Lecture: Embedded Software Architectures

Jan Vitek

ECE/CS

Spring 2011

Reading List

- Mandatory Reading
  - Chapter 5 of ECS textbook

- Optional Reading
  - *N/A*

Software Architecture

- A *software architecture* gives the general structure of an embedded application independent of the actual computation performed
- Choice of architecture impacts issues such as:
  - development time / likelihood of software defects
  - responsiveness and latency
  - code size / complexity
- Rule of thumb:
  - *Select simplest architecture that meets application requirements*
  - Any extraneous complexity/generality costs additional development and verification effort

Software Architectures

- Four well known choices:
  - Simple Round Robin
  - Round Robin with Interrupts
  - Round Robin with Interrupts and Function Queues
  - Real-time Operating System-based architectures

- The architectures are sorted in order of increasing complexity

- Round Robin (RR) architectures are also called *Cyclic Executives* in real-time literature

- The main different between RR and RTOS-based approaches is that in RR scheduling and admission control is done by the developer as opposed to leaving it to the OS

Round Robin

- Simplest architecture, a single loop checks devices in predefined sequence and performs I/O right away

```
while(1) {
   if (device_1_ready()) {  /*Perform D1 I/O and relate computation.*/ }
   if (device_2_ready()) {  /*Perform D2 I/O and relate computation.*/ }
   …
   if (device_N_ready()) {  /*Perform DN I/O and relate computation.*/ }
}
```

- Works well for system with few devices, trivial timing constraints, proportionally small processing costs
- Response time of device *i* equal to WCET of the body of the loop

Round Robin

- Periodic Round Robin
  - In case the system must perform operations at different frequencies
  - Add code to wait a variable amount of time

```
while(1) {
   waitForNextPeriod(10);  // idle for up to 10 ms
   if (device_1_ready()) {  /*Perform D1 I/O and relate computation.*/ }
   ...
```

- Exercise:
  - *Think of how to implement a loop that runs every 10 ms and measures the drift*

Round Robin

- Limitations of the architecture:
  - If some devices require small response times, while other have large WCET it will not be possible to guarantee that all timing constraints will be met
  - The architecture is fragile, adding a new task can easily cause missed deadlines
  - Question:
    - Is the order in which devices appear significant?
    - Same question, but with code for devices having different processing times and timing constraints?

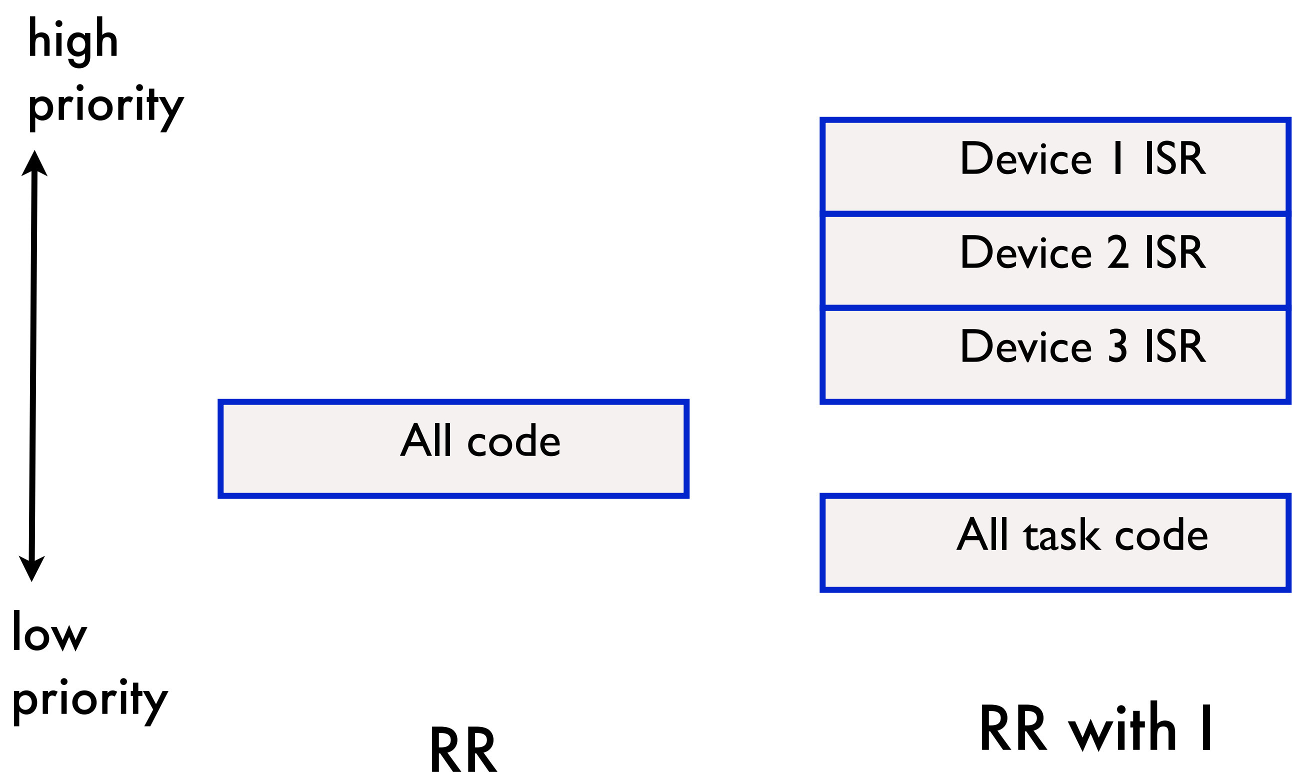
Round Robin with Interrupts

- Hardware events requiring small response times handled by ISRs
- Typically ISRs do little more than set flags and copy data

```
bool f_device_1 = FALSE;
bool f_device_1 = FALSE;
          …
void interrupt handle_dev_1() {
    // handle device 1
    f_device_1 = TRUE;
}
          …
void main() {
  while (1) {
     if(f_device_1) {
       f_device_1 = FALSE;
       // do processing related to device 1…
     if (f_device_2) {
          …
  }
}
```

Round Robin with Interrupts

- Latency of an ISR is function of response time of higher priority ISRs
- Lower bound on latency of RR loop is response time of the ISRs



Round Robin with Interrupts

- Drawbacks
  - All task code executes at same priority
    - One can test some flags multiples times within loop body to reduce latency
  - Shared data bugs
  - Question:
    - *What if one of the device requires large amount of processing time (larger than the time constraint of others?)*

RR+I and Function Queues

- Rather than fixed order, program manages order of execution

```
#define DEV_1 1
#define DEV_2 2
…
void interrupt handle_DEV_1() {
    // deal with device
    enqueue(DEV_1);
}
…
void main() {
  while (1) {
   switch (dequeue()) {
     case DEV_1:   // process DEV_1
          break;
      …
     default:    // empty queue nothing to do
   }
  }
}
```

RR+I and Function Queues

- One could use function pointers, but they add complexity
  - *FP are useful if one does not want to hardwire the devices in the main loop*
- enqueue() reorders queue to improve latency of high priority devices
- For long running functions: break them up into multiple smaller units
  - Question: *does that improve latency?*

- Question
  - *Consider implementation of dequeue(), what kind of data structure would you use (why), is special care needed?*

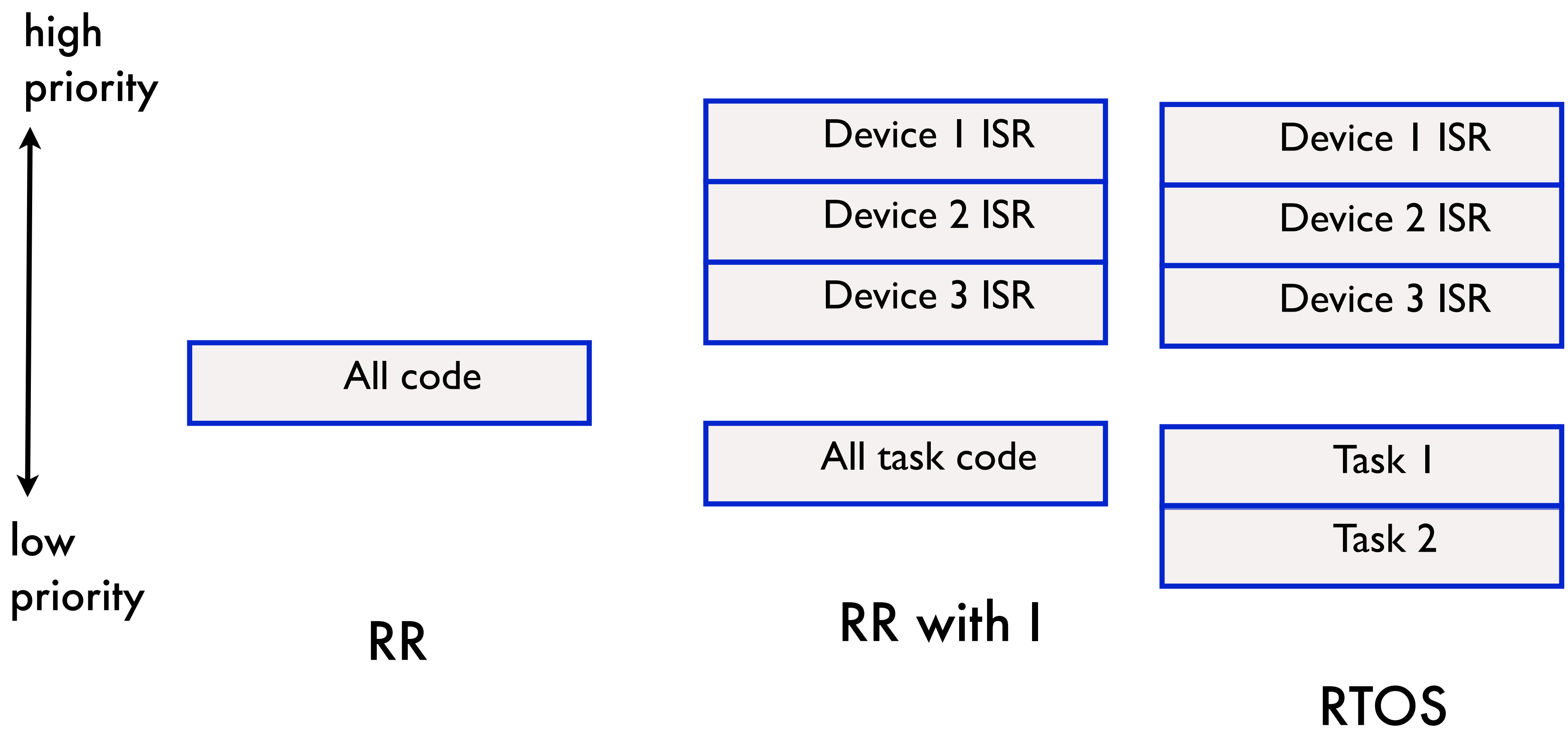
Real-time Operating Systems

- Rely on the operating for scheduling tasks
- Leverage preemptive scheduling to ensure that deadlines are met

```
static pthread_t thread_1;
…
void interrupt handle_DEV_1() {
  // handle device
  CHECK( pthread_wakeup(thread_1) );
}
…
void task_1() {
  while (1) {
    pthread_suspend_np();
    // process device 1 I/O
  }
}
…
```

Real-time Operating Systems

- The scheduler in a RTOS takes care of scheduling all tasks according to their priority
- Long running, low priority, tasks can be preempted by higher priority ones



Real-time Operating Systems

- Services that an RTOS *could* provide:
  - Scheduling tasks
    - create/terminate threads
    - timing threads operations
    - preemption
  - Synchronization
    - semaphores and locks
  - Input/Output
    - interrupt handling
  - Memory management
    - separate stacks
    - segmentation
    - allocation/deallocation
  - File system
    - persistent store
  - Security
    - user vs. kernel space
    - identity management

Conclusion

- Software architectures describe the structure of a system independently of its function
  - *Round Robin* is a simple architecture for devices with few (or uniform) timing constraints
  - *Round Robin with Interrupts* extends RR with low-latency interrupt handling
  - *Round Robin with Interrupts and Function Queues* allows dynamic scheduling of tasks under programmatic control
  - *Real-time Operating Systems* relieve programmers from having to deal with scheduling and time management