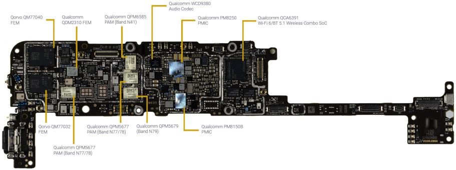


Embedded Software Architectures

Kizito NKURIKIYEYEZU, Ph.D.

Introduction

- This lecture will discuss various architectures for embedded software—the basic structures that are used to put together an embedded system software.
- The best architecture depends on several factors:
  - Real-time requirements of the application (absolute response time)
  - Available hardware (speed, features)
  - Number and complexity of different software features
  - Number and complexity of different peripherals
  - Relative priority of features
- Thus, each software architecture is tradeoff between complexity and control over response and priority

Choosing the best software architecture

- When designing an embedded software, what is the most optimum software architecture to use for a given system?
- The best architecture depends on several factors
  - Real-time requirements of the application (absolute response time)
  - Available hardware (speed, features)
  - Number and complexity of different software features
  - Number and complexity of different peripherals
  - Relative priority of features
- The decision is based on the tradeoff between complexity and control over response and priority:
  - Systems that require little control and poor response can be done with simple architectures
  - Rapid response systems will require more complex program design to be successful.

Example 1 —Air conditioning

- This system can be written with a very simple software architecture.
- The response time can be within a number of tens of seconds.
- The major function is to monitor the temperature readings and turn on and off the air conditioner.
- A timer may be needed to provide the turn-on and turn-off time.

Example 2 —Office telephone with Speaker

Consider a digital telephone answering machine with speech compression. It performs the following operations

- Records about 30 minutes of total voice sampled at 8kHz
- The software design for the answering machine
  - It must respond rapidly to many different events.
  - It has restrictive and various processing requirements.
  - It has different deadlines and different priorities

Example 2 —Office telephone with Speaker

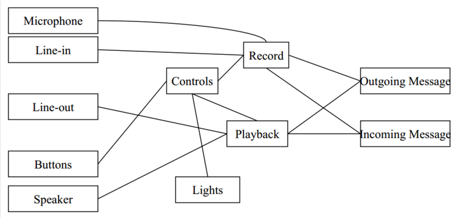
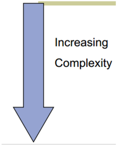


FIG 1. Simplified class diagram of the office telephone

Basic RT software architectures

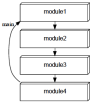
- Round-Robin
- Round-Robin with Interrupts
- Real-Time Operating System



Round Robin

Round Robin

- Simplest architecture
- No interrupts
- Main loop checks each device one at a time, and service whichever needs to be serviced.
- Service order depends on position in the loop.
- No priorities
- No shared data
- No latency issues (other than waiting for other devices to be serviced




```
void main(void) {
   while (TRUE) {
      module1();
      module2();
      module3();
      module4();
   }
}
```

FIG 2. Round Robin[1]

[1]Baier, M. (2014). Embedded software development in research

Round Robin Architecture

```
void main(void) {
  while (true) {
    if (Device_A_needs_service()){
      //Service device A
    }
    if (Device_B_needs_service()){
      //Service device B
    }
    if (Device_C_needs_service()){
      //Service device C
    }
    // Etc...
  }
}
```

LISTING 1: Round Robin Architecture

Round-Robin architecture—Pros and cons

Advantages:

- Simple solution, but sufficient for some applications.
- Exchanging data between tasks is easy.

Drawbacks:

- The worst-case latency of an external request is equal to the execution time of the entire main loop.
  - Architecture fails if any one device requires a shorter response time
  - Most I/O needs fast response time (buttons, serial ports, etc.)
- Implementing additional features can adversely affect the correctness of a system, by increasing latencies beyond acceptable bounds.
- Architecture is fragile to added functionality: adding one more device to the loop may break everything

Example —A digital multimeter

- This uses a round-robin works well for this system because:
  - only 3 I/O devices
  - no lengthy processing
  - no tight response requirements
  - small delays in switch position changes will go unnoticed
- No emergency control
  - No such requirements
  - Users are unlikely to notice the few fractions of a second it takes for the microprocessor to get around the loop
- Adequate because it is a SIMPLE system!
  - Simple devices such as watches, simple microwave ovens, toys, vending machine etc
  - Devices where operations are all user initiated and process quickly
  - Anything where the processor has plenty of time to get around the loop, and the user won't notice the delay

Example —digital multimeter

```
void vDigitalMultiMeterMain (void)
{
    enum {OHMS_1, OHMS_10 VOLTS_100} eSwitchPosition;
    while (TRUE)
    {
        eSwitchPosition = // Read the position of the switch;
        switch (eSwitchPosition)
        {
            case OHMS_1:
                  // Read hardware to measure ohms Format result
            break;
            case OHMS_10:
                 //Read hardware to measure ohms
                 // Format result
            break;
            case VOLTS_100:
                  //Read hardware to measure volts
                  // Format result
            break;
        }
        // Write result to display
    }
}
```

FIG 3. **Digital multi-meter**—It is possible to use a round-robin architecture because its users cannot expect faster response than they can move their hands and the probes

Summary —Round robin architecture

- This is the simplest architecture devoid of interrupts or shared-data concerns
- However several problems arise from its simplicity:
  - If a device has a response time constraints this architecture has problems (e.g. if in the example device Z has a deadline of 15 ms and A and B take 10 ms each.)
  - If any one of the cases at the worst take 5 seconds, the system would have a max. response time of 5 seconds, which would make it less appealing.
  - Architecture is not robust. Addition of a single device might cause all deadlines to be missed.

Round-robin with interrupts

Round-robin with interrupts

- Allows some control of software execution
- Gives more control over priorities.
- Based on Round Robin, but interrupts deal with urgent timing requirements.
- Interrupts a) service hardware and b) set flags
- Main routine checks flags and does any lower priority follow-up processing.


```
void main(void) {
  while (TRUE) {
    module1();
    module2();
    module3();
    if (irq1_do_calc) irq1AddCalc();
    if (irq2_do_calc) irq2AddCalc();
  }
}
void IRQ1_ISR(void) {
   ...
   irq1_do_calc = true;
}
```

FIG 4. Round robin with interrupts

[1] Bajer, M. (2014). Embedded software development in research environment: A practical guide for non-experts. Proceedings - 2014 3rd

Round-robin with interrupts

Principles: Tasks are invoked in round-robin fashion, but interrupt routines take care of urgent operations

- A little bit more control
  - In this architecture, interrupt service routines (ISR) deal with the very urgent needs of the hardware and set corresponding flags
  - Interrupt routines set flags to indicate the interrupt happened
  - main while loop polls the status of the interrupt flags and does any follow-up processing required by a set flag.
- ISR can get good response
- All of the processing that you put into the ISR has a higher priority than the task code

Round-robin with interrupts

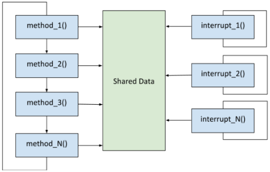
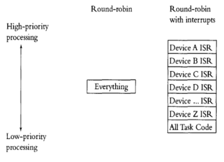


FIG 5. Round Robin with Interrupts[1]

[1] Automaticaddison, A. (2019, May 6). Round-Robin vs Function-Queue-Scheduling. Automatic Addison. https://automaticaddison.com/round-robin-vs-function-queue-scheduling-embedded-software-architecture/#round_robin

Round-robin with interrupts

```
bool fDeviceA = false, fDeviceB =false, fDeviceZ= false;
void interrupt vHandieDeviceA (void){
    // Take care of I/O Device A
    fDeviceA =true;
}
void interrupt vHandleDeviceB (void){
    // Take care of I/O Device B
    fDeviceB =true;
}
void interrupt vHandleDeviceZ (void){
    // Take care of I/O Device Z
    fDeviceZ = true;
}
void main (void){
    while (TRUE){
        if (fDeviceA){
            fDeviceA =false;
            // Handle data to or from I/O Device A
        }
        if (fDeviceB){
            fDeviceB =false;
            // Handle data to or from I/O Device B
        }
        if (fDeviceZ){
            fDeviceZ = false;
            //Handle data to or from I/O Device Z
        }
    }
}
```



Round-robin with interrupts—Pro and cons

Advantages

- Still relatively simple
- Hardware timing requirements better met

Drawbacks

- All task code still executes at same priority
- Maximum delay unchanged
- Worst case response time = sum all other execution times + execution times of any other interrupts that occur

Possible improvements

- Change order flags are checked (e.g., A,B,A,B,A,D)
  - Improves response of A
  - Increases latency of other tasks
- Move some task code to interrupt
  - Decreases response time of lower priority interrupts
  - May not be able to ensure lower priority interrupt code

Real Time Operating System

Real Time Operating System Architecture

- Most complex
- Interrupts signal the need for follow-up tasks
- Instead of a loop deciding what to do next the RTOS decides.
- Interrupts handle urgent operations, then signal that there is more work to do for task code
- One follow-up task can be suspended by the RTOS in favoring of performing a higher priority task.
- Differences with previous architectures
  - We don't write signaling flags (RTOS takes care of it)
  - No loop in our code decides what is executed next (RTOS does this)
  - RTOS knows relative task priorities and controls what is executed next
  - RTOS can suspend a task in the middle to execute code of higher priority

RTOS—Pros and cons

Advantages

- Task do not disturb others —This is actually remarkably hard otherwise
- Provices a standard way for memory protection —if a process tries to access memory that isn't its own, it fails. This is probably a fault and it makes debugging a lot easier.
- Built in priority-based scheduling, abstracting timing information
- Maintainability and

Disadvantages

- An RTOS itself needs some processing time, throughput is affected.
- An RTOS used lot of system resources which is not as good
- Very few tasks run at the same time and their concentration is restricted to few applications to avoid errors
- Quality and industrial-level RTOS are expensive

Conclusion—Architecture Selection

- Select the simplest architecture that will meet your response requirements.
- If your response requirements might necessitate using a real-time operating system then that should probably be your choice.
- Things rarely get smaller/simpler and its a lot easier to start on a more complicated architecture than to migrate to it later when things grew to hairy
- If it makes sense create hybrids

TAB 1. Characteristics of various software architectures

	Priorities available	Worse response time for task code	Code maintainablity	Simplicity
Round-robin	None	Sum of all task code	Poor	Very simple
Round-robin with interrupts	Interrupt routines in priority order, then all task code at the same time	Total of execution time for all task code (plus execution time for interrupt rou-	Good for interrupt routines. Poor for task	Must deal with data shared between interrup routines and task code

The end