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

- asynchronous
- synchronous
Hardware objects

Black boxes that do nothing unless stimulated by external events.

Class
The kind or type or model of a circuit.

Instance
A particular circuit on a particular board.

State
Internal register status or logic status of an object instance.

Provided interface
The set of pins on a circuit that recognize signals.

Required interface
The set of pins on a circuit that generate signals.

Method call
To raise an input signal and wait for a response (synchronous) or just continue (asynchronous).

A serial port

State
Internal registers

Provided interface
- Signal change
- Bit pattern received
- Clock pulse

Required interface
- Signal change
- Interrupt signal

Software Objects

Black boxes that do nothing unless stimulated by external events.

Class
Program behaviour expressed as state variable layout and method code.

Instance
A record of state variables at a particular address (the object’s identity).

State
Current state variable contents of a particular object.

Provided interface
The set of methods a class exports.

Required interface
Method calls issued to other objects.

Method call
Call to a function with the designated object address as the first argument.

The Counter example

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

In Java, not reactive objects!
We will use a little kernel called TinyTimber. We will use files as modules in C.

```c
#include "TinyTimber.h"

typedef struct {
  Object super;
  int x;
  char y;
} MyClass;

#define initMyClass(z) 
  { initObject ,0,z }

Mandatory! Specified and used by the kernel!
Unconstrained!
initMyClass corresponds to a constructor, it includes programmer defined initialization.

Using it
#include "MyClass.h"
MyClass a = initMyClass(13);
```

Comparing with Java

```java
class MyClass{
  int x;
  char y;
  MyClass(int z){
    x=0;
    y=z;
  }
}
```

Our constructors are just preprocessor macros!

But, we are doing all this to do something different than just function calls! We want to have the possibility of introducing the distinction between synchronous and asynchronous messages!
Reactive objects
Encoding state layout
Encoding methods
Programming with TinyTimber

Asynchronous calls

ASYNC(B,meth,73)

A

B (resting)
meth(B,73)

(Pseudo-) parallel execution!

Time

Strictly sequential execution between A and B.

(Pseudo-) parallel execution between A and B.

Strictly sequential execution between B’s methods!

Synchronous calls

SYNC(B,meth,73)

A

B (resting)
meth(B,73)

Some other B method
meth(B,73)

(Pseudo-) parallel execution between A and B’s other method.

Time

Strictly sequential execution between B’s methods and between A and the method called synchronously.
**Observations**

- Serialization of object methods looks just like standard mutual exclusion.
- A synchronous call is just like a mutex-protected function call.
- It is the asynchronous calls that introduce concurrency.
- Asynchronous calls further more need additional temporary storage (if a call cannot execute immediately)
- Suggestion: let an asynchronous call be equivalent to a synchronous call executed by a fresh thread!

**Implementing SYNC**

```c
int synch(Object *to, Meth meth, int arg){
    int result;
    lock(&to->mutex);
    result = method(to,arg);
    unlock(&to->mutex);
    return result;
}
```

Every object has to have its own mutex and we need a way to force every instance to be an object!

**Implementing ASYNC**

```c
void async(Object* to, Method meth, int arg){
    Msg msg = dequeue(&freeQ);
    msg->function = meth;
    msg->arg = arg;
    msg->to = to;
    if(setjmp(msg->context)!=0){
        sync(current->to,current->function,current->arg);
        enqueue(current,&freeQ);
        dispatch(dequeue(&readyQ));
    }
    STACKPTR(msg->context)=&msg->stack;
    enqueue(msg,&readyQ);
}
```

In **TinyTimber.h**

```c
typedef struct{
    mutex mutex;
} Object;

typedef int (*Meth)(Object*,int);
#define SYNC(obj,meth,arg) = :
    sync((Object*)obj,(Meth)meth, arg)

#define DEFINE_SYNC(obj,meth,arg) = \n    sync((Object*)obj,(Meth)meth, arg)
```
### Implementing ASYNC

In TinyTimber.h

```c
#define ASYNC(obj,meth,arg) = 
async((Object *)obj, (Meth)meth, arg)
```

### Summary

- Threads are replaced by asynchronous messages
- Old operation `spawn` is succeeded by `async`
- Old operations `lock` and `unlock` are only used inside `sync`
- The new kernel interface:
  ```c
  void async(Object *to, Meth meth, int arg)
  int sync(Object *to, Meth meth, int arg)
  ```
- Typedefs for `Object` and `Meth`
- Defines for ASYNC and SYNC

### ASYNC to self?

```c
ASYNC(A,meth,73)
```

A current A method meth(A,73)

**Strictly sequential execution!**

### SYNC to self?

```c
SYNC(A,meth,73)
```

A

**DEADLOCK!**
Deadlock

Deadlock arises when requesting new exclusive access to something you already have. In general, a chain of tasks may be involved:

- **T1** holds **m1**
- **T1** wants **m2**
- **T2** holds **m2**
- **T2** wants **m3**
- **T3** holds **m3**
- **T3** wants **m1**

A system in deadlock will remain stuck, unless a thread chooses to back off from its current claim . . .

Deadlock in the real world

Deadlock via **SYNC**

A cycle of possible simultaneous calls to **SYNC**
### Programming idiom

1. **Classes**
   
   All objects must *inherit* `Object`:
   ```cpp
typedef MyClass{
    Object super;
    // extra fields
}
```

2. **Objects**
   
   Object instantiation is done declaratively on the top level (static object structure):
   ```cpp
   ClassA a = initClassA(ival);
   ClassB b1 = initClassB();
   ClassB b2 = initClassB();
   ```

3. **Method calls**
   
   Whenever a method call goes to another object, either `SYNC` or `ASYNC` must be used.

   *(Tiny) Limitation*
   
   All methods must take arguments `self` and an int!

### Connecting the external world

- `interrupt` to `write to port`
- `interrupt` to `write to port`
- `read from port`
Making the methods explicit

The top-level object

Connecting interrupts

The microprocessor itself!
- It is just like any other reactive object!
- it is implicitly instantiated when power is turned on
- its state is all global variables, of which many will be reactive
  objects in their own right
- its methods are the installed interrupt handlers
- its self is only conceptual (there is no concrete pointer . . .)
- The top-level object methods are scheduled by the CPU
  hardware, not by the TinyTimber kernel!

In TinyTimber.c

```c
#define INTERRUPT(vector, stmt) \
ISR(vector) {stmt; schedule();}
```

stmt can be any C statement that manipulates global state and/or
calls methods of internal objects.

Avoid SYNC calls from interrupts, they will report deadlock if the
receiver object is busy.

schedule(); is a refined version of yield().
Example

A Counter example (counter.h)

```c
#include "TinyTimber.h"
typedef struct{
    Object super;
    int val;
} Counter;

#define initCounter(n) {initObject(),n}
```

A Counter example (counter.c)

```c
int inc(Counter *self, int arg){
    self->val = self->val + arg;
}
int reset(Counter *self, int arg){
    self->val = arg;
}
```

Example client

```c
In main.c
Counter counter = initCounter(0);
INTERRUPT(SIG_PIN_CHANGE1,ASYNC(&counter,inc,1));
```

Notice

- Calls to ASYNC(counter,reset,0); will always be inside the body of some method.
- That method will ultimately be called from an interrupt!

Reset

When system starts up, a reset signal is generated by the hardware. There will be an interrupt routine like any other one . . .

Complication

The reset routine cannot return as it has not really interrupted anything!

In the active system view this is interpreted as compute until someone turns off the power!

main()

The main() function in C is an abstraction of the reset handler . . .

. . . just as a program is an abstraction of the notion of running a computer until it stops

In traditional programs main() does indeed return, which can be understood as a request to the OD to turn off the power to the virtual computer that was set up to run the program!

In a reactive system we do not want power to be turned off at all, but we also do not want to let main() compute forever just to keep it from returning . . . a reactive system rests when it is not reacting.
The idle task

Solution

Let `main()` finish by literally putting the CPU to sleep until the next interrupt! (Most architectures have a special machine instruction that does so!)

We want `main()` to finish by calling this instruction:

```c
void idle(){
    ENABLE();
    while(1)SLEEP();
}
```

Implemented in TinyTimber

```c
#define STARTUP(stmt) 
int main(){initialize();stm;idle();}
```

Example client + reset

```c
Counter counter = initCounter(0);
INTERRUPT(SIG_PIN_CHANGE1, ASYNC(&counter,inc,1));
STARTUP(ASYNC(&counter,reset,0));
```

Notice!  No `main()` function

Sanity rules

In a system of reactive objects
- Methods only access variables that belong to `self`.
- Global variables that are not objects, are considered local to the top-level object.
- Method calls between objects that are wrapped within a `SYNC` or `ASYNC` shield.

Properly upheld, these rules guarantee a system that is
- free from deadlock (provided the absence of cyclic `SYNC`)
- free from critical section race conditions