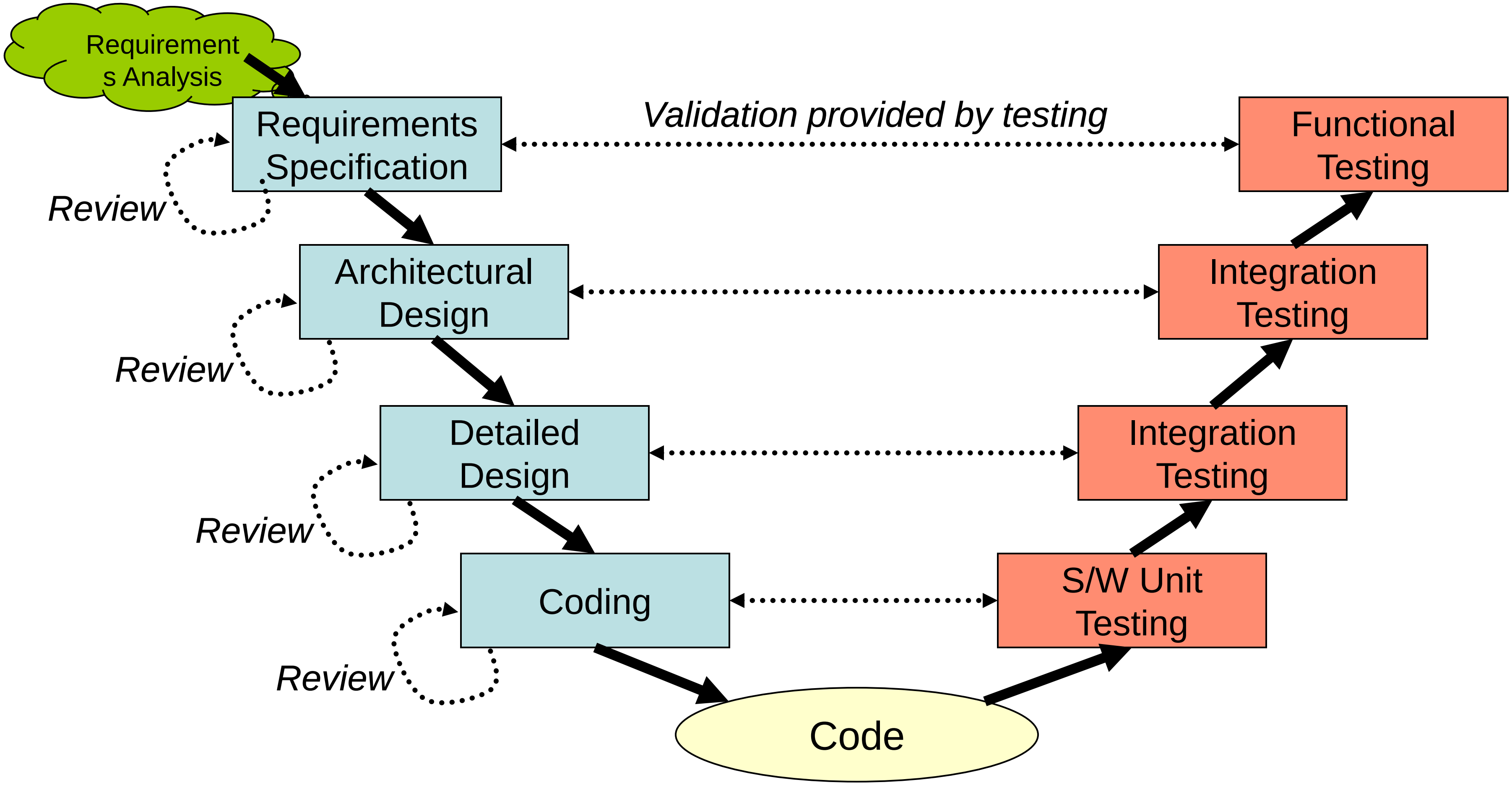
Introduction to Embedded Software Engineering

A Brief Introduction to Advanced Concepts

V Model Overview



Implementing the V

1. Requirements specification
   - Define what the system must do
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
4. Coding and Code Inspections

Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
   - Define big-picture view of what pieces will exist, how they will fit together, and how they will be tested
3. Detailed module and test design
4. Coding and Code Inspections

Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
   - Now we design the software, using flow charts or finite state machines
   - Also define module tests (discussed at end of lecture)
4. Coding and Code Inspections

Software Design Methods

Overview

- Software Goals
- Why *design* software before *coding* it?
- How should software be designed?
  - Pseudo-code
  - Flow charts
  - State machines
- How should software be coded (written)?

- Useful books
  - **The Practice of Programming**, Brian W. Kernighan & Rob Pike, Addison Wesley, 1999
  - **Real-Time Systems Development**, Rob Williams, Butterworth-Heinemann/Elsevier, 2006

Software Goals

- Simplicity – software is short and simple
- Clarity – software is easy for humans and machines to understand
- Generality – software can be used for a broad range of situations

Why Design First?

"He who fails to plan, plans to fail"

"Poor planning produces predictably poor performance"

- Software offers tremendous flexibility in implementing systems
  - Many methods work OK for small programs, but few of these work fine for large or real-time programs
  - Easy to choose a method which does not scale well to large programs
- Starting coding early forces designer to make implementation decisions early, before understanding impact on rest of system (and other programmers)
- Even in programs of moderate size, the details obscure the larger picture ("Can't see the forest for the trees")
  - "What are the independent processes within this system?"
  - "Who else can modify this variable?"
  - "How often will this function run?"
  - "How quickly will the system respond?"
- Companies which don't design their software before coding spend much more time and money debugging code, assuming they stay in business long enough to start selling the product

Software Design Representations

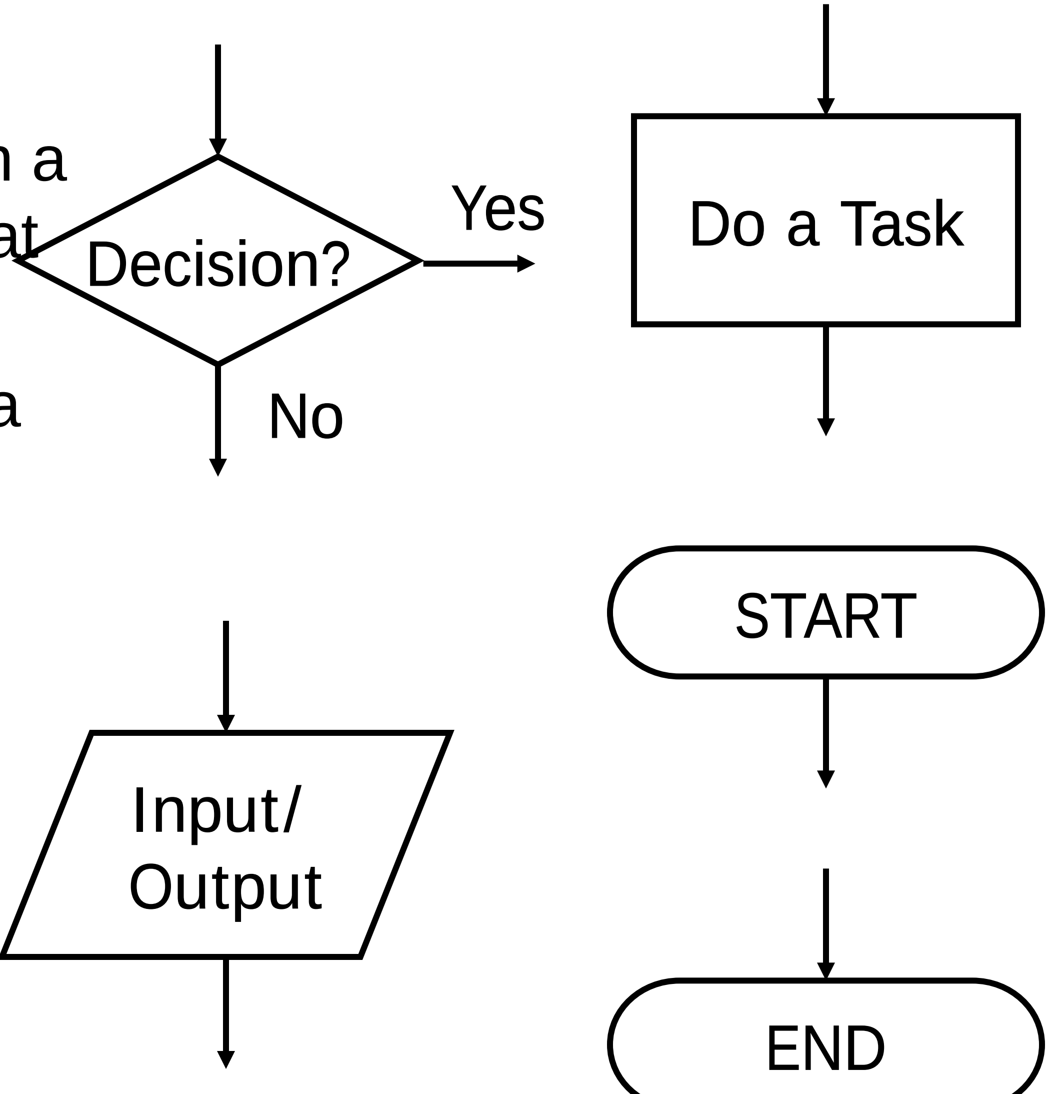
- Pseudocode
  - Easy to write but vague
- Flow Chart
  - Good for describing an algorithm: steps in processing, with conditional (if-else) and repeated (loop) execution
- State machine
  - Good for describing system with multiple states (operating modes) and transition rules

Pseudo Code

- Pseudo code is written in English to describe the functionality of a particular software module (subroutine)
- Include name of module/subroutine, author, date, description of functionality of module, and actual steps
- Often you can take the pseudo code and use them lines in your program as comments
- Avoid a very fine level of detail (although this may sometimes be difficult to do)
- Avoid writing code – use English, not assembly language (or higher-level language) instructions

Flowchart

- Shows flow of control in a processing activity (what gets done)
- Used to show steps in a process, including decision-making
- Does not scale well: becomes confusing if larger than a page



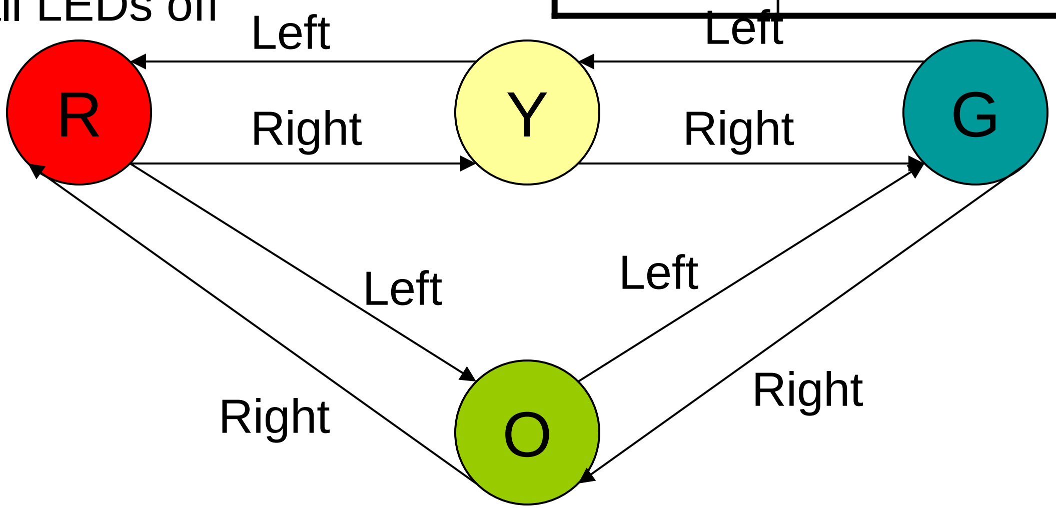
What is State-Based Behavior?

- System is in exactly one of multiple possible states. State:
  - time at which system is stable
  - with constant output conditions
  - awaiting valid trigger events
- A transition between states is triggered by a specific event or events (typically from an input)
  - Transitions may have associated activities (processing, output)
  - Guard conditions may prevent transition from occurring, despite event occurence
- Finite State Diagram (FSD) specifies all states and transitions
- Mealy vs. Moore
  - Mealy: activities occur while entering state. Outputs defined by state and transition event.
  - Moore: activities occur within state or while leaving state. Outputs defined only by current state.

Example: State Descriptions and Transitions

- Flash LEDs in sequence
  - Right: red-yellow-green-off
  - Left: green-yellow-red-off
- States
  - R: only red LED on
  - Y: only yellow LED on
  - G: only green LED on
  - O: all LEDs off

Current State	Next State (Direction= Left)	Next State (Direction= Right)
R	O	Y
Y	R	G
G	Y	O
O	R	G



How Should Software be Coded?

- Code has two requirements
  - To work
  - To communicate how it works to the author and others
    - After the code's author wins the lottery and quits the company, how hard do you want it to be to pick up the pieces?
- Variations in coding styles confuse the reader, so define two aspects of coding style to avoid variation
  - Syntax
  - Semantics
- So use a *Coding Standard or Style Guide* to define correct practices
  - Naming conventions
  - Memory allocation
  - Portability
  - ISRs
  - Comments
  - File locations
  - Eliminates arguments over minor issues

Example Coding Style Guidelines

1. Names
    1. Use descriptive names for global variables, short names for locals
    2. Use active names for functions (use verbs): Initialize_UART
    3. Be clear what a boolean return value means! Check_Battery vs. Battery_Is_Fully_Charged
2. Consistency and idioms
    1. Use consistent indentation and brace styles
    2. Use idioms (standard method of using a control structure): e.g. for loop
    3. Use else-if chains for multi-way branches
3. Expressions and statements
    1. Indent to show structure
    2. Make expressions easy to understand, avoid negative tests
    3. Parenthesize to avoid ambiguity
    4. Break up complex expressions
    5. Be clear: child = (!LC&&!RC)?0:(!LC?RC:LC); is not clear
    6. Be careful with side effects: array[i++] = i++;

Example Coding Style Guidelines

4. Macros
   1. Parenthesize the macro body **and** arguments

      #define square(x)   ((x) * (x))
5. Magic numbers
   1. Give names to magic numbers with either #define or enum

      #define MAX_TEMP (551)

      enum{ MAX_TEMP = 551, /* maximum allowed temperature */

      MIN_TEMP = 38,   /* minimum allowed temperature */ };
   2. Use character constants rather than integers: if ch==65 ???? if ch ==‘A’
   3. Use language to calculate the size of an object: sizeof(mystruct)
6. Comments
   1. Clarify, don’t confuse
   2. Don’t belabor the obvious
   3. Don’t comment bad code – rewrite it instead
   4. Don’t contradict the code

Example Coding Style Guidelines

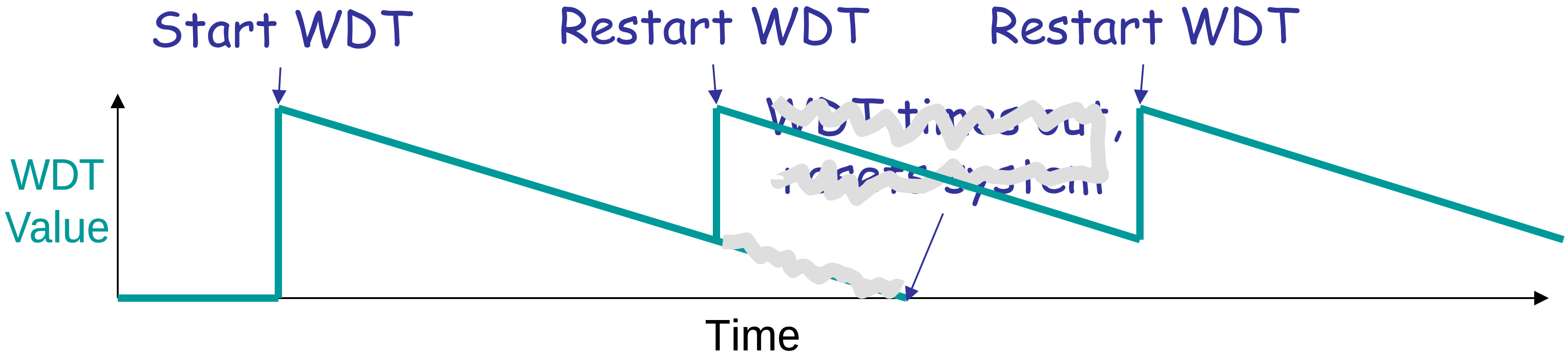
7. Use a standard comment block at the entry of each function
    1. Function Name
    2. Author Name
    3. Date of each modification
    4. Description of what function does
    5. Description of arguments
    6. Pre-conditions
    7. Description of return value
    8. Post-conditions
8. Defensive programming
    1. Upon entering a function, verify that the arguments are valid
    2. Verify intermediate results are valid
    3. Is the computed value which is about to be returned valid?
    4. Check the value returned by any function which can return an invalid value
9. No function should be more than 60 lines long, including comments.

Run-Time Methods for Making Embedded Systems Robust

Today

- Need to make embedded systems robust
  - Implementation flaws: Code may have implementation bugs
  - Design flaws: Real world may not behave the way we expected and designed for
  - Component failures: Sometimes things break
- Run-time mechanisms for robust embedded systems
  - Watchdog timer
  - Stack-pointer monitor
  - Voltage brown-out detector

Watchdog Timer Concepts (WDT)



- Goal: detect if software is not operating correctly
- Assumption: healthy threads/tasks will periodically send a heartbeat ("I'm alive") signal
- Mechanism
  - Use heartbeat signals from tasks to restart a timer
  - If timer ever expires, the system is sick, so reset
- Typically used as a final, crude catastrophic mechanism for forcing system software back into known state

Time-Out Actions

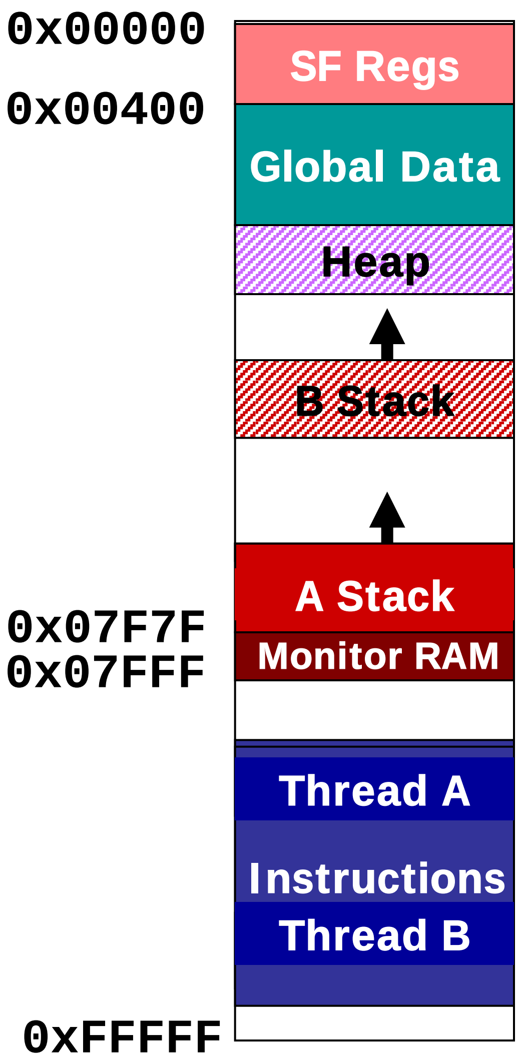
- Simple solution: reset entire system
  - May need to explicitly toggle reset pin to ensure CPU is fully reset (rather than just jumping to reset ISR)
  - Reset should configure all I/O to safe state
- NMI Solution: generate non-maskable interrupt for debug
  - Use NMI ISR to save picture of CPU and thread state
  - Can then examine what happened with debugger or in-circuit emulator
- WDT Time-Out flag in memory
  - Set flag upon time-out before reset
  - Examine this bit in reset ISR to determine whether to boot system normally or with debug mode (without overwriting RAM)

Mechanisms for robust embedded systems

- Watchdog timer
- Stack-pointer monitor
- Brown-out detector

Stack Pointer Monitor

- What makes the stack grow?
  - Nested subroutine calls – each adds 5 bytes (3 bytes for return address, 2 bytes for dynamic link)
    - Local data in the subroutine call – automatic variables
    - Arguments passed to the subroutine
  - Nested interrupt handling – each adds 4 bytes (3 bytes for return address, 1 byte for flag register)
    - Local storage for the interrupt
- How large does the stack get?
  - Starts at the top of RAM, grows to smaller addresses
  - Will overwrite heap or global data if gets too large
  - Need to allocate space for multiple stacks in system with a preemptive scheduler



Mechanisms for robust embedded systems

- Watchdog timer
- Stack-pointer monitor
- Voltage brown-out detector

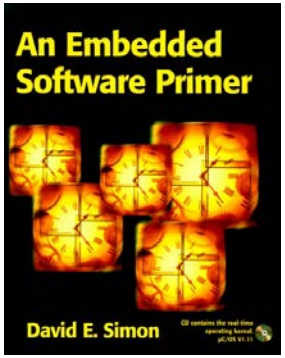
Voltage brown-Out Detector

- Black-out == total loss of electricity
- Brown-out == partial loss of electricity
  - Voltage is low enough that the system is not guaranteed to work completely
  - We can’t guarantee that it won’t do anything at all. Parts may still work.
    - “CPU runs, except for when trying to do multiplies”

- Want to detect brown-out automatically
  - Possibly save critical processor information to allow warm boot
  - Then hold processor in reset state until brown-out ends

Sharing the Processor: A Survey of Approaches to Supporting Concurrency

Today



- Topic - How do we make the processor do things at the right times?
  - For more details see Chapter 5 of D.E. Simon, **An Embedded Software Primer**, Addison-Wesley 1999
- There are various methods; the best fit depends on…
  - system requirements – response time
  - software complexity – number of threads of execution
  - resources – RAM, interrupts, energy available

Round-Robin/Super-Loop

- Extremely simple
  - No interrupts
  - No shared data problems
- Poll each device (**`if (device_A_ready())`**)
- Service it with task code when needed

```
void main(void) {
  while (TRUE) {
    if (device_A_ready()) {
      service_device_A();
    }
    if (device_B_ready()) {
      service_device_B();
    }
    if (device_C_ready()) {
      service_device_C();
    }

  }
}
```

Example Round-Robin Application

```
void DMM_Main(void) {
  enum {OHMS_1, ... VOLTS_100} SwitchPos;
   while (TRUE) {
        switch (SwitchPos) {
        case OHMS_1:
                ConfigureADC(OHMS_1);
                EnableOhmsIndicator();
                x = Convert();
                s = FormatOhms(x);
                break;
        ...
        case VOLTS_100:
                ConfigureADC(VOLTS_100);
                EnableVoltageIndicator();
                x = Convert();
                s = FormatVolts(x);
                break;
        }
        DisplayResult(s);
        Delay(50);
  }
}
```

Problems with Round-Robin

- Architecture supports multi-rate systems very poorly
  - Voice Recorder: sample microphone at 20 kHz, sample switches at 15 Hz, update display at 4 Hz. How do we do this?
- Polling frequency limited by time to execute main loop
  - Can get more performance by testing more often (A/Z/B/Z/C/Z/...)
  - This makes program more complex and increases response time for other tasks
- Potentially Long Response Time
  - In worst case, need to wait for all devices to be serviced
- Fragile Architecture
  - Adding a new device will affect timing of all other devices
  - Changing rates is tedious and inhumane

Event-Triggered using Interrupts

- Very basic architecture, useful for simple low-power devices, very little code or time overhead
- Leverages built-in task dispatching of interrupt system
  - Can trigger ISRs with input changes, timer expiration, UART data reception, analog input level crossing comparator threshold
- Function types
  - Main function configures system and then goes to sleep
    - If interrupted, it goes right back to sleep
  - Only interrupts are used for normal program operation
- Example: bike computer
  - Int1: wheel rotation
  - Int2: mode key
  - Int3: clock
  - Output: Liquid Crystal Display

Bike Computer Functions

Reset

```
Configure timer,
inputs and
outputs

cur_time = 0;
rotations = 0;
tenth_miles = 0;

while (1) {
  sleep;
}
```

ISR 1: Wheel rotation

```
rotations++;
if (rotations>
  R_PER_MILE/10) {
  tenth_miles++;
  rotations = 0;
}
speed =
 circumference/
 (cur_time – prev_time);
compute avg_speed;
prev_time = cur_time;
return from interrupt
```

ISR 2: Mode Key

```
mode++;
mode = mode %
   NUM_MODES;
return from interrupt;
```

ISR 3: Time of Day Timer

```
cur_time ++;
lcd_refresh--;
if (lcd_refresh==0) {
 convert tenth_miles
    and display
 convert speed
    and display
  if (mode == 0)
    convert cur_time
      and display
  else
    convert avg_speed
      and display
  lcd_refresh =
     LCD_REF_PERIOD
}
```

Problems with Event-Triggered using Interrupts

- All computing must be triggered by an event of some type
  - Periodic events are triggered by a timer
- Limited number of timers on MCUs, so may need to introduce a scheduler of some sort which
  - determines the next periodic event to execute,
  - computes the delay until it needs to run
  - initializes a timer to expire at that time
  - goes to sleep (or idle loop)
- Everything (after initialization) is an ISR
  - All code is in ISRs, making them long
  - Response time depends on longest ISR. Could be too slow, unless interrupts are re-enabled in ISR
  - Priorities are directly tied to MCU’s interrupt priority scheme

Round-Robin with Interrupts

- Also called foreground/background
- Interrupt routines
  - Handle most urgent work
  - Set flags to request processing by main loop
- More than one priority level
  - Interrupts – multiple interrupt priorities possible
  - **`main`** code

```
BOOL DeviceARequest, DeviceBRequest,
DeviceCRequest;
void interrupt HandleDeviceA(){
  /* do A's urgent work */
  ...
  DeviceARequest = TRUE;
}
void main(void) {
  while (TRUE) {
    if (DeviceARequest) {
      FinishDeviceA();
    }
    if (DeviceBRequest) {
      FinishDeviceB();
    }
    if (DeviceCRequest) {
      FinishDeviceC();
    }
  }
}
```

Problems with Round-Robin with Interrupts

- All task code has same priority
  - What if device A must be handled quickly, but `FinishDeviceC` (slow) is running?
  - Difficult to improve A's response time
    - Only by moving more code into ISR
- Shared data can be corrupted easily if interrupts occur during critical sections
  - Flags (`DeviceARequest`, etc.), data buffers
  - Must use special program constructs
    - Disable interrupts during critical sections
    - Semaphore, critical region, monitor
  - New problems arise – Deadlock, starvation

Real-Time Operating System (*RTOS, Kernel, ...*)

- As with previous methods
  - ISRs handle most urgent operations
  - Other code finishes remaining work
- Differences:
  - The RTOS can ***preempt*** (suspend) a task to run something else.
  - Signaling between ISRs and task code (service functions) handled by RTOS.
  - We don’t write a loop to choose the next task to run. RTOS chooses based upon priority.

Why These Differences Matter

- Signaling handled by RTOS
  - Shared variables not needed, so programming is easier
- RTOS chooses next task to run
  - Programming is easier
- RTOS can preempt tasks, and therefore schedule freely
  - System can control *task code response time* (in addition to interrupt routine response time)
  - Worst-case wait for highest-priority task doesn’t depend on duration of other tasks.
  - System’s response (time delay) becomes more stable
    - A task’s response time depends only on higher-priority tasks (*usually* – more later)

More RTOS Issues

- Many RTOS's on the market
  - Already built and debugged
  - Debug tools typically included
  - Full documentation (and source code) available
- Main disadvantage: RTOS costs resources (e.g. uC/OSII compiled for 80186. YMMV)
  - Compute Cycles: 4% of CPU
  - Money: ???
  - Code memory: 8.3 KBytes
  - Data memory: 5.7 KBytes

Comparison of Priority Levels Available