Embedded Systems Programming

Lecture 5

Verónica Gaspes
www2.hh.se/staff/vero

Center for Research on Embedded Systems
School of Information Science, Computer and Electrical Engineering

What we are looking at

```
struct Params params;
```

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}
```

We provide means for these two mains to execute concurrently! As if we had 2 CPUs!

What we are looking at

```
struct Params params;
```

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}
```

We provide means for these two mains to execute concurrently! As if we had 2 CPUs!

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**.

We should also provide a way of interleaving fragments of the threads.

We introduced a way of yielding execution so that another thread can take over. Let's see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

We should also provide a way of interleaving fragments of the threads.

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**

We introduced a way of yielding execution so that another thread can take over. Let's see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

We should also provide a way of interleaving fragments of the threads.

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**

We introduced a way of yielding execution so that another thread can take over. Let's see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

We introduced a way of yielding execution so that another thread can take over. Let's see how this function might be invoked.

What might a program look like?

```
main(){
  spawn(decoder_main);
  controller_main();
}
```

Notice that the function spawn takes a *function* as an argument!

The role of spawn is to provide one extra **Program Counter** and **Stack Pointer**

We should also provide a way of interleaving fragments of the threads.

We introduced a way of yielding execution so that another thread can take over. Let's see how this function might be invoked.

Calling `yield()`

Explicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c
jsr yield
ld c, r0
cmp #37, r0
ble label34
...
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling `yield()`

Explicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c
jsr yield
ld c, r0
cmp #37, r0
ble label34
...
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling `yield()`

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
     jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling `yield()`

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ⟵  Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
     jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling yield()

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c

  ← Interrupt on pin 3!

ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
    jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Calling `yield()`

Implicitly

```
ld a, r1
ld b, r2
add r, r2
st r2, c
```

← `Interrupt on pin 3!`

```
ld c, r0
cmp #37, r0
ble label34
...
```

```
vector_3:
    push r0-r2
    jsr yield
    pop r0-r2
    rti
```

```
yield:
    sub #2, sp
    ...
    mov #0, r0
    rts
```

Installing interrupt handlers

```
#include<avr/interrupt.h>

...
ISR(interrupt_name){
...
 // code as in a function body!
...
}
```

Preventing interrupts in avr-gcc

```
cli();
// ... code that must not be interrupted ...
sei();
```

Installing interrupt handlers

```
#include<avr/interrupt.h>

...
ISR(interrupt_name){
...
 // code as in a function body!
...
}
```

Preventing interrupts in avr-gcc

```
cli();
// ... code that must not be interrupted ...
sei();
```

Preventing interrupts

Why should we consider disabling interrupts? What parts of the program should be protected?

The critical section problem

What will happen if the `params` struct is read (by the controller) at the same time as it is written (by the decoder)?

I.e., what if the scheduler happens to insert some decoder instructions while some, but not all, of the controller's reads have been done?

This problem is central to concurrent programming where there is any ammount of sharing!

The critical section problem

What will happen if the `params` struct is read (by the controller) at the same time as it is written (by the decoder)?

I.e., what if the scheduler happens to insert some decoder instructions while some, but not all, of the controller's reads have been done?

This problem is central to concurrent programming where there is any ammount of sharing!

The critical section problem

What will happen if the `params` struct is read (by the controller) at the same time as it is written (by the decoder)?

I.e., what if the scheduler happens to insert some decoder instructions while some, but not all, of the controller's reads have been done?

This problem is central to concurrent programming where there is any ammount of sharing!

Our embedded system

```
struct Params p;
```

```
while(1){
   ...
   p.minDistance = e1;
   p.maxSpeed = e2;
}
```

```
while(1){
   local_minD = p.minDistance;
   local_maxS = p.maxSpeed;
   ...
}
```

Possible interleaving

```
p.minDistance = 1;
p.maxSpeed = 1;
                                  local_minD = 1;
p.minDistance = 200;
p.maxSpeed = 150;
                                  local_maxS = 150
```

Our embedded system

```
struct Params p;
```

```
while(1){
   ...
   p.minDistance = e1;
   p.maxSpeed = e2;
}
```

```
while(1){
   local_minD = p.minDistance;
   local_maxS = p.maxSpeed;
   ...
}
```

Possible interleaving

```
p.minDistance = 1;
p.maxSpeed = 1;
                                 local_minD = 1;
p.minDistance = 200;
p.maxSpeed = 150;
                                 local_maxS = 150
```

The classical solution

Apply an access protocol to the critical sections that ensures mutual exclusion

Require that all parties follow the protocol

Access protocols are realized by means of a shared datastructure known as a mutex or a lock.

The classical solution

Apply an access protocol to the critical sections that ensures mutual exclusion

Require that all parties follow the protocol

Access protocols are realized by means of a shared datastructure known as a mutex or a lock.

The classical solution

Apply an access protocol to the critical sections that ensures mutual exclusion

Require that all parties follow the protocol

Access protocols are realized by means of a shared datastructure known as a mutex or a lock.

Mutual exclusion

```
struct Params p;
mutex m;
```

```
while(1){
    ...
    lock(&m);
    p.minDistance = e1;
    p.maxSpeed = e2;
    unlock(&m);
}
```

```
while(1){
    lock(&m);
    local_minD = p.minDistance;
    local_maxS = p.maxSpeed;
    unlock(&m);
    ...
}
```

The datatype `mutex` and the operations `lock` and `unlock` are defined in the kernel: each mutex has a queue of threads that are not in the ready queue. The operations move threads to and from the ready queue!

What we have learned . . .

- We know how to read and write to I/O device registers
- We know how to run several computations in parallel by time-slicing the CPU
- We know how to protect critical sections by means of a mutex

But . . .

What we have learned . . .

- We know how to read and write to I/O device registers
- We know how to run several computations in parallel by time-slicing the CPU
- We know how to protect critical sections by means of a mutex

But . . .

What we have learned . . .

- We know how to read and write to I/O device registers
- We know how to run several computations in parallel by time-slicing the CPU
- We know how to protect critical sections by means of a mutex

But . . .

What we have learned ...

- We know how to read and write to I/O device registers
- We know how to run several computations in parallel by time-slicing the CPU
- We know how to protect critical sections by means of a mutex

But ...

What we have learned ...

- We know how to read and write to I/O device registers
- We know how to run several computations in parallel by time-slicing the CPU
- We know how to protect critical sections by means of a mutex

But ...

Still not satisfied!

```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
    struct Packet packet;
    while(1){
        radio_read(&packet);
        decode(&packet,&params);
    }
}
```

⟵ Time slicing ⟶

Each thread gets half of the CPU cycles, irrespective of whether it is waiting or computing!

Still not satisfied!

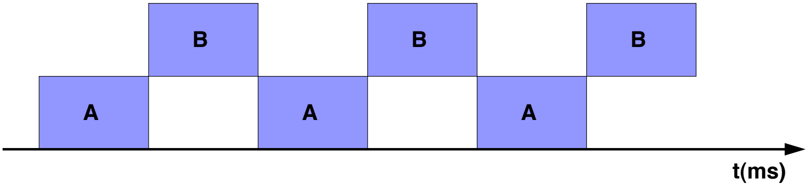
```
void controller_main() {
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main() {
   struct Packet packet;
   while(1){
      radio_read(&packet);
      decode(&packet,&params);
   }
}
```

⟵ Time slicing ⟶

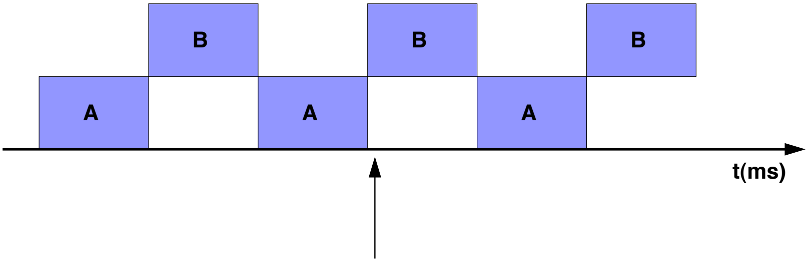
Each thread gets half of the CPU cycles, irrespective of whether it is waiting or computing!

Consequence 1



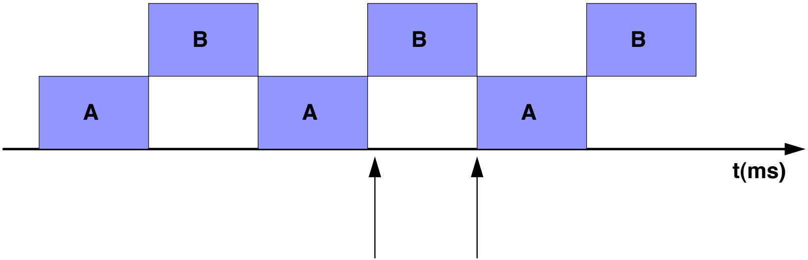
Say each thread gets **T**ms for execution, both waiting and computing!

Consequence 1



Say that an event that **A** is waiting for occurs now . . .

Consequence 1

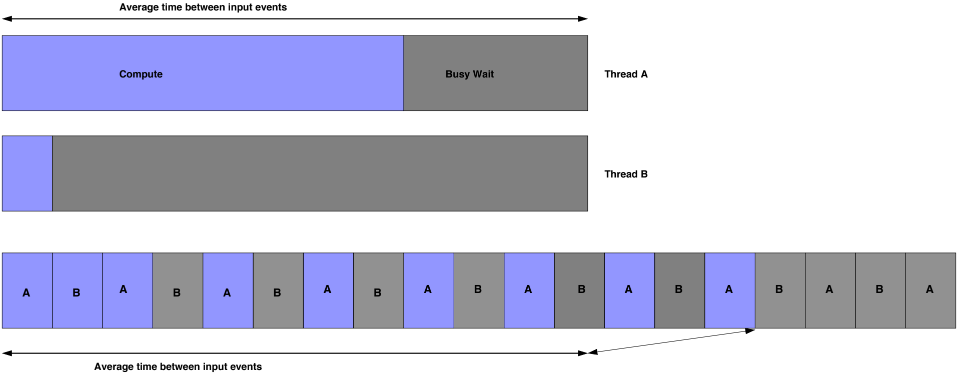


. . . it will not be noticed until now!

Consequence 1

With **N** threads in the system, each getting **T**ms for execution, a status change might have to wait up to **T*(N-1)**ms to be noticed!

Consequence 2



Busy waiting makes waiting indistinguishable from computing.
Thread A cannot keep up with event rate!

Busy waiting and Time slicing

Minus ...

1. Not a satisfactory technique for input synchronization if the system must meet real-time constraints!
2. Not a satisfactory technique for a system that is battery driven: 100% CPU cycle usage (100% power usage!).

Could we do otherwise?

An input synchronization technique that does not require the receiver of data to actively ask whether data has arrived.

Busy waiting and Time slicing

Minus . . .

1. Not a satisfactory technique for input synchronization if the system must meet real-time constraints!
2. Not a satisfactory technique for a system that is battery driven: 100% CPU cycle usage (100% power usage!).

Could we do otherwise?

An input synchronization technique that does not require the receiver of data to actively ask whether data has arrived.

Busy waiting and Time slicing

Minus . . .

1. Not a satisfactory technique for input synchronization if the system must meet real-time constraints!
2. Not a satisfactory technique for a system that is battery driven: 100% CPU cycle usage (100% power usage!).

Could we do otherwise?

An input synchronization technique that does not require the receiver of data to actively ask whether data has arrived.

Busy waiting and Time slicing

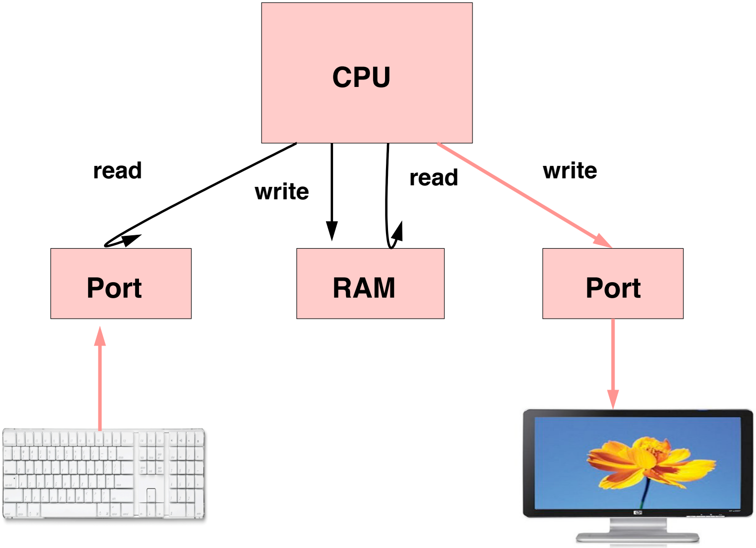
Minus . . .

1. Not a satisfactory technique for input synchronization if the system must meet real-time constraints!
2. Not a satisfactory technique for a system that is battery driven: 100% CPU cycle usage (100% power usage!).

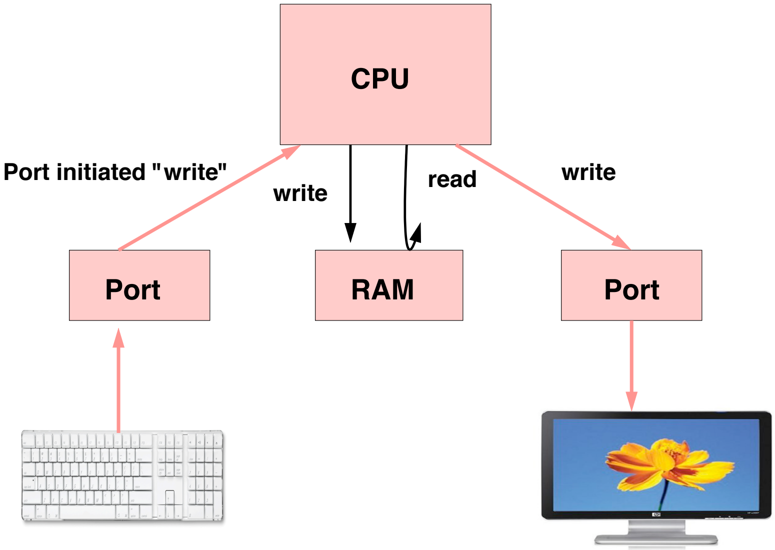
Could we do otherwise?

An input synchronization technique that does not require the receiver of data to actively ask whether data has arrived.

The naked computer – a mismatch



The naked computer – alternative



An analogy

You are expecting delivery of your latest web-shop purchase

Busy waiting

Go to the post-office again and again to check if the delivery has arrived.

Reacting to an interrupt

Receive a note in your mailbox that the goods can be picked up.

The CPU reacts to an interrupt signal by executing a designated ISR (interrupt service routine)

This has consequences for the way we structure programs. They become inside-out!

An analogy

You are expecting delivery of your latest web-shop purchase

Busy waiting

Go to the post-office again and again to check if the delivery has arrived.

Reacting to an interrupt

Receive a note in your mailbox that the goods can be picked up.

The CPU reacts to an interrupt signal by executing a designated ISR (interrupt service routine)

This has consequences for the way we structure programs. They become inside-out!

An analogy

You are expecting delivery of your latest web-shop purchase

Busy waiting

Go to the post-office again and again to check if the delivery has arrived.

Reacting to an interrupt

Receive a note in your mailbox that the goods can be picked up.

The CPU reacts to an interrupt signal by executing a designated ISR (interrupt service routine)

This has consequences for the way we structure programs. They become inside-out!

An analogy

You are expecting delivery of your latest web-shop purchase

Busy waiting

Go to the post-office again and again to check if the delivery has arrived.

Reacting to an interrupt

Receive a note in your mailbox that the goods can be picked up.

The CPU reacts to an interrupt signal by executing a designated ISR (interrupt service routine)

This has consequences for the way we structure programs. They become inside-out!

An analogy

You are expecting delivery of your latest web-shop purchase

Busy waiting

Go to the post-office again and again to check if the delivery has arrived.

Reacting to an interrupt

Receive a note in your mailbox that the goods can be picked up.

The CPU reacts to an interrupt signal by executing a designated ISR (interrupt service routine)

This has consequences for the way we structure programs. They become inside-out!

ISRs vs functions

Busy waiting

We defined functions like sonar_read that can be called in the program. The CPU decides when to call the function:

```
while(1){
    sonar_read();
    control();
}
```

Input detection = the exit from the busy waiting fragment (a function return)

Reacting

We define ISRs. These are not called from the program, but the code is executed when an interrupt occurs:

```
ISR(SIG_SONAR){
  control();
}
```

Input detection = invocation of the ISR (as if the hardware did a function call)

ISRs vs functions

Busy waiting

We defined functions like `sonar_read` that can be called in the program. The CPU decides when to call the function:

```
while(1){
   sonar_read();
   control();
}
```

Reacting

We define ISRs. These are not called from the program, but the code is executed when an interrupt occurs:

```
ISR(SIG_SONAR){
   control();
}
```

Input detection = the exit from the busy waiting fragment (a function return)

Input detection = invocation of the ISR (as if the hardware did a function call)

ISRs vs functions

Busy waiting

We defined functions like sonar_read that can be called in the program. The CPU decides when to call the function:

```
while(1){
   sonar_read();
   control();
}
```

Input detection $=$ the exit from the busy waiting fragment (a function return)

Reacting

We define ISRs. These are not called from the program, but the code is executed when an interrupt occurs:

```
ISR(SIG_SONAR){
   control();
}
```

Input detection $=$ invocation of the ISR (as if the hardware did a function call)

ISRs vs functions

Busy waiting

We defined functions like `sonar_read` that can be called in the program. The CPU decides when to call the function:

```
while(1){
   sonar_read();
   control();
}
```

Input detection = the exit from the busy waiting fragment (a function return)

Reacting

We define ISRs. These are not called from the program, but the code is executed when an interrupt occurs:

```
ISR(SIG_SONAR){
  control();
}
```

Input detection = invocation of the ISR (as if the hardware did a function call)

ISRs vs functions

Busy waiting

We defined functions like sonar_read that can be called in the program. The CPU decides when to call the function:

```
while(1){
   sonar_read();
   control();
}
```

Input detection = the exit from the busy waiting fragment (a function return)

Reacting

We define ISRs. These are not called from the program, but the code is executed when an interrupt occurs:

```
ISR(SIG_SONAR){
  control();
}
```

Input detection = invocation of the ISR (as if the hardware did a function call)

Two ways of organizing programs

CPU centric

One thread of control that runs from start to stop (or forever) reading and writing data as it goes.

Reacting CPU

A set of code fragments that constitute the reactions to recognized events.

The main part of the course from now on will focus on the reactive view.

Two ways of organizing programs

CPU centric

One thread of control that runs from start to stop (or forever) reading and writing data as it goes.

Reacting CPU

A set of code fragments that constitute the reactions to recognized events.

The main part of the course from now on will focus on the reactive view.

Two ways of organizing programs

CPU centric

One thread of control that runs from start to stop (or forever) reading and writing data as it goes.

Reacting CPU

A set of code fragments that constitute the reactions to recognized events.

The main part of the course from now on will focus on the reactive view.

Two ways of organizing programs

CPU centric

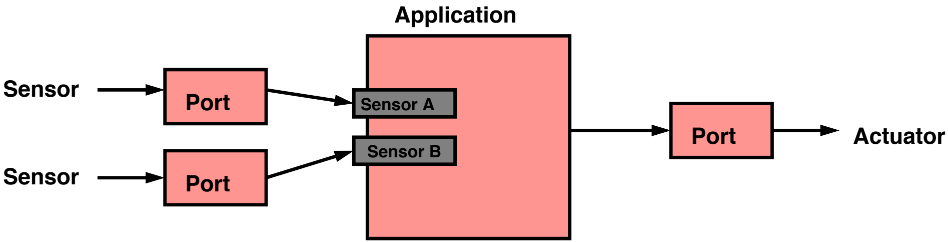
One thread of control that runs from start to stop (or forever) reading and writing data as it goes.

Reacting CPU

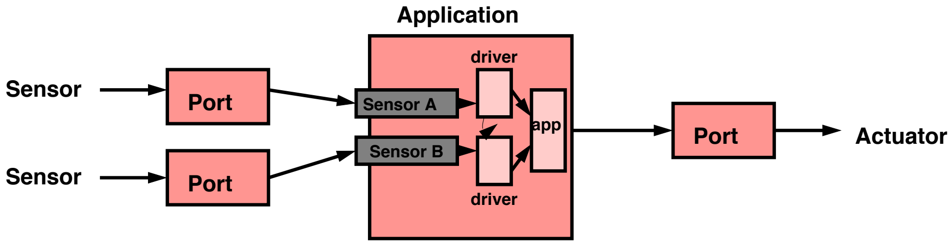
A set of code fragments that constitute the reactions to recognized events.

The main part of the course from now on will focus on the reactive view.

The reactive embedded system



The reactive embedded system



Reactive Objects

Boxes

Represent software or hardware reactive objects that:

- Maintain an internal state (variables, registers, etc)
- Provide a set of methods as reactions to external events (ISRs, etc)
- Simply rest between reactions!

Arrows

Represent event or signal or message flow between objects that can be either

- synchronous
- asynchronous

Reactive Objects

Boxes

Represent software or hardware reactive objects that:

- Maintain an internal state (variables, registers, etc)
- Provide a set of methods as reactions to external events (ISRs, etc)
- Simply rest between reactions!

Arrows

Represent event or signal or message flow between objects that can be either

- synchronous
- asynchronous

Reactive Objects

Boxes

Represent software or hardware reactive objects that:

- Maintain an internal state (variables, registers, etc)
- Provide a set of methods as reactions to external events (ISRs, etc)
- Simply rest between reactions!

Arrows

Represent event or signal or message flow between objects that can be either

- synchronous
- asynchronous

Reactive Objects

Boxes

Represent software or hardware reactive objects that:

- Maintain an internal state (variables, registers, etc)
- Provide a set of methods as reactions to external events (ISRs, etc)
- Simply rest between reactions!

Arrows

Represent event or signal or message flow between objects that can be either

- asynchronous 
- synchronous 

Hardware objects

Hardware devices are reactive objects

A black box that does nothing unless stimulated by external events.

Serial port - state

Internal registers

Serial port - stimuli

- Signal change
- Bit pattern received
- Clock pulse

Serial port - emissions

- Signal change
- Interrupt signal

Hardware objects

Hardware devices are reactive objects

A black box that does nothing unless stimulated by external events.

Serial port - state

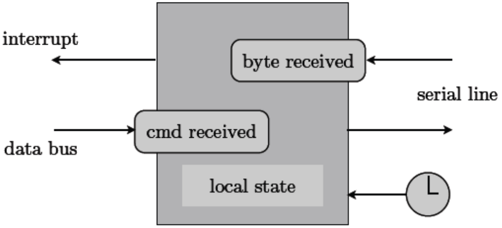
Internal registers

Serial port - stimuli

- Signal change
- Bit pattern received
- Clock pulse

Serial port - emissions

- Signal change
- Interrupt signal



Hardware objects

Hardware devices are reactive objects

A black box that does nothing unless stimulated by external events.

Serial port - state

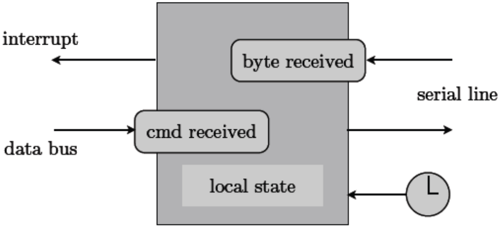
Internal registers

Serial port - stimuli

- Signal change
- Bit pattern received
- Clock pulse

Serial port - emissions

- Signal change
- Interrupt signal



Hardware objects

Hardware devices are reactive objects

A black box that does nothing unless stimulated by external events.

Serial port - state

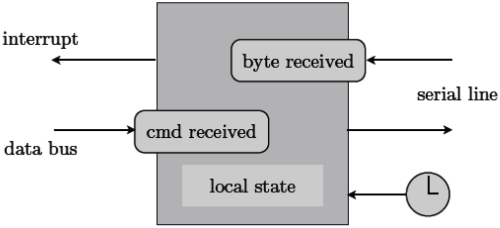
Internal registers

Serial port - stimuli

- Signal change
- Bit pattern received
- Clock pulse

Serial port - emissions

- Signal change
- Interrupt signal



Software objects

We would like to regard software objects as reactive objects ...

The Counter example

```
class Counter{
  int x;
  Counter(){x=0;}
  void inc(){x++;}
  int read(){return x;}
  void reset(){x=0;}
  void show(){
     System.out.print(x);}
}
```

Counter state

`x`

Counter - stimuli

`inc()`, `read()`, `reset()`, `show()`

Counter - emissions

`print()` to the object `System.out`

Software objects

We would like to regard software objects as reactive objects . . .

The Counter example

```
class Counter{
  int x;
  Counter(){x=0;}
  void inc(){x++;}
  int read(){return x;}
  void reset(){x=0;}
  void show(){
     System.out.print(x);}
}
```

Counter state

`x`

Counter - stimuli

`inc()`, `read()`, `reset()`, `show()`

Counter - emissions

`print()` to the object `System.out`

Software objects

We would like to regard software objects as reactive objects . . .

The Counter example

```
class Counter{
  int x;
  Counter(){x=0;}
  void inc(){x++;}
  int read(){return x;}
  void reset(){x=0;}
  void show(){
     System.out.print(x);}
}
```

Counter state

x

Counter - stimuli

inc(), read(), reset(), show()

Counter - emissions

print() to the object System.out

Software objects

We would like to regard software objects as reactive objects . . .

The Counter example

```
class Counter{
  int x;
  Counter(){x=0;}
  void inc(){x++;}
  int read(){return x;}
  void reset(){x=0;}
  void show(){
     System.out.print(x);}
}
```

Counter state

`x`

Counter - stimuli

`inc()`, `read()`, `reset()`, `show()`

Counter - emissions

`print()` to the object `System.out`

Software objects

We would like to regard software objects as reactive objects . . .

The Counter example

```
class Counter{
  int x;
  Counter(){x=0;}
  void inc(){x++;}
  int read(){return x;}
  void reset(){x=0;}
  void show(){
     System.out.print(x);}
}
```

Counter state

`x`

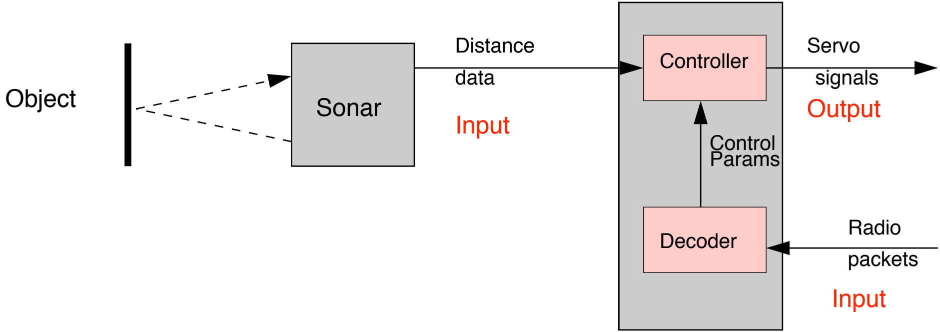
Counter - stimuli

`inc()`, `read()`, `reset()`, `show()`

Counter - emissions

`print()` to the object `System.out`

Back to our running example



All messages/events are asynchronous! Either generated by the CPU or by the sonar hw or by the communication hardware.

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Reactive Objects

Object Oriented Programming?

- Objects have local state
- Objects export methods
- Objects communicate by sending messages
- Objects rest between method invocation

Examples of intuitive objects

People, cars, molecules, . . .

Bonus

Principles and methodologies from OOP become applicable to embedded, event-driven and concurrent systems!

Java? C++?

The Counter example again

```
class Counter{
    int x;
    Counter(){x=0;}
    void inc(){x++;}
    int read(){return x;}
    void reset(){x=0;}
}
```

One thread

```
public static void main(){
  Counter c = new Counter();
  c.inc();
  System.out.println(c.read());
}
```

Creating a new object just creates a passive piece of storage! Not a thread of control!

Other threads that use the same counter are sharing the state!

Counting visitors to a park

Java? C++?

The Counter example again

```
class Counter{
    int x;
    Counter(){x=0;}
    void inc(){x++;}
    int read(){return x;}
    void reset(){x=0;}
}
```

One thread

```
public static void main(){
  Counter c = new Counter();
  c.inc();
  System.out.println(c.read());
}
```

Creating a new object just creates a passive piece of storage! Not a thread of control!

Other threads that use the same counter are sharing the state!

Counting visitors to a park

Java? C++?

The Counter example again

```
class Counter{
    int x;
    Counter(){x=0;}
    void inc(){x++;}
    int read(){return x;}
    void reset(){x=0;}
}
```

One thread

```
public static void main(){
  Counter c = new Counter();
  c.inc();
  System.out.println(c.read());
}
```

Creating a new object just creates a passive piece of storage! Not a thread of control!

Other threads that use the same counter are sharing the state!

Counting visitors to a park

Java? C++?

The Counter example again

```
class Counter{
    int x;
    Counter(){x=0;}
    void inc(){x++;}
    int read(){return x;}
    void reset(){x=0;}
}
```

One thread

```
public static void main(){
  Counter c = new Counter();
  c.inc();
  System.out.println(c.read());
}
```

Creating a new object just creates a passive piece of storage! Not a thread of control!

Other threads that use the same counter are sharing the state!

Counting visitors to a park

Java? C++?

The Counter example again

```
class Counter{
    int x;
    Counter(){x=0;}
    void inc(){x++;}
    int read(){return x;}
    void reset(){x=0;}
}
```

One thread

```
public static void main(){
  Counter c = new Counter();
  c.inc();
  System.out.println(c.read());
}
```

Creating a new object just creates a passive piece of storage! Not a thread of control!

Other threads that use the same counter are sharing the state!

Counting visitors to a park

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

OO and Concurrency

OO Languages:

- An object is a passive piece of global state
- A method is a function
- Sending a message is calling a function

Our model says

- An object is an independent process with local state
- A method is a process fragment
- Sending a message is interprocess communication

This is one of the reasons why we choose to build our own kernel supporting reactive objects and programming in C.

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!

Reactive objects in C

We will need to provide ways for

- Create reactive objects
- Declare protected local state
- Receive messages
  - synchronously
  - asynchronously
- Bridge the hardware/software divide (run ISRs)
- Schedule a system of reactive software objects.

This will be the contents of a kernel called TinyTimber that we will learn how to design and use!