Survey of software architectures: function-queue-scheduling architecture and real time OS

Reference: Simon chapter 5
Last class: round robin with interrupts and without interrupts
Function-Queue-Scheduling Architecture
Function-Queue-Scheduling Architecture

• In this architecture, the interrupt routines add function pointers to a queue of function pointers for the main function to call.

• What makes this architecture worthwhile is that no rule says the main task code has to call the functions in the order that the interrupt occurred.

• It can call them based on any priority scheme of your choosing.

• Any task code functions that need quicker response can be executed earlier.

• All you need is some coding in the routines that queue up the function pointers.
!! Queue of function pointers;
void interrupt vHandleDeviceA (void) 
{
  !! Take care of I/O Device A
  !! Put function_A on queue of function pointers
}

void interrupt vHandleDeviceB (void) 
{
  !! Take care of I/O Device B
  !! Put function_B on queue of function pointers
}

void function_A (void) 
{
  !! Handle actions required by device A
}

void function_B (void) 
{
  !! Handle actions required by device B
}

void main(void) 
{
  while (TRUE) 
  {
    while (!!Queue of function pointers is empty)
      !! Call first function on queue
  } //end of while (TRUE)
} //end of void main()
!! Queue of function pointers;
void interrupt vHandleDeviceA (void)
{
  !! Take care of I/O Device A
  !! Put function_A on queue of function pointers
}
void interrupt vHandleDeviceB (void)
{
  !! Take care of I/O Device B
  !! Put function_B on queue of function pointers
}

void function_A (void) {
  !! Handle actions required by device A
}
void function_B (void) {
  !! Handle actions required by device B
}

void main(void)
{
  while (TRUE)
  {
    while (!!Queue of function pointers is empty)
      !! Call first function on queue
  } //end of while (TRUE)
} //end of void main()
!! Queue of function pointers;
void interrupt vHandleDeviceA (void)
{
  !! Take care of I/O Device A
  !! Put function_A on queue of function pointers
}
void  interrupt vHandleDeviceB  (void)
{
  !! Take care of I/O Device B
  !! Put function_B on queue of function pointers
}

void function_A  (void) {
  !! Handle actions required by device A
}

void function_B  (void) {
  !! Handle actions required by device B
}

void main(void)
{
  while (TRUE)
  {
    while (!!Queue of function pointers is empty)
      !! Call first function on queue
  } //end of while (TRUE)
} //end of void main()
Example

- While the code is here, Device A interrupts, then Device B interrupts.

- FunctionA and FunctionB are placed in the queue

- While code executes functionA, another interrupt occurs, for DeviceA.

- The code will finish functionA, then will do functionB and finally function A.

!! Queue of function pointers;
void interrupt vHandleDeviceA (void)
{
    !! Take care of I/O Device A
    !! Put function_A on queue of function pointers
}
void interrupt vHandleDeviceB (void)
{
    !! Take care of I/O Device B
    !! Put function_B on queue of function pointers
}

void function_A (void) {
    !! Handle actions required by device A
}

void function_B (void) {
    !! Handle actions required by device B
}

void main(void)
{
    while (TRUE)
    {
        while (!Queue of function pointers is empty)
            !! Call first function on queue
        //end of while (TRUE)
    } //end of void main()
!! Queue of function pointers;
void interrupt vHandleDeviceA (void) {
    !! Take care of I/O Device A
    !! Put function_A on queue of function pointers
}
void interrupt vHandleDeviceB (void) {
    !! Take care of I/O Device B
    !! Put function_B on queue of function pointers
}

void function_A (void) {
    !! Handle actions required by device A
}

void function_B (void) {
    !! Handle actions required by device B
}

void main(void) {
    while (TRUE) {
        while (!Queue of function pointers is empty)
            !! Call first function on queue
    } //end of while (TRUE)
} //end of void main()
void interrupt vHandleDeviceA (void) {
    !! Take care of I/O Device A
    !! Put function_A on queue of function pointers
}

void interrupt vHandleDeviceB (void) {
    !! Take care of I/O Device B
    !! Put function_B on queue of function pointers
}

void function_A (void) {
    !! Handle actions required by device A
}

void function_B (void) {
    !! Handle actions required by device B
}

void main(void) {
    while (TRUE) {
        while (!!Queue of function pointers is empty)
            !! Call first function on queue
    } //end of while (TRUE)
} //end of void main()
Issues with function-queue-scheduling architectures

• In this architecture the worst wait for the highest-priority task code function is the length of the longest of the task code functions.

• This worst case happens if the longest task code function has just started when the interrupt for the highest-priority device occurs.

• This is because, the function that is being currently processed needs to finish first and only after this is done, the highest priority function in the queue can be executed.

• The response for lower-priority task code functions may get worse.

• If one of the lower-priority task code functions is quite long, it will affect the response for the higher-priority functions.
Issues with function-queue-scheduling architectures

- In this architecture the worst wait for the highest-priority task code function is the length of the longest of the task code functions.

- This worst case happens if the longest task code function has just started when the interrupt for the highest-priority device occurs.

- The response for lower-priority task code functions may get worse.

- If one of the lower-priority task code functions is quite long, it will affect the response for the higher-priority functions.

That’s right... Since interrupts now only add functions to the queue (and don’t execute them immediately), you need to wait until the current function that is being executed to end.
Issues with function-queue-scheduling architectures

• In this architecture the worst wait for the highest-priority task code function is the length of the longest of the task code functions. This worst case happens if the longest task code function has just started when the interrupt for the highest-priority device occurs.

• The response for lower-priority task code functions may get worse.

• If one of the lower-priority task code functions is quite long, it will affect the response for the higher-priority functions.

Yes. If an higher priority interrupt keeps happening, then that particular function will always be the first in the queue to be processed.
Issues with function-queue-scheduling architectures

- In this architecture the worst wait for the highest-priority task code function is the length of the longest of the task code functions.
  - Yes. If an lower priority function is (eventually) executed, and if its really slow... then the higher priority functions will have to wait until the slower function is done.
- The response for lower-priority task code functions may get worse.
- If one of the lower-priority task code functions is quite long, it will affect the response for the higher-priority functions.
Queues vs round robin

- Of course there are some trade-offs.
- In queues higher priority functions have better response, however lower priority functions will suffer.
- In round robin architectures, all functions will eventually be processed.
- With queues, there is an off chance that all low priority functions will never be processed, if high priority interrupts keep happening.
Real-time operating system architecture
Real time architectures

void interrupt vHandleDeviceA (void){
!! Take care of I/O Device A
!! Set signal X
}
void interrupt vHandleDeviceB (void){
!! Take care of I/O Device B
!! Set signal Y
}
void Task1 (void){
  while (TRUE){
!! Wait for Signal X
!! Handle data to/from I/O Device A
  }
}
void Task2 (void){
  while (TRUE){
!! Wait for Signal Y
!! Handle data to/from I/O Device B
  }
}

• Today is just a brief introduction to real time architectures.

• We will be spending most of the next couple of weeks discussing this type of architecture in detail.

• Interrupt routines take care of the urgent operations.

• They then “signal” that there is work to be done in the task code.
Basics of RTOS architectures

- The necessary signaling between the interrupt routines and the task code is handled by the real time operating system (RTOS).
- The RTOS code is not outlined in the code.
- We do **NOT** need shared variables for this.
- No loop in the code decides what to do next. The RTOS also decides what tasks run next.

```c
void interrupt vHandleDeviceA (void){
    !! Take care of I/O Device A
    !! Set signal X
}
void interrupt vHandleDeviceB (void){
    !! Take care of I/O Device B
    !! Set signal Y
}

void Task1 (void){
    while (TRUE){
        !! Wait for Signal X
        !! Handle data to/from I/O Device A
    }
}
void Task2 (void){
    while (TRUE){
        !! Wait for Signal Y
        !! Handle data to/from I/O Device B
    }
}
```
Basics of RTOS architectures

- The RTOS can suspend one task subroutine in the middle of processing in order to run another.
- If Task1 is the highest priority task, then when the vHandleDeviceA interrupt occurs, this task will be done immediately.
- The worst-case wait scenario for Task1 is then zero (plus the interrupt handling times).
Side effect #1 of RTOS architectures: stable response

- The system response is very stable, even when we change the code.
- In round robin architectures and queues, the response time depends on the various task code subroutines (even the lowest priority ones).
- In RTOS architectures, changing the low priority functions, does not affect high-priority functions.
Side effect #2 of RTOS architectures: widely available for purchase

- RTOS are so important and useful that there are several companies that specialize in selling the operating system for each architecture.

- ChibiOS/RT, FreeRTOS, MicroC/OS, Salvo

- Some are even compatible with the Atmel ATmega chips!

- For example, visit the FreeRTOS website.
Disadvantages of RTOS architectures

- The RTOS itself uses a certain amount of processing time.
- The RTOS requires storage space... and in embedded systems space is very sparse.

```c
void interrupt vHandleDeviceA (void) {
    !! Take care of I/O Device A
    !! Set signal X
}
void interrupt vHandleDeviceB (void) {
    !! Take care of I/O Device B
    !! Set signal Y
}

void Task1 (void) {
    while (TRUE) {
        !! Wait for Signal X
        !! Handle data to/from I/O Device A
    }
}
void Task2 (void) {
    while (TRUE) {
        !! Wait for Signal Y
        !! Handle data to/from I/O Device B
    }
}
Summary of software architectures
Which architecture should you use?

- Select the simplest architecture that will meet your response requirements.

- If your system has response requirements that might necessitate using a real-time operating system, then use a real-time operating system.

- If it makes sense, you can even create hybrids architectures (e.g. round-robin with scheduling working side by side a RTOS).
<table>
<thead>
<tr>
<th>Priorities available</th>
<th>Worst time response for task code</th>
<th>Stability of response when code changes</th>
<th>Simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Round-robin</strong></td>
<td>None.</td>
<td>Sum of all task code.</td>
<td>Poor.</td>
</tr>
<tr>
<td><strong>Round-robin with interrupts</strong></td>
<td>Interrupt routines in priority order, then all task code with the same priority.</td>
<td>Total of execution time for all task code (plus execution time for interrupt routines).</td>
<td>Good for interrupt routines. Poor for task code.</td>
</tr>
<tr>
<td><strong>Function queue-scheduling</strong></td>
<td>Interrupt routines in priority order, then all task code in priority order.</td>
<td>Execution time for the longest function (plus execution time for interrupt routines).</td>
<td>Relatively good.</td>
</tr>
<tr>
<td><strong>Real time operating system</strong></td>
<td>Interrupt routines in priority order, then all task code in priority order.</td>
<td>Zero (plus execution time for interrupt routines).</td>
<td>Very good.</td>
</tr>
</tbody>
</table>