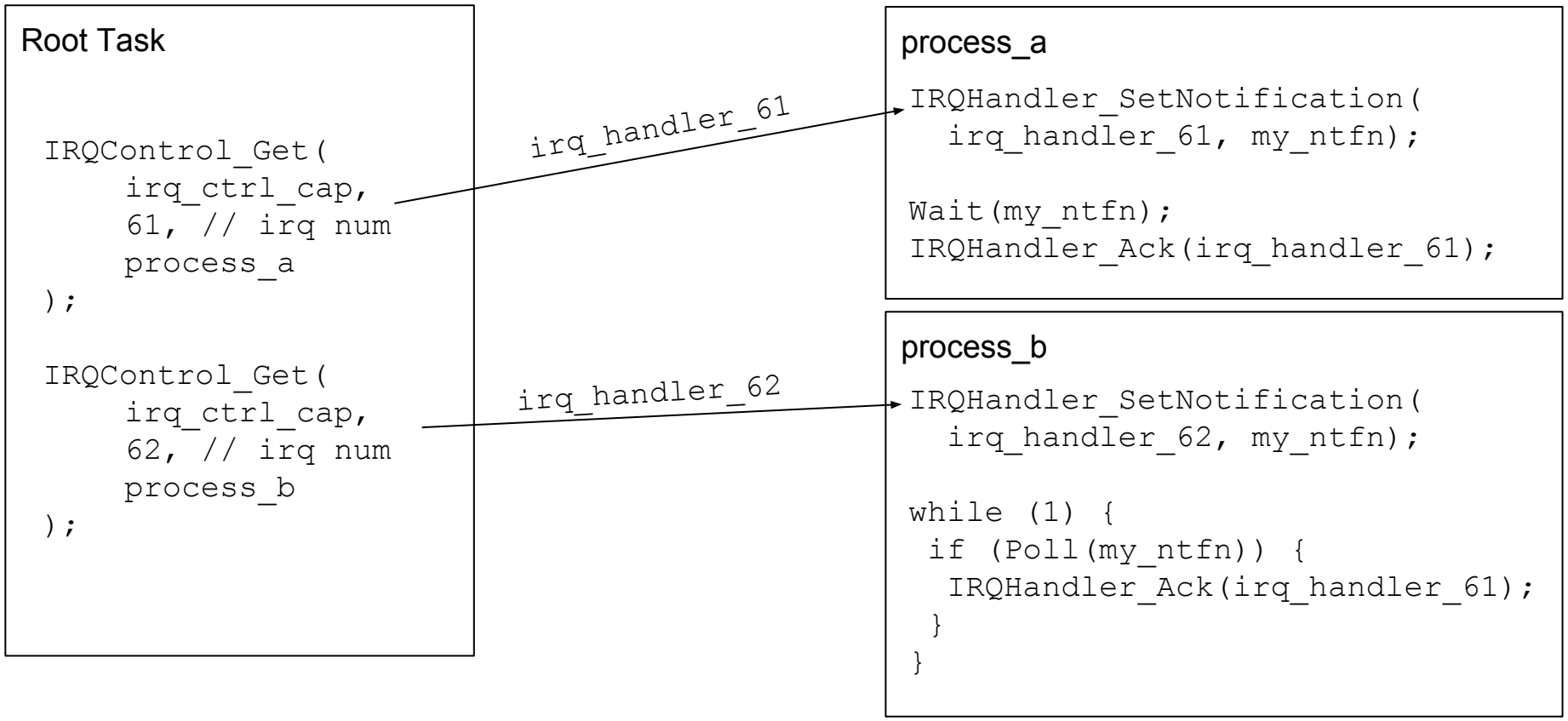
- Kernel creates frames for device regions during boot
- Root task gets caps to all device frames
- `deviceRegions` field in bootinfo contains (for each region):
  - Paddr
  - Size
  - Frame cap cptrs

- Map device frames in like any other frame

Interacting with Hardware

Interrupts

- Interrupts will result in signals on notifications
  - So you can `seL4_Wait` or `seL4_Poll` for interrupts



Interacting with Hardware

IO Ports (x86)

- Read and written by invoking io port caps
  - E.g. `seL4_IA32_IOPort_In8(io_port_cap, io_port_num)`
- IO port caps represent access to a contiguous range of IO ports
- IO port caps can be minted into new caps representing a subset of the original cap's range
- Root thread starts with a cap to all IO ports

Interacting with Hardware

Summary

- Device frames are just memory
- Interrupts are delivered as notifications
- Userland has access to IO ports on x86

Up next: Using drivers with seL4

Using Drivers

Libraries

- platsupport
  - seL4-agnostic device drivers
    - Expect memory to be mapped in for them
    - Provide functions for handling interrupts
- sel4platsupport
  - seL4-specific wrappers of platsupport drivers
    - Map in the device frames
    - Receives interrupt notifications and invokes handlers
- CAmkES
  - Generates code to interact with hardware on seL4
  - Abstracts devices as components

Using Drivers

platsupport

- Collections of drivers organised into groups
  - Arch: architecture (ie. arm/x86)
  - Plat: (platform) a specific soc/board (e.g. imx31, exynos4, exynos5)
  - Mach: (machine) a group of platforms with parts in common (e.g. exynos)
- Some devices have common interfaces across platforms
  - E.g. timers, character devices
- Others don't
- Most platforms have drivers for:
  - Serial port
  - Timer
  - Clock
  - i2c
  - gpio

Using Drivers

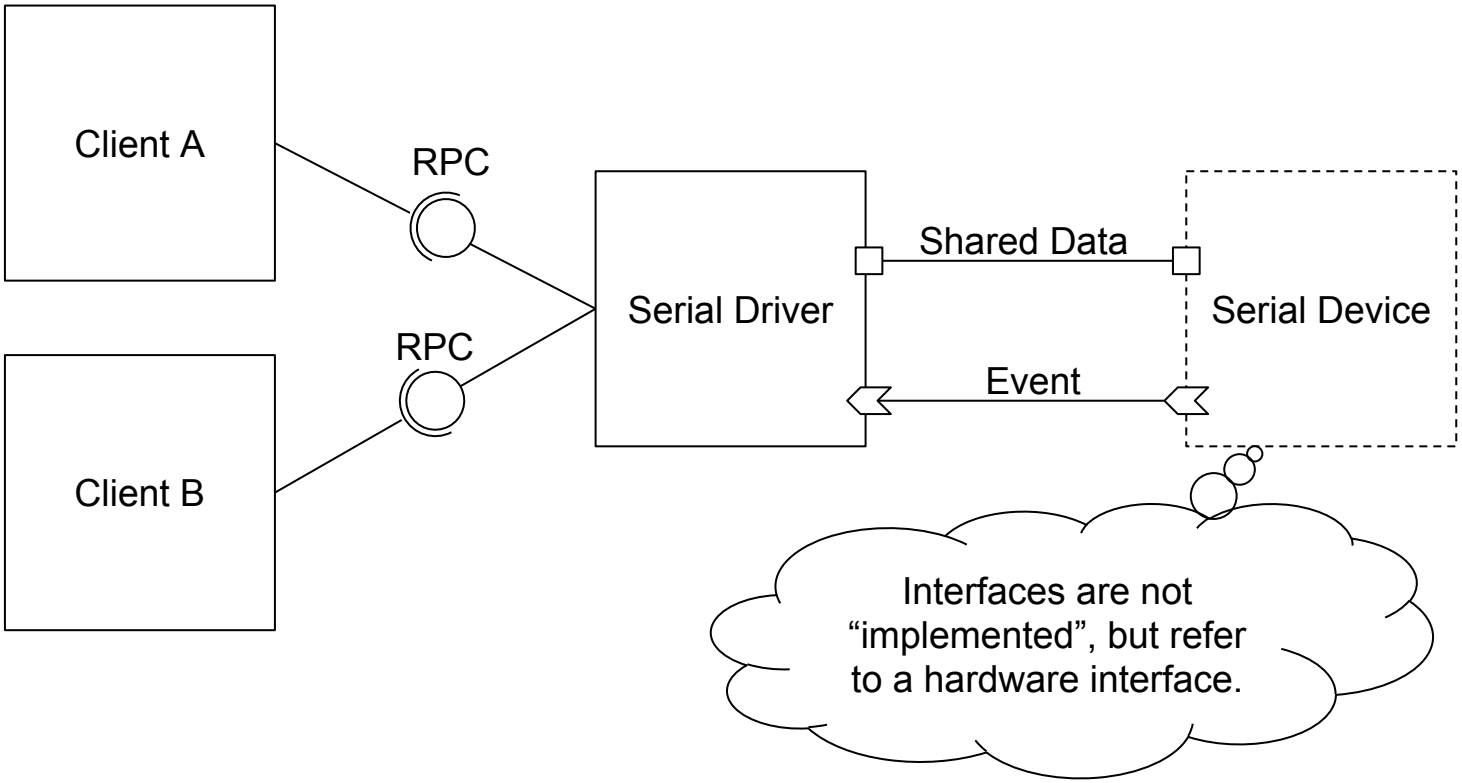
sel4platsupport

- sel4platsupport drivers need:
  - simple
    - device frame caps and irq control caps
  - vspace
    - mapping device frames
  - vka
    - creating kernel objects
    - managing caps
- Can request driver for default some devices on current platform
  - E.g. `sel4platsupport_get_default_timer`

Using Drivers

CAmkES

- Devices are treated like components



Using Drivers

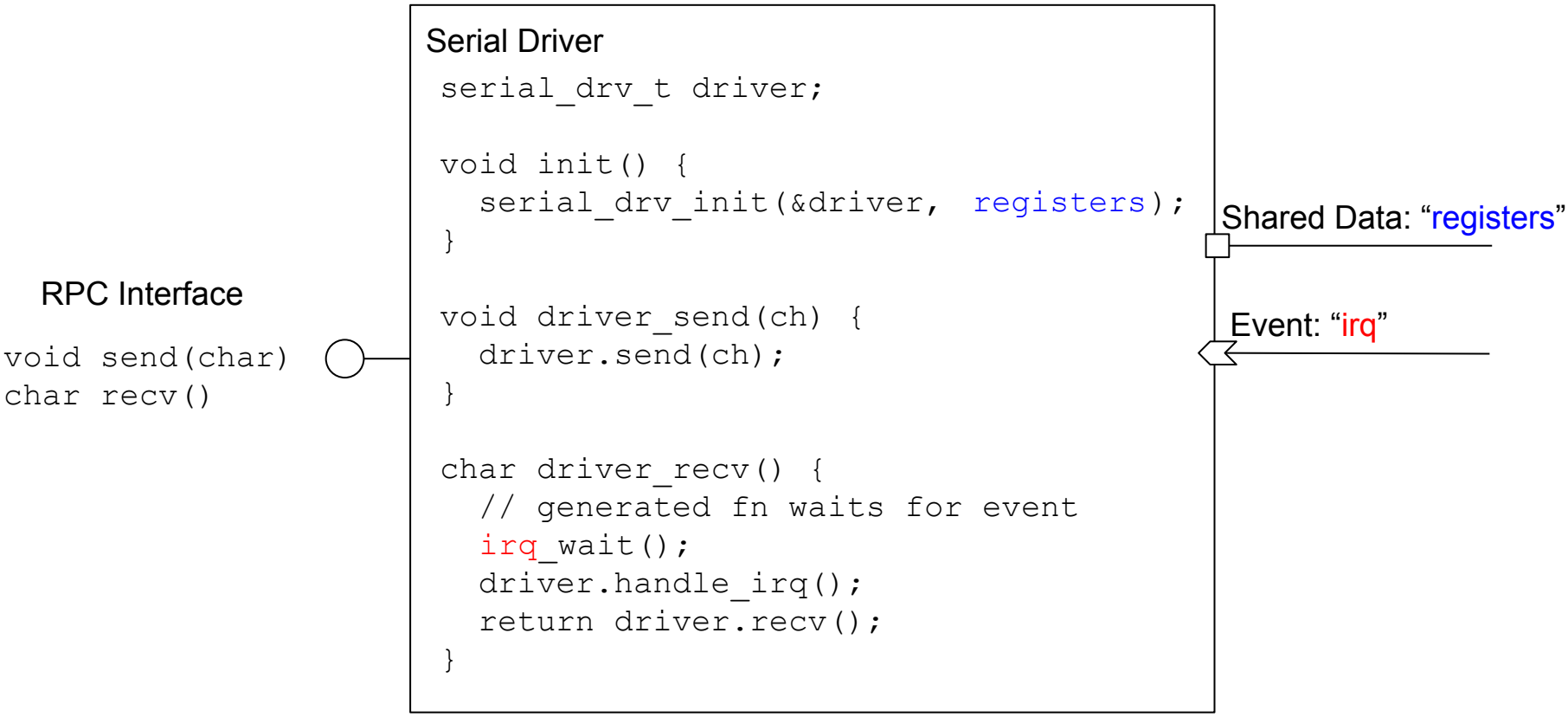
Accessing Hardware in CAmkES

- Memory-Mapped IO
  - Dataports (shared memory)
- Interrupts
  - Events
- IO Ports (x86)
  - RPC

Drivers talk to devices like any other components!

Using Drivers

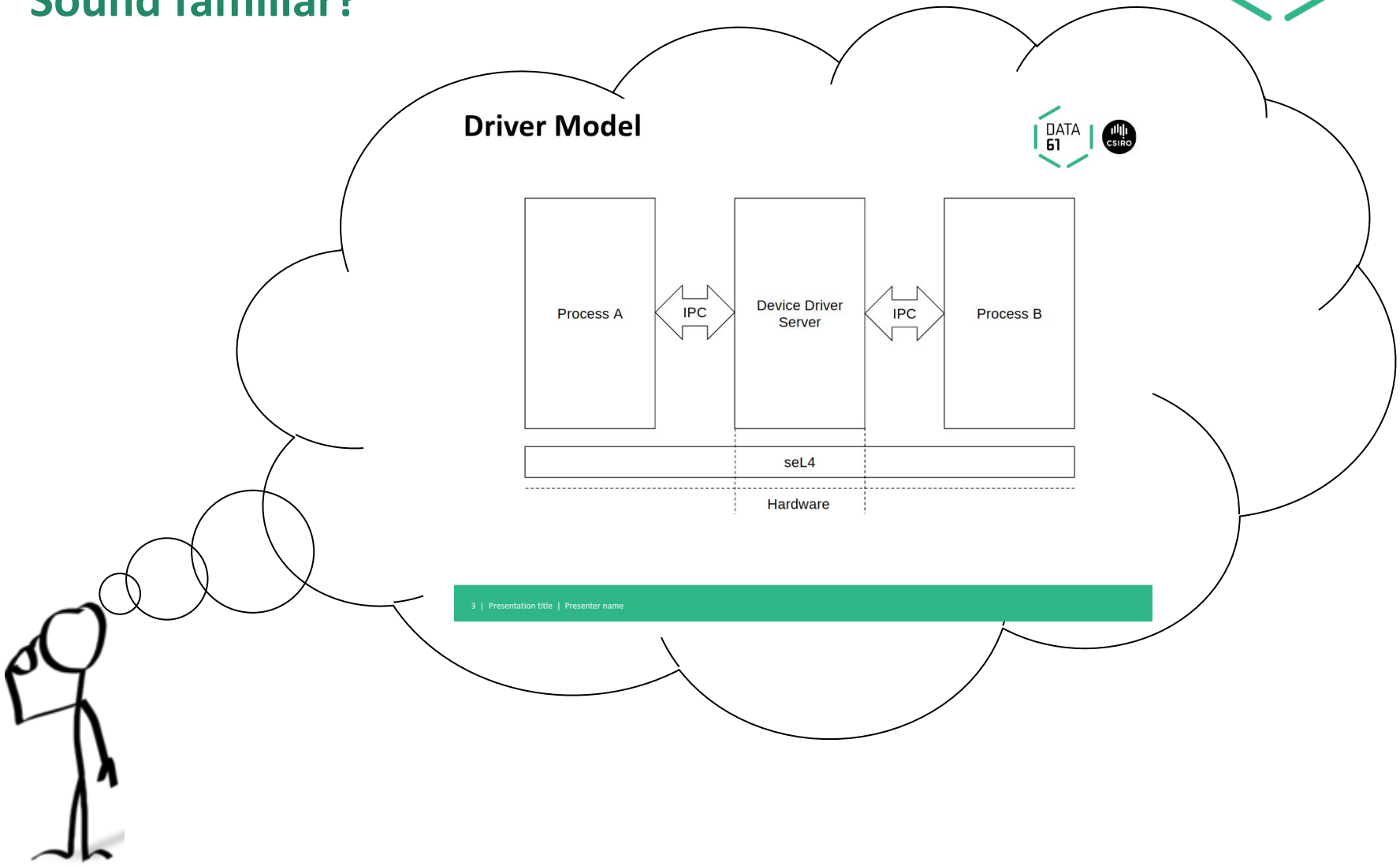
Typical Driver Usage



Actual driver implementation lives in platsupport!

Using Drivers

Sound familiar?



Using Drivers

Summary

- Libraries
  - platsupport: seL4 agnostic drivers
  - sel4platsupport: sel4-specific platsupport wrappers
- On bare seL4
  - Provide sel4platsupport drivers with required interfaces
- On CAmkES
  - Devices look like components
  - Driver components are typically wrappers around platsupport drivers

Using Drivers

Exercise

- Initialising/using a platsupport timer driver
  - seL4 tutorials: hello-timer
  - CAmkES tutorials: hello-camkes-timer

Writing Drivers

- A simple driver for an imaginary timer
  - Function to initialise device
  - Functions to perform operations with device]
    - E.g. read/write to a serial port, start/stop/read a timer
  - Functions to handle external events (ie. interrupts)

```
struct tmr_drv {
  void* vaddr;
  void (*handle_irq)(tmr_drv* t);
};

init_timer(tmr_drv* t, void* vaddr)
{
  t->vaddr = vaddr;
  t->handle_irq = my_timer_handle_irq;
}
```

```
my_timer_handle_irq(tmr_drv *t) {
  // ack the interrupt
  *(t->vaddr + IRQ_REG_OFFSET) = 0;
}

start_timer(tmr_drv *t) {
  *(t->vaddr + CTRL_REG_OFFSET) |=
    (1 << START_TIMER_BIT);
}

read_timer_counter(tmr_drv *t) {
  return *(t->vaddr + COUNT_REG_OFFSET);
}
```

Writing Drivers

- What resources does this driver need?
  - Someone needs to map the timer device frame into our address space
  - Someone needs to call the “handle_irq” function pointer when a timer interrupt arrives

```
struct tmr_drv {
  void* vaddr;
  void (*handle_irq)(tmr_drv* t);
};

init_timer(tmr_drv* t, void* vaddr)
{
  t->vaddr = vaddr;
  t->handle_irq = my_timer_handle_irq;
}
```

```
my_timer_handle_irq(tmr_drv *t) {
  // ack the interrupt
  *(t->vaddr + IRQ_REG_OFFSET) = 0;
}

start_timer(tmr_drv *t) {
  *(t->vaddr + CTRL_REG_OFFSET) |=
    (1 << START_TIMER_BIT);
}

read_timer_counter(tmr_drv *t) {
  return *(t->vaddr + COUNT_REG_OFFSET);
}
```

Writing Drivers

On bare seL4

- Assuming we have:
  - Simple
    - Timer device frame cap
    - Irq handler cap for timer interrupts
  - Vka
  - Vspace

Writing Drivers

On bare seL4

```
// Mapping in the timer device frame

timer_frame_cap = simple_get_frame_cap(&simple, TIMER_PADDR);
timer_vaddr = vspace_map_pages(
                  &vspace,
                  &timer_frame_cap,   // frame caps
                  NULL,               // cookies
                  seL4_CanWrite | seL4_CanRead,    // rights
                  1,                  // num pages
                  12,                 // size bits (4k page)
                  0);                 // cacheable

timer_drv t;
init_timer(&t, timer_vaddr);
```

Writing Drivers

On bare seL4

```
// Making interrupt notification

// allocate (retype) notification object
Vka_object_t irq_notification;
vka_alloc_notification(&vka, &irq_notification);

// allocate a cspace slot for irq handler cap
cspacepath_t irq_handler_path;
vka_cspace_alloc_path(&vka, &irq_handler_path);

// get the irq handler cap
simple_get_IRQ_control(&simple, TIMER_IRQ, irq_handler_path);

// associate the irq handler with the notification
seL4_IRQHandler_SetNotification(
  irq_handler_path.capPtr,
  irq_notification.cptr);
```

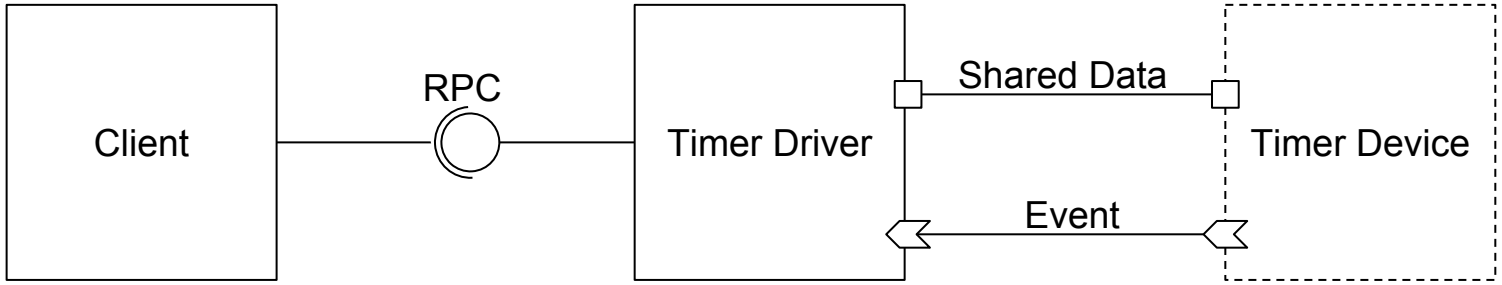
Writing Drivers

On bare seL4

```
// Invoking callback on interrupt

while (true) {

  // block until interrupt arrives
  seL4_Wait(irq_notification.cptr);

  // invoke callback
  t.handle_irq(&t);
}
```

Writing Drivers

On CAmkES




```
struct tmr_drv {
  void* vaddr;
  void (*handle_irq)(tmr_drv* t);
};

init_timer(tmr_drv* t, void* vaddr)
{
  t->vaddr = vaddr;
  t->handle_irq = my_timer_handle_irq;
}
```

```
my_timer_handle_irq(tmr_drv *t) {
  // ack the interrupt
  *(t->vaddr + IRQ_REG_OFFSET) = 0;
}

start_timer(tmr_drv *t) {
  *(t->vaddr + CTRL_REG_OFFSET) |=
    (1 << START_TIMER_BIT);
}

read_timer_counter(tmr_drv *t) {
  return *(t->vaddr + COUNT_REG_OFFSET);
}
```

Writing Drivers

On CAmkES

```
component Timer {
  hardware;
  dataport Buf regs;
  emits Irq irq;
}
component TimerDriver {
  dataport Buf registers;
  consumes Irq interrupt;
  provides TimerInterface iface;
}
component Client {
  control;
  uses TimerInterface tmr_iface;
}
procedure TimerInterface {
  void start();
  int read();
}
```

```
assembly {
  composition {
    component Timer timer_device;
    component TimerDriver driver;
    component Client client;

    connection seL4HardwareMMIO regs_con(
        from driver.registers,
        to timer_device.regs);
    connection seL4HardwareInterrupt irq_con(
        from timer_device.irq,
        to driver.interrupt);
    connection seL4RPC client_con(
        from client.tmr_iface, to driver.iface);
  } configuration {
    timer_device.regs_attributes =
        "0x12340000:0x1000";
    timer_device.irq_attributes = 42;
  }
}
```

Writing Drivers

On CAmkES

```
// client.c

int run(void) {
  tmr_iface_start();

  while (true) {
    int time = tmr_iface_read();
    printf("%d\n", time);
  }
}
```

Imported from glue code
Exported to glue code

```
// timer_driver.c

tmr_drv timer_driver;

void handle_irq(void) {
  timer_driver.handle_irq(&timer_driver);
  interrupt_reg_callback(handle_irq);
}

void iface_init(void) {
  init_timer(&timer_driver, registers);
  interrupt_reg_callback(handle_irq);
}

void iface_start(void) {
  start_timer(&timer_driver);
}

int iface_read(void) {
  return read_timer_counter(&timer_driver);
}
```

Writing Drivers

Summary

- Drivers broken into 2 parts
  - Interaction with device
    - Agnostic of environment
  - Allocation of system resources
    - Depends on environment
    - CAmkES and sel4 libraries have different approaches

Writing Drivers

Final Exercise

- You should have an emulator for the freescale i.mx31
- Reference manual at http://goo.gl/MlY9GE

- Make an app from scratch in CAmkES or bare seL4 (pick your favourite) that prints a message once per second
  - Without using platsupport/sel4platsupport

- Hints
  - Copy and modify a tutorial app
  - Use EPIT2 (frame paddr: 0x53f98000, irq no.: 27)
    - Why not EPIT1?