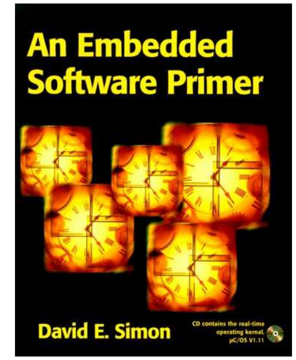
Sharing the Processor: A Survey of Approaches to Supporting Concurrency

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

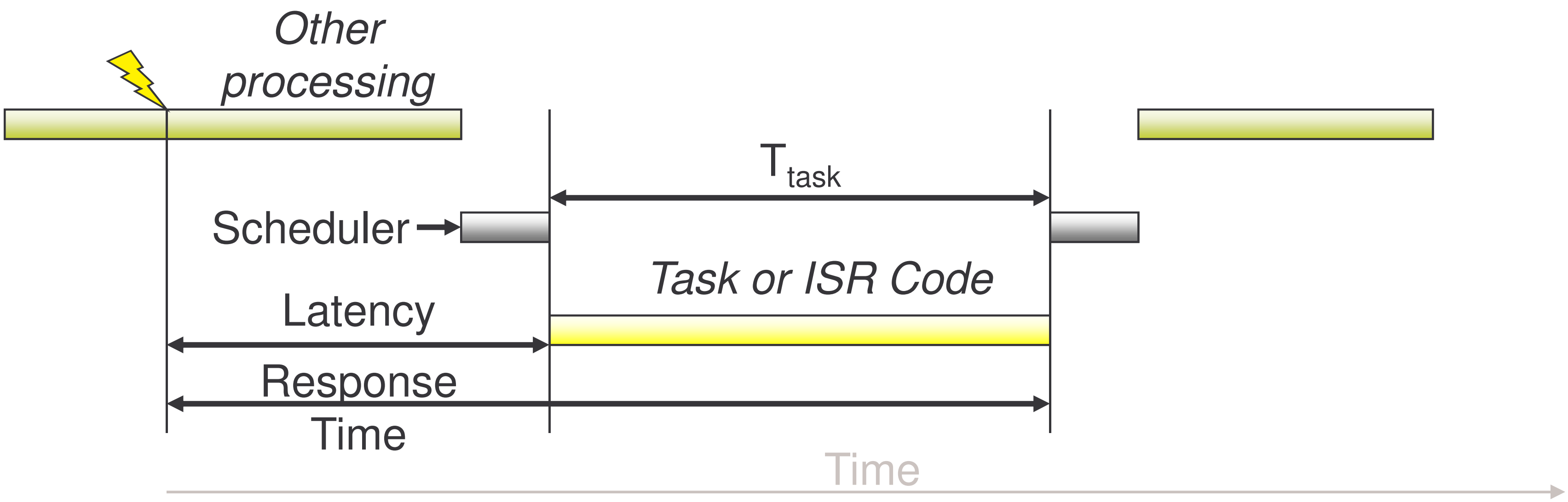
RTOS Cult De-Programming

Real-Time Operating System

- How do we schedule the tasks on the CPU?
- An infinite loop in main
- Real-time operating system
- Is there anything else available?


```
while (1) {
   ...
};
```

Definitions



- $T_{Release}(i)$ = Time at which task i is becomes ready to run
- $T_{response}(i)$ = Delay between request for service and completion of service for task i
- $T_{task}(i)$ = Time needed to perform computations for task i
- $T_{ISR}(i)$ = Time needed to perform interrupt service routine i

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);
   }
}
```

Sample Application - Network Videophone

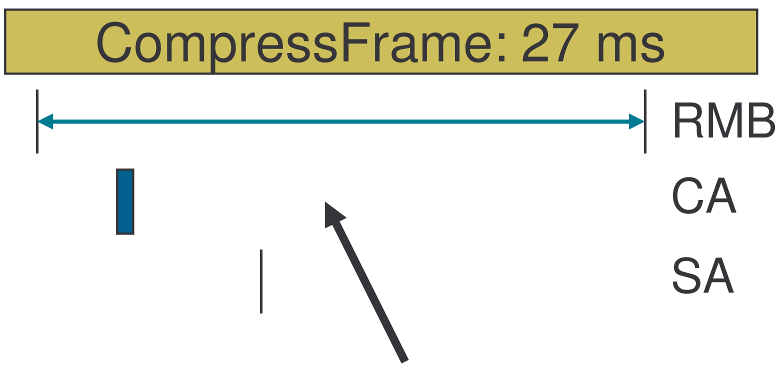
- Video
  - 30 frames/s
  - 360 x 240 images
  - Compress/ Decompress with MPEG-2
- Audio
  - 8 kHz sampling
  - Compress with GSM 06.10
- Processor
  - 3000 MIPS
- Tasks have deadlines

Service	Direction	Function	WCET	Deadline
Video	Send	SampleFrame	1 ms	33.3 ms
		CompressFrame	27 ms	33.3 ms
		SendFrame	0.1 ms	33.3 ms
	Receive	ReceiveFrame	0.1 ms	33.3 ms
		DecompressFrame	2.7 ms	33.3 ms
		DisplayFrame	1 ms	33.3 ms
Audio	Send	ReadMicBuffer	0.001 ms	20 ms
		CompressAudio	0.160 ms	20 ms
		SendAudio	0.001 ms	20 ms
	Receive	ReceiveAudio	0.001 ms	20 ms
		DecompressAudio	0.160 ms	20 ms
		LoadAudioBuffer	0.001 ms	20 ms

Scheduling NV with Round-Robin

Round robin works for either video or audio, but not both

Need to split up video **`CompressFrame()`**



All audio tasks: Deadline is 20 ms from beginning of first task

```
void main() {
  while(TRUE) {
    if (TimeToSample) {
     SampleFrame();
     CompressFrame();
     SendFrame();
    }
    if (FrameWaiting) {
     ReceiveFrame();
     DecompressFrame();
     DisplayFrame();
    }
  }
}
```

Limitations of 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

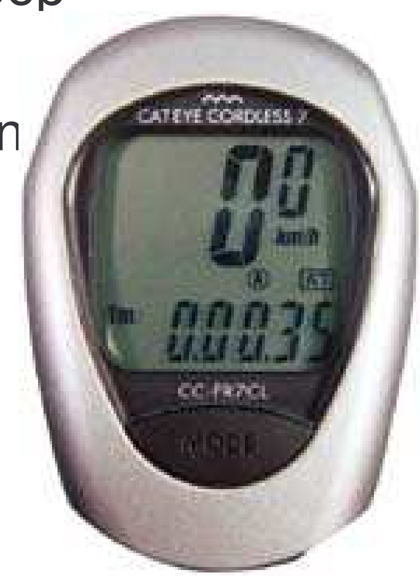
- In worst case, need to wait for all devices to be serviced
- $$\max\left(T_{response}(j)\right) = \sum_{\forall t} T_{task}(t)$$

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
}
```

Limitations of 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

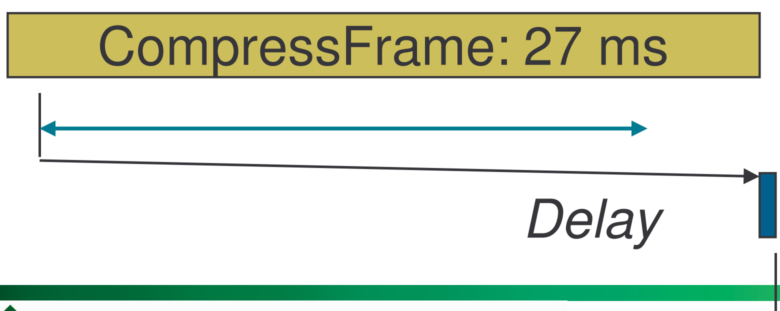
- Interrupts – multiple interrupt priorities possible
- **main** code

```
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();
    }
  }
}
```

Scheduling NV with Round Robin + Interrupts

```
BOOL ReadMicBuffer_Req = FALSE,
  SampleFrame_Req = FALSE;

interrupt void HandleMicBuffer()
{
  copy contents of mic buffer
  ReadMicBuffer_Done = TRUE;
}
interrupt void
HandleSampleFrame() {
  Sample a frame of video
  SampleFrame_Done = TRUE;
}
```




```
void main(void) {
  while (TRUE) {
    if (ReadMicBuffer_Done) {
       CompressAudio();
       SendAudio();
       ReadMicBuffer_Done=FALSE;
    }
    if (SampleFrame_Done) {
       CompressFrame();
       SendFrame();
       SampleFrame_Done = FALSE;
    }
    etc.
  }
}
```

Limitations of Round-Robin with Interrupts

All task code has same priority

- What if device A must be handled quickly, but **`FinishDeviceC`** (slow) is running?
- $$\max\left(T_{response}(j)\right) = \sum_{\forall t} T_{task}(t) + \sum_{\forall i} T_{ISR}(i)$$
- 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

Run-To-Completion Scheduler

- Use a ***scheduler*** function to run task functions at the right rates
  - Table stores information per task
    - Period: How many ticks between each task release
    - Release Time: how long until task is ready to run
    - ReadyToRun: task is ready to run immediately
  - "round-robin" scheduler runs forever, examining schedule table which indicates tasks which are ready to run (have been "released")
  - A periodic timer interrupt triggers an ISR, which updates the schedule table
    - Decrements "time until next release"
    - If this time reaches 0, set that task's Run flag and reload its time with the period
- Follows a "run-to-completion" model
  - A task's execution is ***not interleaved*** with any other task
  - Only ISRs can interrupt task
  - After ISR completes, the previously-running task resumes
- Priority is determined by position in table. Hard to change dynamically

RTC Scheduler App Programmer's Interface

API enables control of tasks at more efficient level

- Add Task(task, time period, priority)
  - task: address of task (function name without parentheses)
  - time period: period at which task will be run (in ticks)
  - priority: lower number is higher priority. Also is task number.
  - automatically enables task
- Remove Task(task)
  - removes task from scheduler.
- Run Task(task number)
  - Signals the scheduler that task should run when possible and enables it
- Run RTC Scheduler()
  - Run the scheduler!
  - Never returns
  - There must be at least one task scheduled to run before calling this function.
- Enable_Task(task_number) and Disable_Task(task_number)
  - Set or clear enabled flag, controlling whether task can run or not
- Reschedule_Task(task_number, new_period)
  - Changes the period at which the task runs. Also resets timer to that value.

Limitations of Run-To-Completion Scheduler

Tasks run to completion – problem with long tasks

- Maximum response time for a task is the duration of the longest task
- Long tasks complicate programming
  - No elegant way to start an operation (e.g. flash programming) and yield processor for 10 ms
  - Can improvise
    - Trigger another task
    - Use a state machine within this task

Prioritization implies unfair processor allocation – starvation possible

Function-Queue Scheduling

Interrupt routine enqueues a function to be called by **`main`**

Queue provides scheduling flexibility

- Functions can be enqueued with any order desired
- Use priority of device to determine position in queue

```
void interrupt HandleDeviceA(){
  /* do urgent work for A */
  ...
  Enqueue(Queue,FinishDeviceA);
}
...
void FinishDeviceA(void) {
  /* do remainder of A's work */
}

void main(void) {
  while (TRUE) {
    while (NotEmpty(Queue)) {
      f = Dequeue(Queue);
      f();
    }
  }
}
```

Limitations of Function-Queue Scheduling

- What if a long lower-priority function (**`FinishDeviceC`**) is executing and we need to run **`FinishDeviceA`**?
  - Must wait until **`FinishDeviceC`** completes
  - $\max\left(T_{response}(j)\right) = \max\left(T_{task}(t)\right)\forall t + \sum_{\forall i} T_{ISR}(i)$
  - Cooperative multitasking, no pre-emption
- What if the lowest-priority functions never get to run?
  - Heavily loaded system

Real-Time OS (*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

High

↕

Low

Round-Robin	*Round-Robin + Interrupts*	*Function-Queue, RTC and RTOS*
All Code	Device A ISR	Device A ISR
	Device B ISR	Device B ISR
	Device ... ISR	Device ... ISR
	Device Z ISR	Device Z ISR
	All Task Code	Task 1 Code
		Task 2 Code
		Task 3 Code
		Task 4 Code
		Task 5 Code
		Task 6 Code

Software Architecture Characteristics

	Priorities Available	Worst Case $T_{Response}$ for Highest Priority Task Code	Stability of $T_{Response}$ when Code Changes	Simplicity
Round-robin	None	ΣT_{Task}	Poor	Very simple
Round-robin with interrupts	Prioritized interrupt routines, then task code at same priority	$\Sigma T_{Task} + \Sigma T_{Interrupt}$	Good for interrupts, poor for task code	Must deal with shared data (interrupts/tasks)
RTC and Function-queue scheduling	Prioritized interrupt routines, then prioritized task code	$\max(T_{Task}) + \Sigma T_{Interrupt}$	Relatively good	Must deal with shared data and must write/get scheduler code
Real-time operating system	Prioritized interrupt routines, then prioritized task code	$\Sigma T_{Interrupt} + T_{OS}$	Very good	Most complex (much is handled by RTOS)