

Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

**Lecture 6: Inheritance
and Finite State Machines**

Contents

This lecture will include:-

- Inheritance:
 - Creating new classes based on the features of existing classes
 - How this can be used to further improve code structure for concurrency.
- Finite State Machines
 - Background
 - Example: simple lights sequencer

Inheritance



- Consider a tiger.
- It is a kind of cat, with the attributes of a cat (age, colour etc) but with some additional attributes. Also, tigers make a different sound to domestic cats
- C++ allows us to create a class that contains all of the member functions and variables of a Cat (without having to re-write them), but with the option to adapt them and add more.
- This is called ***inheritance***

Recalling Our Cat Class

Recalling from last lecture (with a few tweaks):-

```
Class Cat
```

```
{
```

```
public:
```

```
    int    Sound(boolean isContent);
```

```
    float  bodyMassToAgeRatio(void);
```

```
    void   setAge(int ageOfCat);
```

```
    int    getAge();
```

```
    void   setWeight(float weightOfCat);
```

```
    float  getWeight();
```

```
private:
```

```
    int    age;
```

```
    float  weight;
```

```
};
```

Recalling Our Cat Class

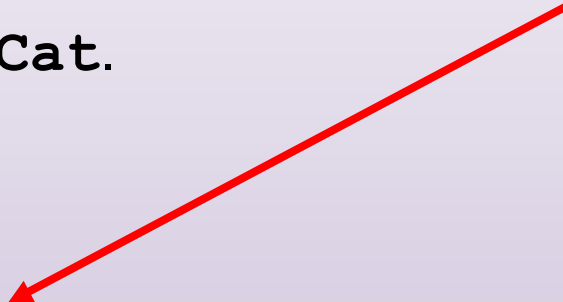
...with a modified `Sound()` function:-

```
int Cat::Sound(boolean isContent)
{
    if(isContent) {
        playSoundFile("purr.wav");
    } else {
        playSoundFile("meow.wav");
    }
    return 0;        // 0 indicates success
}
```

Public inheritance

A **Tiger** class, based on **Cat**.

The base class is
Cat



```
#include "Cat.h"
```

```
Class Tiger : public Cat
```

```
{
```

```
public:
```

```
    Tiger(int age);                // constructor
```

```
    int    Sound(boolean isContent); // overridden function
```

```
    void    incrementZebrasEaten();
```

```
    int     getTotalZebrasEaten();
```

```
private:
```

```
    int     eatenZebras;
```

```
};
```

and the implementation..

```
Tiger::Tiger()    // Default constructor
{
    // that does nothing.
}                // e.g. Tiger tony;
```

```
Tiger::Tiger(int ageOfCat) : Cat(ageOfCat)    // Initializes base class (Cat) with the age.
{
    // age is passed as an argument.
}
```

For the overridden member function, **Sound(boolean isContent) :**

```
int Tiger::Sound(boolean isContent)
{
    // tigers do not purr, so ignore the isContent argument
    playSoundFile("roar.wav");
    return 0;
}
```

Thus:

Example: creating an instance of **Tiger**

```
Tiger zimba;
```

```
void setup()
{
  Serial.begin(115200);
  zimba.setAge(15);
  Serial.print("The age is ");
  Serial.println(zimba.getAge()); // just as for Cat.
  zimba.Sound(false); // plays roar sound
  zimba.Sound(true);  // also plays roar sound
  Serial.print("The body mass is: ");
  Serial.print(zimba.bodyMass()); // displays body mass
}
```


How Can Inheritance Facilitate Our Concurrency Model?

- Last time, we saw how having a class to deal with concurrency can drastically improve the structure of the code.
- Why not create classes which encompass everything: namely the concurrency timings plus the finite state machines.
- This is where inheritance wins out:-
 - A class for a simple lights controller could inherit from the Concurrent base-class.
 - The FSM would be integrated into the class.
 - Net result a module capable of concurrency, with everything encapsulated.
 - Code required by the main program will be minimal.

How Can Inheritance Facilitate Our Concurrency Model?

- For each module, declare a new class.
- From our previous example, let's declare a class called RedBlinker for the red LED.
- Create a file called **RedBlinker.h**

```
#ifndef RedBlinker_h
#define RedBlinker_h

#include "Concurrent.h" // Base class

class RedBlinker : public Concurrent {
public:
    void process();
    RedBlinker();

private:
    int state;
};

#endif
```

Take a similar approach to create YellowBlinker and GreenBlinker modules

How Can Inheritance Facilitate Our Concurrency Model?

- Add a corresponding .cpp file. In this case **RedBlinker.cpp**

```
#include "RedBlinker.h"  
#include "hal.h"
```

```
RedBlinker::RedBlinker()  
{  
    isRunning = false;  
    module_delay = 500;  
    state = 0;  
}
```

**Constructor:
Setting-up the
module**

```
void RedBlinker::process()  
{  
    // Only process the finite state machine on a 'tick' / 'step'  
    if (actionTask())  
    {  
        // FSM belongs here  
    }  
}
```

```
switch (state) FSM  
{  
    case 0:  
        HAL_ledRed1On;  
        state = 1;  
        break;  
    case 1:  
        HAL_ledRed1Off;  
        state = 0;  
        break;  
}
```

Take a similar approach to
create YellowBlinker and
GreenBlinker modules

Private vs Protected?

- Class members of the `Concurrent` base-class have originally been declared as `private`.
- Compilation errors will occur.
- Problem: `private` members are not accessible to derived classes.
- Possible solution: make them `public` – but this is undesirable.
- Possible solution: make them **protected**
- **protected** members are fully visible to derived classes, but are otherwise `private`.

Private vs Protected?

A small change therefore needs to be made to the `Concurrent` base-class definition.

Was

private:

```
unsigned long    module_time;  
unsigned long    module_delay;  
bool             module_doStep;  
bool             isRunning;
```

Now

protected:

```
unsigned long    module_time;  
unsigned long    module_delay;  
bool             module_doStep;  
bool             isRunning;
```

Further Additions

Add new public member function to `Concurrent` base-class.
This allows the module to be run or held based on a variable.

In `Concurrent.h` (as a public entity)
`void setRunning(bool start);`

In `Concurrent.cpp`
`void Concurrent::setRunning(bool start)`
`{`
 `isRunning = start;`
`}`

Main Code

```
#include "hal.h"

#include "RedBlinker.h"
#include "YellowBlinker.h"
#include "GreenBlinker.h"

// Create instances of each class
RedBlinker    redBlinkControl;
YellowBlinker yellowBlinkControl;
GreenBlinker  greenBlinkControl;

void setup() {
    HAL_gpioInit();

    // Initialize the red LED module
    redBlinkControl.setRunning(true);

    // Initialize the yellow LED module
    yellowBlinkControl.setRunning(true);

    // Initialize the green LED module
    greenBlinkControl.setRunning(true);
}
```

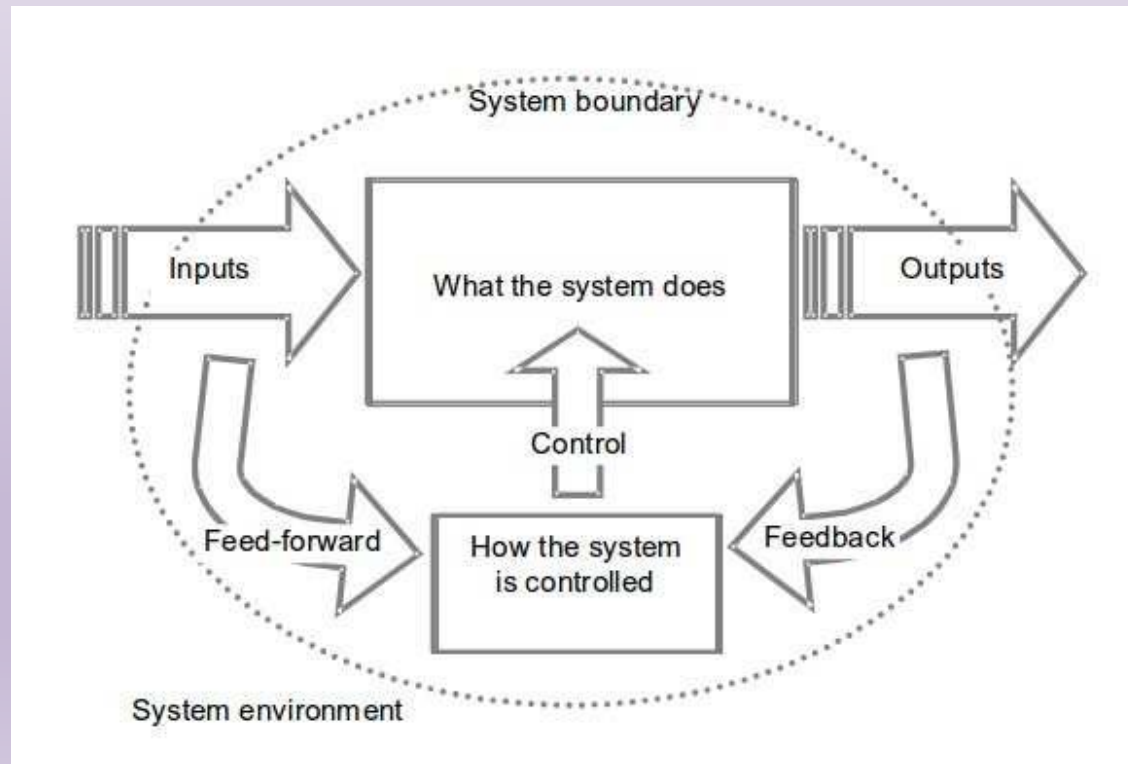
Everything is now encapsulated
Everything is now modularized

```
void loop() {
    // Let the modules do their thing.
    redBlinkControl.process();
    yellowBlinkControl.process();
    greenBlinkControl.process();
}
```

Finite State Machines

Systems Have....

- A boundary and an environment
- Inputs, which they transform into outputs (actuators)
- Control (feedback or feed-forward)



Parallel execution

- The environment is asynchronous to the system.
 - i.e. the environment works in parallel to your system
- Inputs are sporadic in general
 - Your system must observe its inputs
- If you miss an input you have lost information, i.e. you have failed to keep up with your environment.

Hypothetical Example

Suppose you have a single input and a single output.

The input is a push-button switch and the output is a single LED.

- The LED is either OFF, or repeatedly blinking on for a second, off for a second.
- The push-button toggles the LED from off to blinking and back again.
- The LED is toggled on a press and release of a button
- The button can be pressed and released at any time, i.e. the LED can be toggled at any time.

Probably we can try....

```
bool bPushed() {...}    // true on a pushed button
bool bReleased() {...}  // true on a released button
...
void loop() {
    ...
    if ( bPushed() ) {
        delay(1000);
        if ( bReleased() ) {
            LEDon();
            delay(1000); // that is wait 1000 milliseconds
            LEDoff();
            delay(1000); // same difference
        }
    }
}
```

Problem is....

- `bPushed()` is true precisely when `bReleased()` is false
- `bPushed() && bReleased()` is **never** true
- they have to occur in a sequence
- you are not monitoring the state of the button while you are waiting for the second to complete
- need to do two things at the same time
- `delay()` must not be seen in your code!

The challenge

There are two things you need to do at the same time

1. Monitor the button with all its complexities with respect to its bouncing behaviour.
2. Control the LED with all its delays

While you are observing a delay for the LED you must be watching the button as well.

Behaviour and time

Fundamentally, both the button and the LED have behaviour and behaviour takes time. This means:-

- If you are watching the button to recognize its behaviour, your eyes are not on the LED
- While you are waiting on the LED, you have no idea what the button is doing.

How do we solve this problem?

What is behaviour?

- We will say that behaviour is sequence of changes in time.
- We need a formal language with syntax and semantics to capture and reason about behaviour.
- This is where Finite State Machines are helpful.

FSMs

Formally an FSM can be defined as a collection of:

- **Inputs:** An input is a variable that is controlled by the system's environment.
- **Outputs:** An output is a variable that is controlled by the system.
- **States:** A state is a mapping between outputs and values, i.e. a state tells you what is the value of every output in your system.
- **Next State (transition) function:** This is a function that takes in the current state and the current values of the inputs and calculates the next state.
- **Initial State:** This is the state at the beginning of time.

What is behaviour again?

- FSM generates a sequence of states in time.
- The state sequence depends on the inputs, i.e. the environment determines which particular sequence (trajectory, history, run) the system is currently on.
- Subsequent runs are not guaranteed to generate the exact same sequence.
- The set of all possible runs constitutes the behaviour of the FSM.

FSM Diagram

We will use a diagram to describe FSMs:-

1. List all possible states
2. Declare all variables including inputs and outputs.
3. For each state, list possible transitions with their conditions, to determine the next state (transition) function.
4. For each state list associated actions (assignments for outputs and variables)
5. For each state, ensure exclusive and complete exiting conditions, i.e.
 - The transition function gives you at most one next state:- if the transition function gives you more than one state then the FSM will be non-deterministic.
 - The transition function gives you at least one next state:- if the transition function fails to calculate a next state this means you have not predicted all possible cases for your inputs.

General Parallel Template

```
void loop() {  
    {  
        FSM_1;  
    }  
    {  
        FSM_2;  
    }  
    {  
        FSM_3;  
    }  
}
```

- Each FSM executes 1 state every `loop()` iteration and then drops out of its `switch` statement. This constitutes a step for the FSM.
- All FSMs are executing the same number of steps as the iterations of the `loop()` function.
- This means that on iteration per iteration basis, the FSMs are executing in parallel.

Summary

- Showed how inheritance can be used to fully encapsulate timings and FSMs into classes, resulting in a highly modularized structure.
- Introduced the concept of Finite State Machines (FSMs)