3.10 FSM and RingBuffer

Finite State Machines

Ring buffers

Overview

Aims

- Learn about FSMs and Ring buffers

Learning outcomes – you should be able to…

- Implement very simple finite state machines
- Use Ring buffers as efficient FIFO buffers

Finite state machines

Finite state machine (FSM)

An FSM is an abstract machine often used in computing and digital logic

The behaviour/operation of the "machine" depends on the state it is in – an FSM changes its behaviour according to the state

- E.g. the operation of play/pause button in most music players depends on the whether the player is currently playing or already paused

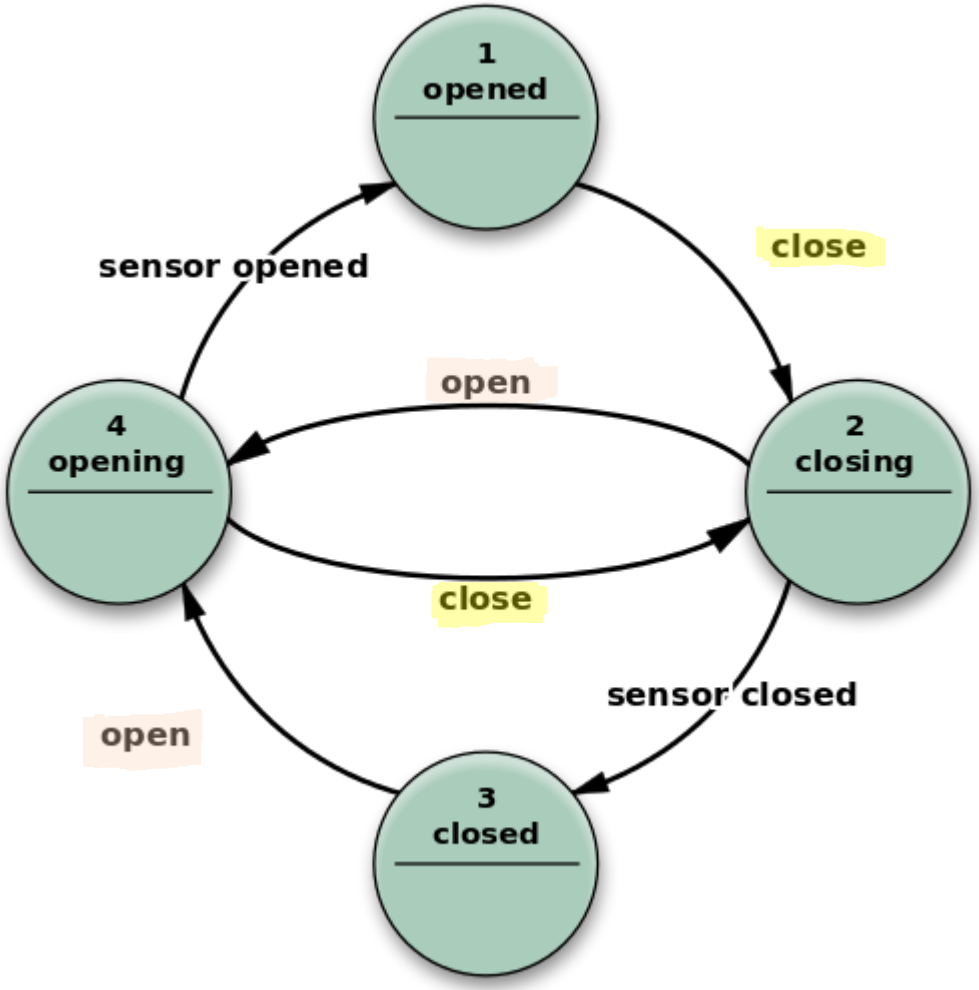
An FSM can only be in one state at a time – called the current state

- E.g. the music player may be in either the playing or the paused state but not both at the same time

The current state of an FSM may be changed from one state to another (a **state transition**) in response to some triggering condition

- E.g. if the the music player is currently in the playing state and play/pause button is pressed, then the player pauses music playback and transitions to the paused state

Example FSM for an electronic door



Use of FSMs

We have 2 simple uses for FSMs (but there are many)

1. If we want to implement any programme that has a concept of different modes of operation where the behaviour should be different in different modes, then we should use an FSM
   - E.g. a system which needs to do different things in calibration mode vs. running mode
2. If we have task that takes longer than the desired superloop duration to complete, then we need to break the task into sub-tasks, each of which will be tackled in a consecutive superloop iteration. To do this we use an FSM to keep track of which subtask should be executed.
   - E.g. if we need superloop to run 100 times/sec (i.e. 10 ms per iteration) but one of the tasks takes 50 ms to complete, then it will need to be broken into small chunks (subtasks) which can be completed over 5 iterations

We will focus on the first use only in 2018-2019.

Implementing an FSM

Declare constants or enum values to represent the different states

You need a variable (suitably intialized) to represent the current state – call this the state variable

You need selection statements (e.g. if-statement or switch-statement) which uses the state variable to decide which code to execute

In each state, the behaviour in response to inputs will usually be different

In each state, there will also be code that looks for special triggering conditions that indicate a state transition should occur

When the triggering condition arises

- change the value of the state variable
- If necessary, write code to execute any actions that must be performed during the transition from the current state to the new state, e.g. stopping music playback when the state transitions from playing to paused

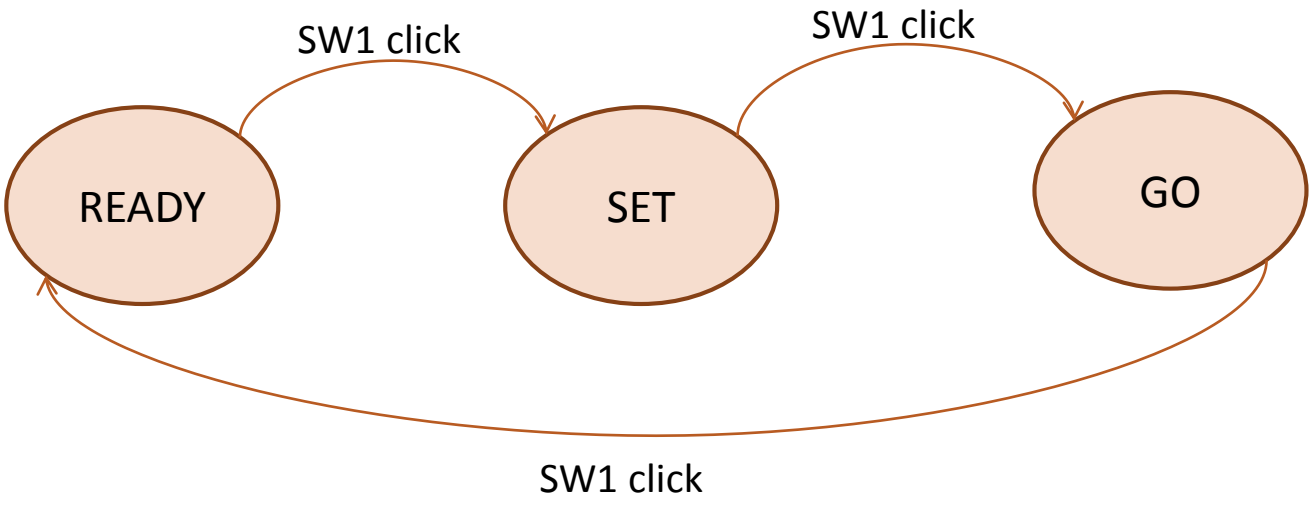
Using FSM to implement operation modes


```
…
void loop(void) {
  static int gameState = READY;
  int buttonState;

  // common input used by all states
  buttonState = readButtonState();

  // execute behaviour specific to current state of FSM
  if (gameState == READY) {
      if (buttonState == CLICKED)
        gameState = SET; // transition to SET for next time
      else
        // do ready mode stuff…
  } else if (gameState == SET) {
      if (buttonState == CLICKED)
        gameState = GO; // transition to GO for next time
      else
        // do SET mode stuff…
  } else { // default – GO mode
      if (buttonState == CLICKED)
        gameState = READY; // transition to READY for next time
      else
        // do GO mode stuff…
  }
}
```

Worked example – basic operation modes using FSM

Develop a short programme to do the following:

- The system has 3 modes: READY, SET, and GO.
- Each time SW1 is clicked, change to the next mode. (The next mode after GO is READY.)
- In each of the modes, the different behaviour is simply to print the current mode name to serial output



Contd.

There are two main ways to implement this kind of FSM, depending on its complexity.

For a very simple FSM (such as this one) that has very little to do in each state (i.e. the code that is specific to each state is quite short), it is reasonable to implement the FSM completely inside the loop function.

*See **Lec310_FSM1_NoFunctions_1819** for this kind of implementation.*

Quite often you would like to perform some specific actions only when first entering a state (i.e. after transitioning from some different state). You can do this by using a static variable to note when the state changes and performing the entry action only when this variable indicates that the state has changed

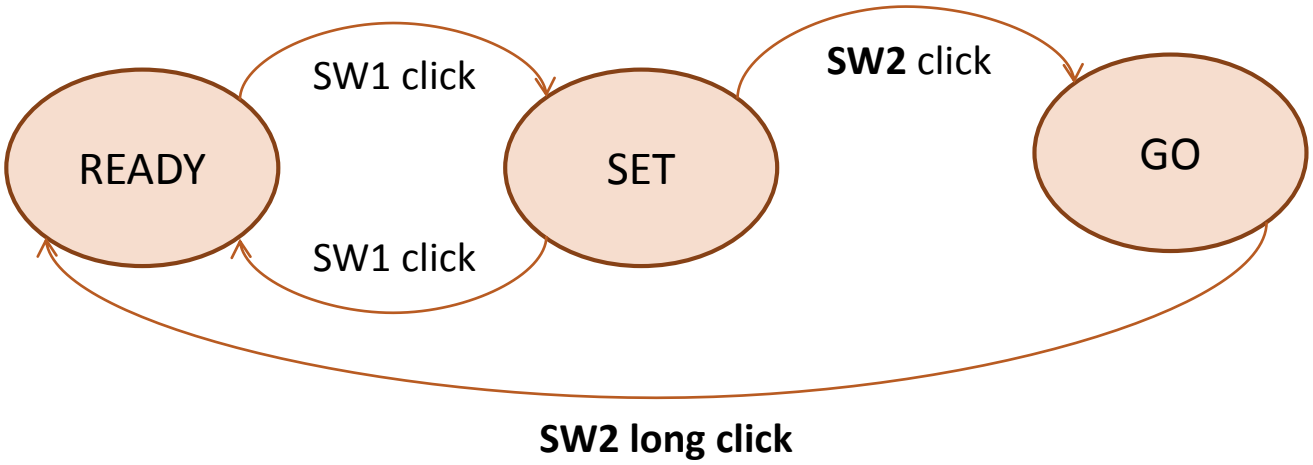
*See **Lec310_FSM2_NoFunctions_1819** for this slightly enhanced functionality.*

Worked example – operation mode functions using FSM

This is very similar to the previous problem but the button clicks differ and we will handle the state behaviour inside functions rather than directly in the loop function.

Develop a short programme to do the following:

- The system has 3 modes: READY, SET, and GO with transitions as indicated below
- In each of the modes, the behaviour is as follows:
  - READY: print the mode on entry into the state, blink red LED slowly (1s)
  - SET: print the mode on entry into the state, blink red LED faster (250ms)
  - GO: print the mode on entry into the state, alternately blink red and green LED fast (100ms)



Contd.

For more complex FSMs that has a bit more to do in each state (i.e. the code that is specific to each state is longer), it makes more sense to implement the handling of each specific state inside its own function.

Usually each state function can cause a state transition and this is communicated back to the caller by return the next state to use (which can be the same or changed).

Often the detection of external inputs (e.g. button clicks) should not be repeated in each individual state, so we do this before running the FSM and pass the resulting inputs into the state functions as needed.

*See **Lec310_FSM3_WithFunctions_1819** for this slightly more complex implementation of an FSM.*

Self test– display modes using FSM

Develop a short programme to do the following:

- Keep track of the currently selected Bar LED and flash the currently selected Bar LED with a specific pattern (see below) that depends on both a counter value and the display "mode" (which is just a "state" by another name)
- Increment the selected Bar LED each time SW1 is clicked, regardless of display mode. The valid LED numbers are 0 to 3 inclusive and wrap around must be implemented to stay within this range.
- On changing to a new display mode and at 1 second intervals do the following based on display mode…
  - Simple flash: 1 short flash of the selected LED, where 0 means BarLED1
  - Double flash: 2 short flashes of the selected LED
  - Adjacent flash: 3 short flashes of the selected LED and the next higher numbered LED, e.g. 0 means BarLED1 and BarLED2 both light together
- Define functions as appropriate

Ring buffer

Ring buffer

A ring buffer (AKA circular buffer) is backed by a fixed size data structure (usually an array) and is usually used to implement a FIFO queue in embedded systems

- New items are added to the end/back of the ring buffer
- Old items are removed from the start/front of the ring buffer

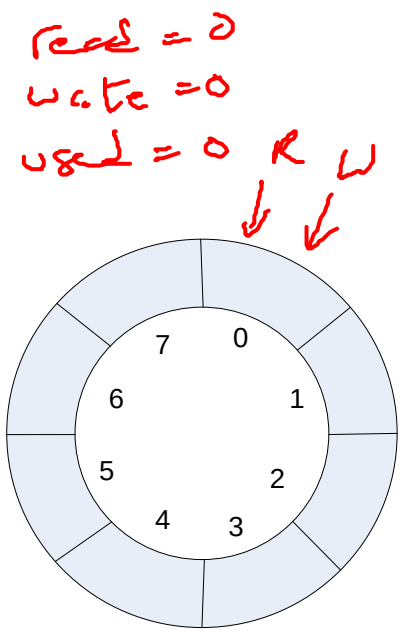
The main benefit to the ring buffer is its fixed size and the fact that data does not need to be copied when an item is removed from the front of the FIFO – this makes is fast

The main consequence of the fixed size is that usually ring buffers are designed so that adding one more item to a full ring buffer results in the oldest item being overwritten. Only the most recent N items can be stored (where N is the size of the ring buffer)

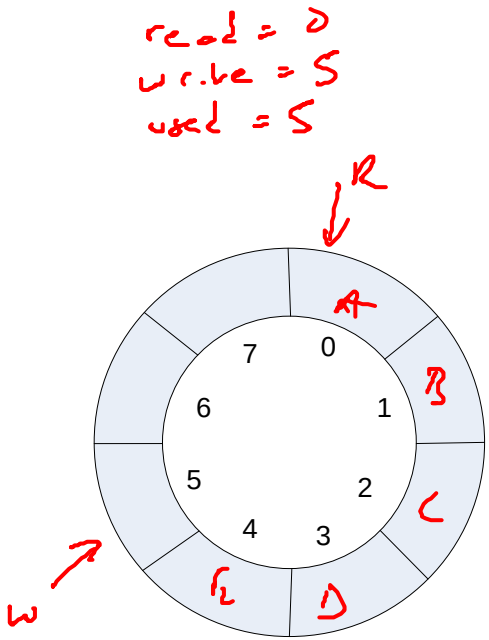
A ring buffer normally has a read index (where the oldest item is located) and a write index (where the next new item should be added) and perhaps a variable to track the number of used elements.

The read and write index wrap around at 0 and the buffer size. (This is what gives the ring buffer its name!)

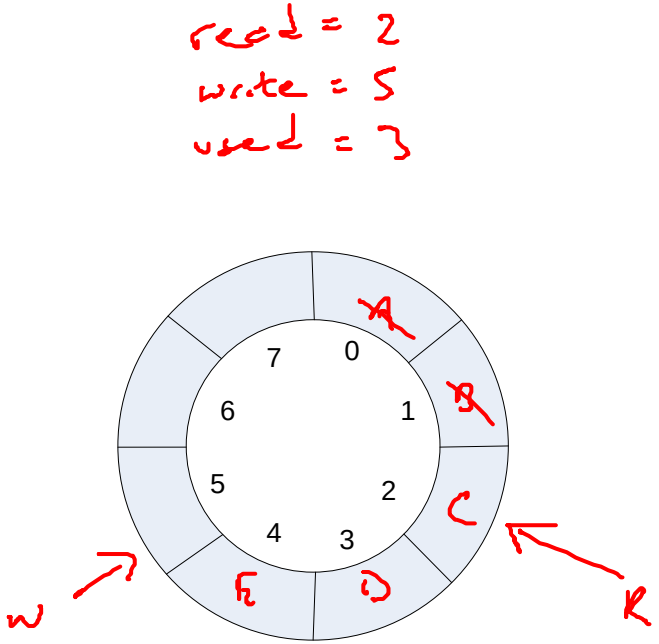
initial



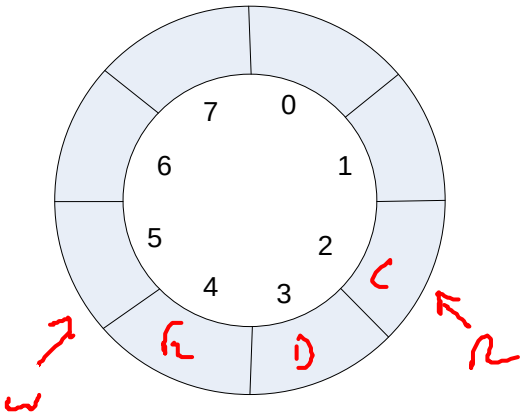
Add 5 items



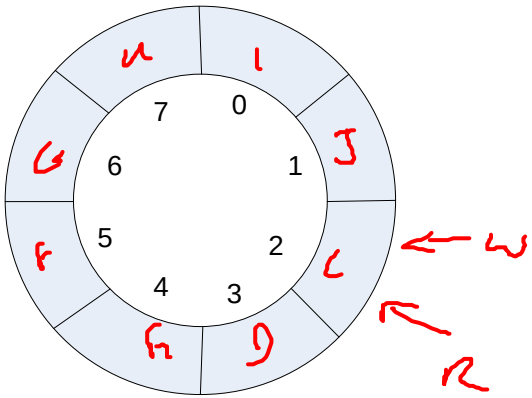
Remove 2 items



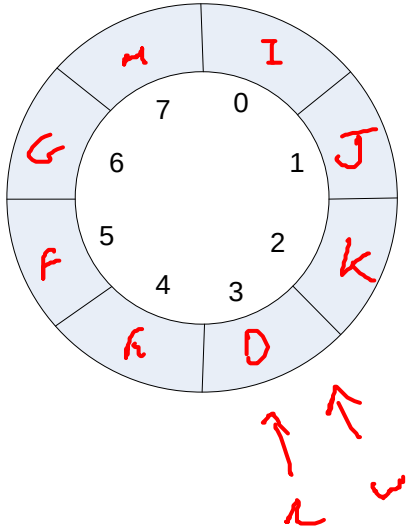
3 items



Add 5 items (now full)



Add 1 more
(overwrites oldest)



Ring buffer example

A ring buffer has functions for the following:

- Intialize the ring buffer
- Append 1 element (to the current end/back of the ring buffer)
- Remove 1 element (from the current start/front of the ring buffer)
- Get the element at a particular index (relative to the current start of the ring buffer)

Develop a short programme to do the following:

- Use a ring buffer of char
- When SW1 is clicked, append a random character (A-Z) to ring buffer
- When SW2 is clicked remove one char from the ring buffer
- Dump the contents of the ring buffer after each operation

See ***BasicRingBuffer*** for an example that demonstrates this.