Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

Lecture 8: Schedulers

Contents

This lecture will include:-

- Introduction to schedulers
- Real-time systems
- Types of schedulers
- Implementation of a non-pre-emptive fixed-time scheduler

Scheduler

- Recall the following from our Summative assessment document (published on Blackboard).

- Our coursework involves:-
  - Traffic Light Controller
  - 7-segment display via a shift register
  - Orientation detection via accelerometer.
  - Button 1 for counting-up button presses on the 7-segment display.
  - Button 2 for changing the mode of operation.

Scheduler

- We can possibly implement each task individually now with little problem. These were the Formative assessment exercises.
- However we will need to do several things simultaneously (concurrently)
- When we run everything together, we must change the priority on the Traffic Light according to the orientation of the accelerometer.
- In the same time we must always monitor the buttons and change the mode or count accordingly
- The buttons must be debounced as well.
- And there is also the heartbeat.
- Some subsystems will run all the time irrespective of the mode. Others only run when needed for their respective mode.
- This means we need a part of our system that controls what runs when.

Scheduler

- We see straight away that we must be able to:
    - Run tasks in parallel, e.g. Traffic Lights **and** Buttons
    - Make sure the debouncing does not kill the timing for the Traffic Lights.

- We must also run different tasks in different modes.

- This forces us to use a **Scheduler**.

Scheduler

- A scheduler is a module, like any other module in your system
- Its task is to decide what tasks run at any one time in your system.
- Schedulers must know the deadlines and priorities of each task they need to schedule.
- Usually when we talk about schedulers, we talk about **Real Time systems**.

Real-Time Systems

- We define real-time systems with

Correct answer given off-time is wrong!

- This means that given the choice of
    - A numerically correct and late result, or
    - An approximately correct and on-time result

we always have to take the latter rather than the former.

Real-Time Systems

- Hard Real-Time Systems are usually defined that way:-
  - Need results to be delivered on time, but at the expense of some accuracy

- Soft Real-Time Systems are defined as:-
  - Meeting timing constraints most of the time.
  - Some deadline miss is tolerated.

- **Question:** How will you characterize your coursework?

Types of Schedulers

- Pre-emptive vs non-pre-emptive.
- Periodic vs sporadic task set
- Dynamic vs static task set
- Fixed vs adaptive scheduler
- A successful schedule is a schedule where all tasks meet their deadlines.

Where we stand

NON-ENCAPSULATED APPROACH

- The non-encapsulated modules can be configured to perform a task at required time instants.
- These modules can also be put into running or held states.

- To allow a module to run:-

```
init_module0_clock = false;
```

- To hold a module from running:-

```
init_module0_clock = true;
```

This inverted logic seems illogical…….

Where we stand

NON-ENCAPSULATED APPROACH

```
{ // module 0
   static unsigned long module_time, module_delay,;
   static bool module_doStep;

   if (init_module0_clock) {
     module_delay = 500;
     module_time = millis();
     module_doStep = false;
     init_module0_clock = false;
   }
   else {
     unsigned long m = millis();
     if ( (m - module_time) > module_delay ) {
       module_time = m;
       module_doStep = true;
     }
     else module_doStep = false;
   }

   if (module_doStep) {
     // Do your task here
   }
 }
```

If **`init_module_clock`** is **`true`** then the timing code will not run during this iteration.

If **`init_module_clock`** is **`false`** at the start of the iteration, then the timing code will run.

Running / holding a module
NON-ENCAPSULATED APPROACH

```
void loop
{
    // module 0 code


    init_module0_clock = true;
}
```

Module code sets the value of **`init_module_clock`** to **`false`**.

`init_module_clock` is set to **`true`**.

Net result: module is held.

Running / holding modules

NON-ENCAPSULATED APPROACH

Consider the very first example:-

- Module 0 drove the Red LED (delay time 500ms)
- Module 1 drove the Yellow LED (delay time 300ms)
- Module 2 drove the Green LED (varied timings)
- Assuming SW1 and SW2 are configured as input pullups: adding two lines of code at the end of the `loop()` function:-

```
init_module0_clock = HAL_sw1Pressed;
init_module1_clock = HAL_sw2Pressed;
```

stops Module 0 running when SW1 is pressed, and stops Module 1 running when SW2 is pressed.

- This technique forms part of the mechanism behind the scheduler.

Developing the scheduler

ALL APPROACHES

Determine which modules will run all the time

- Heartbeat
- Debouncers

and which will be scheduled to run / be halted according to the mode of the system.

- Traffic lights
- Button counter
- Orientation sensor

Create a new module for the scheduler.

- Module delay ***<u>MUST</u>*** be less than that of the modules to be scheduled
- Requires a Finite State Machine (FSM)

Developing the scheduler

ALL APPROACHES

- Heartbeat and debouncer(s) will run continuously and will not need scheduling. They do not feature in the analysis.
- FSM will need to be developed such that the “mode” will advance to the next mode on a debounced press of SW2.
- The 5 modes are:-
  - **Mode 1:** Traffic lights on. Counter and Orientation off.
  - **Mode 2:** Counter on. Traffic lights and Orientation off.
  - **Mode 3:** Orientation on. Traffic lights and Counter off.
  - **Mode 4:** Traffic lights and Counter on. Orientation off.
  - **Mode 5:** Traffic lights and Orientation on. Counter off.
- Are there issues in transitioning on just a debounced press?

Formulating an FSM for the scheduler

See the DocCam presentation.

Implementing the scheduler - 1

NON-ENCAPSULATED APPROACH (Snippet of FSM)

```
switch(state)
{
         :   etc
  case n:
    // Mode 1
    init_module0_clock = false;     // traffic lights running
    init_module1_clock = true;      // counter halted
    init_module2_clock = true;      // orientation halted
    if (B2_state == DEBOUNCED_PRESS) state = n+1;
    break;
  case n+1:
    // Mode 2
    init_module0_clock = true;      // traffic lights halted
    init_module1_clock = false;     // counter running
    init_module2_clock = true;      // orientation halted
         :
         :   etc
}
```

The value of ***n*** is left for you to determine

In this example:-
Module 0 controls the traffic lights
Module 1 controls the counter
Module 2 controls the orientation sensor

Implementing the scheduler - 2

PARTIAL ENCAPSULATION APPROACH

Recall the partial encapsulation example from Week 5
Add provision for a scheduler.

Include the `Concurrent` class library:-

```
#include "Concurrent.h"
```

For each code module, we created an instance of the `Concurrent` class.

```
Concurrent redControl;
Concurrent yellowControl;
Concurrent greenControl;
```

Now create an instance of the `Concurrent` class for the scheduler.

```
Concurrent scheduler;
```

Implementing the scheduler - 2

PARTIAL ENCAPSULATION APPROACH

Recall the partial encapsulation example from Week 5
Add provision for a scheduler.

Create a function for the Scheduler FSM:-

```
void schedulerFSM()
{
  static int state = 0;
  // FSM code goes here
}
```

In `setup()` each module can be set up:

```
  redControl.setModuleDelay(500);
  redControl.setRunning(false);
  yellowControl.setModuleDelay(300);
  yellowControl.setRunning(false);
  greenControl.setModuleDelay(600);
  greenControl.setRunning(false);
  scheduler.setModuleDelay(???);
  scheduler.setRunning(true);
```

Implementing the scheduler - 2

PARTIAL ENCAPSULATION APPROACH

Recall the partial encapsulation example from Week 5
Add provision for a scheduler.

```
void loop()
{
  if (redControl.actionTask())
  {
    redTask();
  }

  if (yellowControl.actionTask())
  {
    yellowTask();
  }

  if (greenControl.actionTask())
  {
    unsigned long new_delay;
    new_delay = greenTask();
    greenControl.setModuleDelay(new_delay);
  }

  if (scheduler.actionTask())
  {
     schedulerFSM();
  }
}
```

Original code

Scheduler added

Implementing the scheduler - 2

PARTIAL ENCAPSULATION APPROACH Snippet of `schedulerFSM()`

The value of ***n*** is left for you to determine

```
switch(state)
{
         :   etc
  case n:
    // Mode 1
    redControl.setRunning(true);          // Red module running
    yellowControl.setRunning(false);      // Yellow module halted
    greenControl.setRunning(false);       // Green module halted
    if (B2_state == DEBOUNCED_PRESS) state = n+1;
    break;
  case n+1:
    // Mode 2
    redControl.setRunning(false);         // Red module halted
    yellowControl.setRunning(true);       // Yellow module running
    greenControl.setRunning(false);       // Green module halted
         :   etc
         :
}
```

Implementing the scheduler - 3

FULL ENCAPSULATION APPROACH

Recall the full encapsulation example from Week 7. Add provision for a scheduler.

Firstly, create a class for the Scheduler, in a file called Scheduler.h

```
#ifndef _Scheduler_h_
#define _Scheduler_h_

#include "Concurrent.h"    // Base class
#include "Hal.h"


class Scheduler : public Concurrent {
  public:
    void              process(switch_state_t B2_state);
    Scheduler();
    bool              getRunRed();
    bool              getRunYellow();
    bool              getRunGreen();

  private:
    int               state;
    bool              runRed, runYellow, runGreen;
};

#endif
```

New accessor functions

New member variables

Implementing the scheduler - 3

FULL ENCAPSULATION APPROACH

Recall the full encapsulation example from Week 7. Add provision for a scheduler.

Next, implement the Scheduler, in a file called Scheduler.cpp

```
#include "Scheduler.h"
#include "hal.h"

Scheduler::Scheduler()
{
  isRunning = false;
  module_delay = ???;
  state = 0;
}
```

Constructor:
Need to carefully choose the module delay

```
bool Scheduler::getRunRed()
{
  return runRed;
}
```

Accessor Function:
Need equivalent functions for runYellow and runGreen

```
void Scheduler::process(switch_state_t B2_state)
{
  // Only process the finite state machine on a 'tick' / 'step'
  if (actionTask())
  {
    // FSM belongs here

  }
}
```

Scheduler FSM
Needs to be implemented here

Implementing the scheduler - 3

FULL ENCAPSULATION APPROACH Snippet of the scheduler FSM.

```
switch(state)
{
         :   etc
  case n:
    // Mode 1
    runRed = true;              // Red module running
    runYellow = false;          // Yellow module halted
    runGreen = false;           // Green module halted
    if (B2_state == DEBOUNCED_PRESS) state = n+1;
    break;
  case n+1:
    // Mode 2
    runRed = false;             // Red module halted
    runYellow = true;           // Yellow module running
    runGreen = false;           // Green module halted
         :   etc
         :
}
```

The value of ***n*** is left for you to determine

Implementing the scheduler - 3

FULL ENCAPSULATION APPROACH - Main Code.

```
#include "hal.h"

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


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


void setup() {
  HAL_gpioInit();

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

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

  // Initialize the green LED module
  greenBlinkControl.setRunning(false);

  // Initialize the scheduler
  scheduler.setRunning(true);

}
```

```
void loop() {
  switch_state_t    B2_state;

  // Let the modules do their thing.
  redBlinkControl.process();
  yellowBlinkControl.process();
  greenBlinkControl.process();

  // Acquire B2_state through debouncers

  // Run the scheduler
  scheduler.process(B2_state);

  // Turn modules on/off using
  // information from scheduler
  redBlinkControl.setRunning(scheduler.getRunRed());
  yellowBlinkControl.setRunning(scheduler.getRunYellow());
  greenBlinkControl.setRunning(scheduler.getRunGreen());
}
```

Summary

During this session we covered.

- The need for scheduling.
- An overview of schedulers.
- How code modules can be activated by schedulers
- The development of a scheduler.

Next week we will look at techniques for making code more robust.