Managing Flow of Activities

Chapter 5

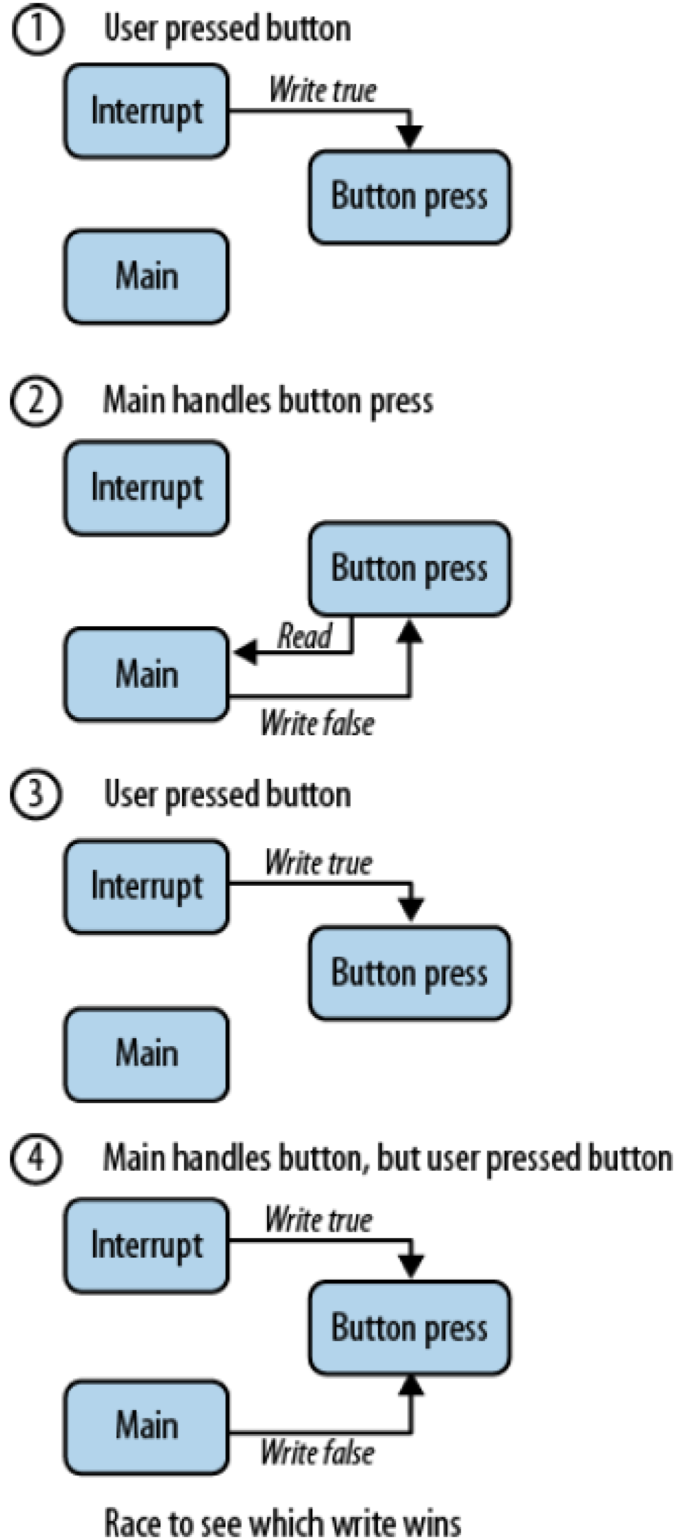
How a System Starts

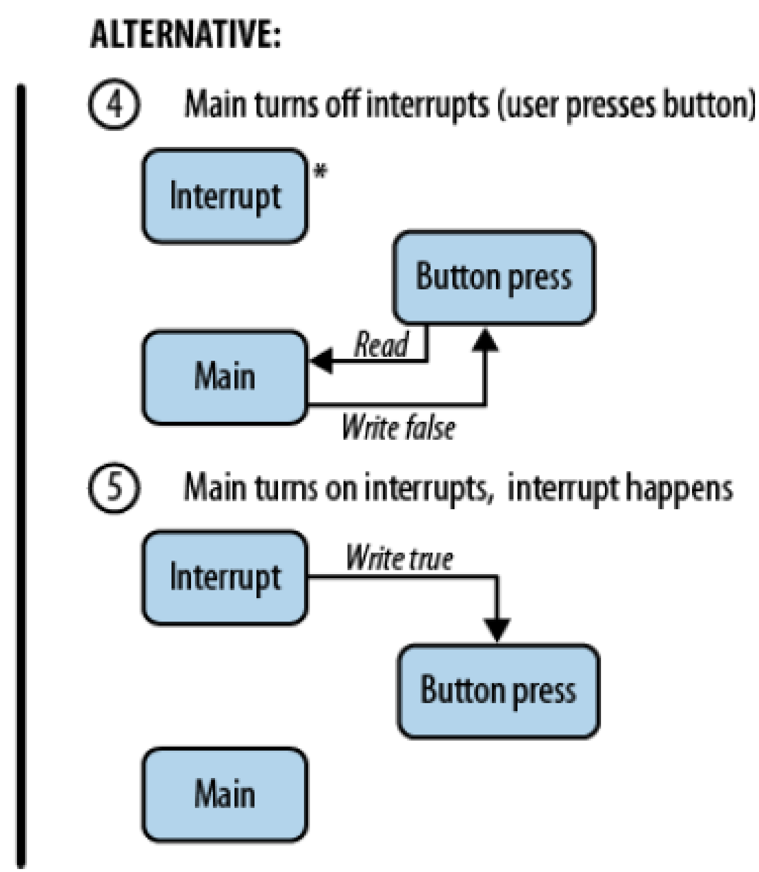
- Boot loader
  - A driver to start and initialize the whole system
- For ATmega128
  - Initialization
    - Set up the interrupt vector table
    - Clear SREG
    - Set stack pointer
    - Enable sleep
    - Initialize DATA and BSS
    - Configure on-board devices (SPI)
    - Enable interrupt
  - **Start app (main)**
    - **Setup**
    - **Get into the while loop**

Problems in the While Loop

- Multi-task + Interrupts
  - Multiple components work simultaneously.
  - Tasks share the same resource: processor, ...
  - Tasks communicate among each other.
  - Interrupts change the flow of code.
- Race condition
  - Tasks write to the shared memory at the same time and lead to an uncertain result.
  - Any memory shared between tasks can exhibit the uncertainty, leading to unstable and inconsistent behavior.

- Case 20


1 User pressed button
Interrupt
Write true
Button press
Main
2 Main handles button press
Interrupt
Button press
Read
Main
Write false
3 User pressed button
Interrupt
Write true
Button press
Main
4 Main handles button, but user pressed button
Interrupt
Write true
Button press
Main
Write false
Race to see which write wins


ALTERNATIVE:
4 Main turns off interrupts (user presses button)
Interrupt
*
Button press
Read
Main
Write false
5 Main turns on interrupts, interrupt happens
Interrupt
Write true
Button press
Main

Reentrant Functions

- Race condition could be triggered by calling non-reentrant functions.
- A reentrant function is a function where there is a provision to
  - interrupt the function in the course of execution,
  - service the interrupt service routine, and then
  - resume the earlier going on function,
  - without hampering its earlier course of action.
- Non-reentrant functions
  - Have static or global variables
  - Some C functions: malloc, printf (global variables)

Typical Solutions

- Mutex : mutual exclusion
  - The critical resource is protected so only one task can modify it at a time.
  - But, mutex requires that tasks cannot be interrupts.
    - Either the mutex is interrupted,
    - Or the interrupt is blocked by the mutex.
- Atomic : an action that cannot be interrupted.
  - When accessing the critical resource, interrupts are disabled.
  - But, interrupt latency may be increased or interrupts may be missed.
- Potential lockup

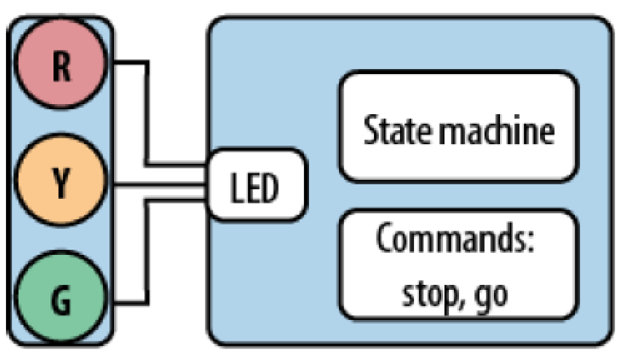
Better Solutions

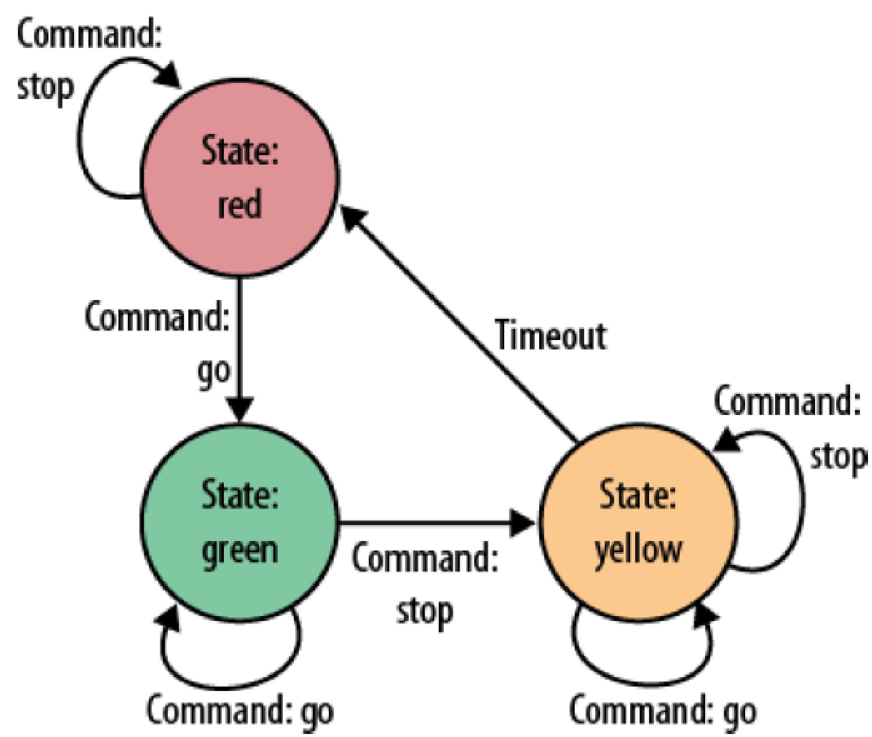
- Finite state machine
  - Separate app variables from interrupt variables
    - App variables: states
    - Interrupt variables: events
  - App only READs interrupt variables
  - Interrupt do NOT reads or writes app variables
- Scheduler
  - Use a task queue
    - Interrupt adds new tasks to queue
    - App removes and executes tasks in queue
  - Queue operations are ATOMIC
  - Both interrupt and app have BOUNDED queue operation time.

State Machine

- Coding template

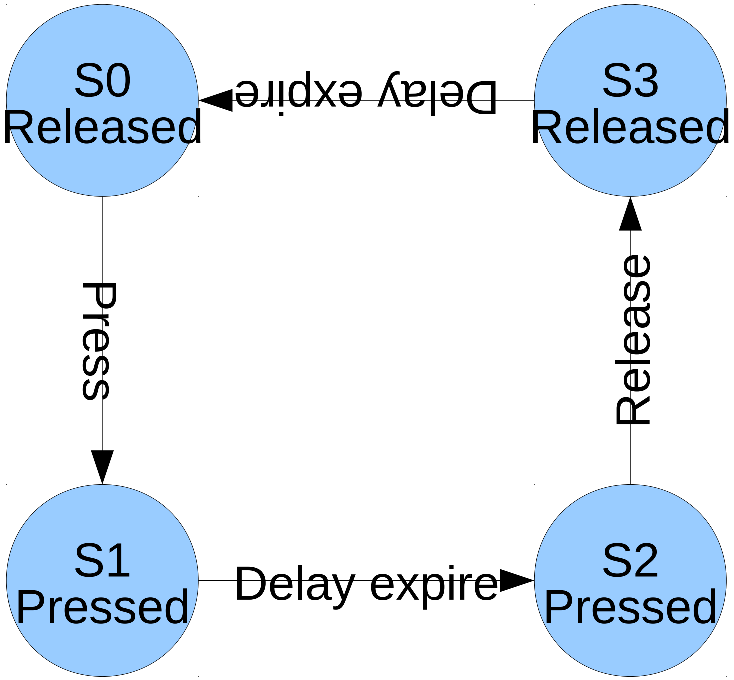
```
While (1) {
   Look for event;
   Switch-cases;
}
```





State Machine

- Events:
  - Press
  - Release
  - Delay expire
- States:
  - S0: released
  - S1: pressed
  - S2: pressed
  - S3: released



State Machine 1

- State-centric (** preferred **)
  - Template

```
case (state):
  if event valid for this state
    handle event
    prepare for new state
    set new state
```

  - Example

```
switch (state) {
case (green light):
  if (event is stop command)
    turn off green light
    turn on yellow light
    set state to yellow light
    start timer
  break;
```

State Machine 1

- Case 17, Pic24ep
  - Modules: button, led, delay, app
    - Button:
      - Polling
      - Not interrupt: event is triggered by button status not by button change
    - Delay:
      - Delay function
      - Timer interrupt
  - States: S0, S1, S2, S3
  - Events
    - Button_Pressed
    - Button_Released
    - Delay_Expired

State Machine 1

- Coding tips
  - Should have a .h for events and a .h for states.
  - Should enum events for each event modules.
  - Event modules (button and delay)
    - They shall not read or write states.
    - They only generate events regardless current states.
  - State machine shall be independent to the implementation of event modules.
  - Each module (except main) shall have a .h file and a .c file.

State Machine 2

- State-centric with Hidden Transitions
  - Template

```
case (state):
  make sure current state is actively doing what it needs
  if event valid for this state
    call next state function
```

  - Example

```
case (green light):
  turn on green light (even if it is already on)
  if (event is stop)
    turn off green light
    call next state function
  break;
```

State Machine 3

- Event-centric
  - Template

```
case (event):
  if state transition for this event
    go to new state
```

  - Example

```
switch (event)
case (stop):
  if (state is green light)
    turn off green light
    go to next state
  // else do nothing
  break;
```

State Machine 4

- Table-driven (** preferred **)

State machine engine **Table data**

STATES	ACTION Light	EVENTS Go	Stop	Time-out
RED	red	GREEN	RED	RED
YELLOW	yellow	RED	YELLOW	RED
GREEN	green	GREEN	YELLOW	GREEN

Current state →

Event "go"
New current state = GREEN

```
struct sStateTableEntry {
  tLight light;       // all states have associated lights
  tState goEvent;     // state to enter when go event occurs
  tState stopEvent;   // ... when stop event occurs
  tState timeoutEvent;// ... when timeout occurs
};
```

State Machine 4

- Table-driven (** preferred **)
  - Example is not very correct.

```
typedef enum { kRedState = 0, kYellowState = 1, kGreenState = 2 } tState;

struct sStateTableEntry stateTable[] = {
  { kRedLight,    kGreenState,  kRedState,    kRedState},   // Red
  { kYellowLight, kYellowState, kYellowState, kRedState},   // Yellow
  { kGreenLight,  kGreenState,  kYellowState, kGreenState}, // Green
}

void HandleEventGo(struct sStateTableEntry *currentState){
  // turn off the light (unless we're just going to turn it back on)
  if (currentState->light != currentState->go.light) {
    LightOff(currentState->light);
  }
  currentState = currentState->go;
  LightOn(currentState->light);
  StartTimer();
}
```

State Machine 4

- Case 18, Pic24ep

States	Events		
	pressed	released	expired
S0	S1 ledon startdelay	S0	S0
S1	S1	S1	S2
S2	S2	S3 ledoff startdelay	S2
S3	S3	S3	S0

State Machine 4

- Coding tips
  - Event centric
  - Each event transition includes
    - Next state
    - Activity
  - Each state table row includes
    - Current state
    - All possible event transitions
  - State table must be initialized before the while loop.
  - All events must be handled in the while loop.

Scheduler

- Issues with state machine
  - Best for states with single event transition.
  - When states have multiple event transitions and multiple events occur, what is the order to handle events?
- Ideas: when multiple events occur concurrently, we need to handle them one by one.
- So, need an event queue and a scheduler to dispatch events in the queue.
- Each event is a task.

Scheduler

- Coding tips
  - Nested loops in main
    - Outside loop: while(1)
    - Inside loop: scheduler
    - Independent to queue structure and operations
  - Scheduler is implemented according to application's requirements.
  - Pop and append operations
    - must run atomically
    - must run with deterministic time
  - The inside loop executes tasks when there are tasks in queue.

List-based Queue

- Data structure
  - List (single linked or double-linked)
- Operations
  - initQueue: initialize queue
  - isEmpty: check if queue is empty
  - append: add a new event to queue
  - pop: retrieve the next event in queue and remove the event from the queue
- Priority
  - First in first out
  - Take OS class for other priorities
- Problem
  - Heap allocation
  - Unlimited length
  - Non-deterministic time

Set-based Queue

- Case 19, Pic24ep
- Data structure
  - Array-based set
  - Fixed size, allocated in data section
- Operations
  - initQueue: initialize queue
  - isEmpty: check if queue is empty
  - append: add a new event to queue
  - pop: retrieve the next event in queue and remove the event from the queue
  - Deterministic operation time
- Priority
  - First in first out
- Problem
  - Any event cannot be repeated before being processed.

Interrupts

- Interrupt procedure
  - An interrupt request (IRQ) happens inside the processor, triggered by a peripheral, the software, or a fault.
  - The processor saves the context.
  - The processor looks in the interrupt vector table to find the callback function (ISR) associated with the interrupt.
  - The call back function runs.
  - The processor restores the context and resume from where it was interrupted.

Interrupts

- Priority
  - Determine which ISR to be called when multiple interrupts occur
  - Determine whether an interrupt can supersede the one currently running.
- Nonmaskable
  - Cannot be disabled
  - NMIs
    - Processor exception
    - Reset

IRQ

- Interrupts must be configured and enabled.
  - By compiler's startup code
  - By the software
  - Only enable what are needed.
- Setup
  - Disable interrupts first.
  - Configure interrupt control registers.
  - Example
    - Atmega128
    - Pic24ep512ug810

Interrupt Latency

- Context latency and ISR latency
  - Example
    - 30MHz processor
    - 44100Hz interrupt to handle audio
    - 680 cycles between two interrupts
    - 20 cycles total of switching context
    - 345 cycles total of ISR
    - 2.94% of time is for switching context
    - 50.5% of time for processing interrupts
  - Rules of reducing interrupt overhead
    - Only save the registers that are written by the ISR
    - Only use inline functions
    - Only process data that must be processed

ISR

- Initialize the interrupt vector table
  - The start up code (bootloader) sets up the IVT.
- Call IST
  - Make IST short to reduce latency
  - Do not call nonreentrant functions
  - Turn off other interrupts (flawed vs correct)

```
HandyHelperFunction:
  disable interrupts
  do critical things
  enable interrupts

CriticalFunction:
  disable interrupts
  call HandyHelperFunction
  do supposedly critical things
  enable interrupts
```

```
HandyHelperFunction:
  interrupt status  = disable interrupts()
  do critical things
  enable interrupts(interrupt status)

CriticalFunction:
  interrupt status = disable interrupts()
  call HandyHelperFunction
  do critical things
  enable interrupts(interrupt status)
```

ISR

- Disable
  - Save interrupt bit
  - Set interrupt bit to 0

```
mk=(1<<SREG_I);
st=SREG&mk;
SREG&=~mk;
```

- Enable
  - Restore interrupt bit

```
SREG|=st;
```

Pros and Cons of Interrupts

- Pros
  - A system is time-critical.
  - An event is expensive to check for and happens very rarely.
  - A short background task lets the system run more smoothly.
- Cons
  - Incur overhead
  - Make the system less deterministic
  - Hard to track down bugs
  - Need processor-dependent configuration

Non-Interrupts

- Polling
  - Last option to consider
  - Need a time-out in case polling does not get respond in time
- System Tick
  - For periodic applications
  - Must process everything during the interval of two ticks