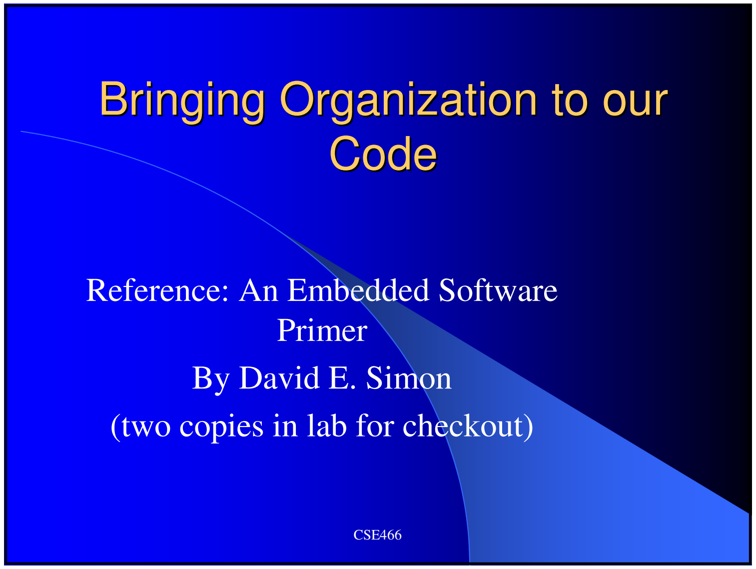


Bringing Organization to our Code

Reference: An Embedded Software Primer

By David E. Simon

(two copies in lab for checkout)

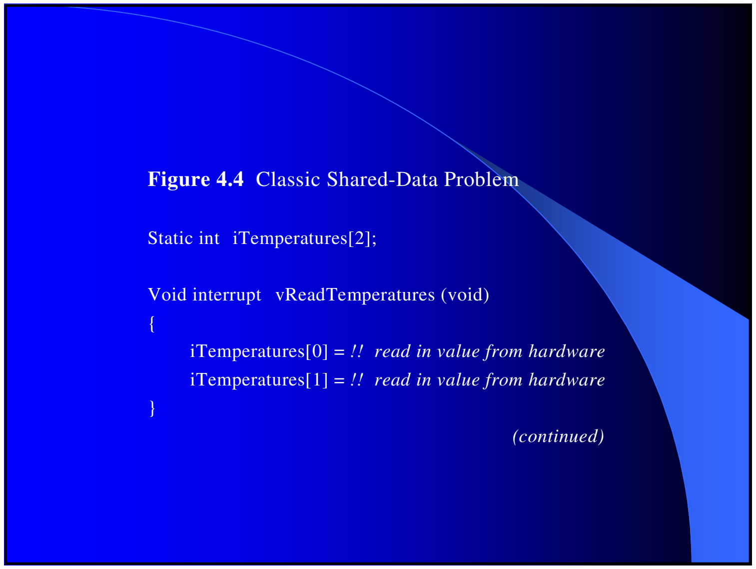


Figure 4.4 Classic Shared-Data Problem

```
Static int  iTemperatures[2];

Void interrupt  vReadTemperatures (void)
{
      iTemperatures[0] = !!  read in value from hardware
      iTemperatures[1] = !!  read in value from hardware
}
```

(continued)

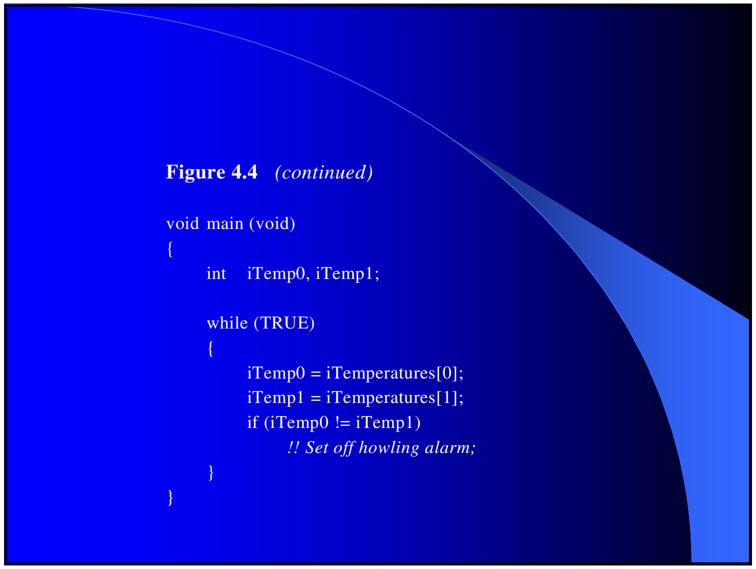
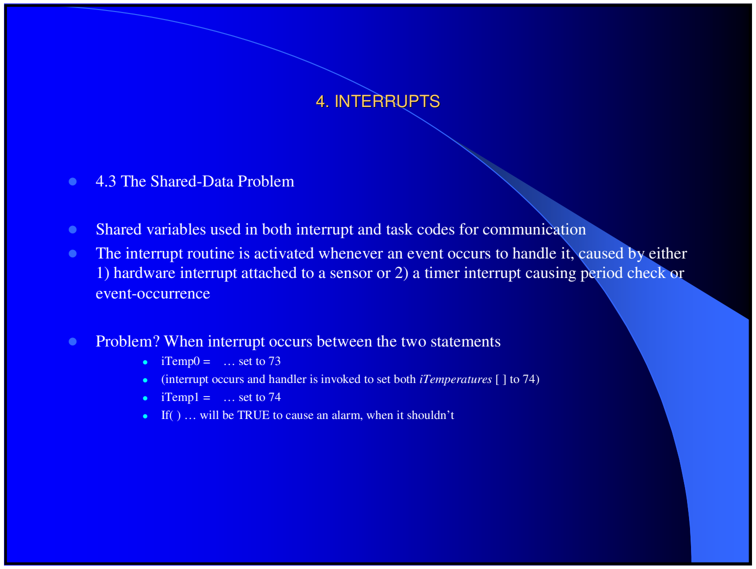


Figure 4.4 *(continued)*

```
void main (void)
{
      int   iTemp0, iTemp1;

      while (TRUE)
      {
            iTemp0 = iTemperatures[0];
            iTemp1 = iTemperatures[1];
            if (iTemp0 != iTemp1)
                  !! Set off howling alarm;
      }
}
```



4. INTERRUPTS

- 4.3 The Shared-Data Problem

- Shared variables used in both interrupt and task codes for communication
- The interrupt routine is activated whenever an event occurs to handle it, caused by either 1) hardware interrupt attached to a sensor or 2) a timer interrupt causing period check or event-occurrence

- Problem? When interrupt occurs between the two statements
  - iTemp0 = … set to 73
  - (interrupt occurs and handler is invoked to set both *iTemperatures* [ ] to 74)
  - iTemp1 = … set to 74
  - If( ) … will be TRUE to cause an alarm, when it shouldn't

4. INTERRUPTS

- The Shared-Data Problem

- Fig 4.5 and Fig 4.6
- Code in Fig 4.5 eliminates setting of local variables, but interrupt can still occur within the if()-statement, causing a false-alarm to be called.

- Code in Fig 4.6 lists the assembly version, which shows that an interrupt can occur after the MOVE R1, … instruction and before MOVE R2, …. Since the first MOVE operation takes a few microseconds to execute before the second MOVE operation, enough time for the hardware to assert an interrupt signal

- The interrupt routine does not change the values in R1 after the call – saved on the stack

---

Figure 4.5 Harder Shared-Data Problem

```
Static int  iTemperatures[2];

Void interrupt  vReadTemperatures (void)
{
      iTemperatures[0] = !!  read in value from hardware
      iTemperatures[1] = !!  read in value from hardware
}
```

(continued)

Figure 4.5 *(continued)*

```
void main (void)
{
      int   iTemp0, iTemp1;

      while (TRUE)
      {
            if (iTemperatures[0] != iTemperatures[1])
                  !! Set off howling alarm;
      }
}
```

Figure 4.6 Assembly Language Equivalent of Figure 4.5

```
.
.
      MOVE          R1,  (iTemperatures[0])
      MOVE          R2,  (iTemperatures[1])
      SUBTRACT      R1,  R2
      JCOND         ZERO,  TEMPERATURES_OK
      .
      .
      ;  Code goes here to set off the alarm
      .
      .
TEMPERATURES_OK:
      .
      .
```

4. INTERRUPT

- 4.3 The Shared-Data Problem - 2
- Solving the Shared Data Problem
  - Use ***disable*** and ***enable*** interrupt instructions when task code accesses shared data
  - Code in Fig 4.7 solves the problem, since even if the hardware asserts an interrupt signal to read the new temperature values (in the handler), the microprocessor will complete the task code first
  - If the task code is in C, the compiler will insert enable/disable instructions in the corresponding assembly code (See Fig 4.8)
  - If the task code in C doesn't have enable/disable constructs, then the embedded programmer must use other mechanisms to allow enable/disable of interrupts
  - Other ways: Atomic or Critical Section code segments for enable/disable interrupt

---

Figure 4.7 Disabling Interrupts Solves the Shared Data Problem from Figure 4.4

```
Static int  iTemperatures[2];

Void interrupt   vReadTemperatures (void)
{
   iTemperatures[0] = !!  read in value from hardware
   iTemperatures[1] = !!  read in value from hardware
}
```

(continued)

Figure 4.7 *(continued)*

```
void    main (void)
{
   int  iTemp0, iTemp1;
   while (TRUE)
   {
        disable ();  /* Disable interrupts while we use the array *
        iTemp0 = iTemperatures[0];
        iTemp1 = iTemperatures[1];
        enable ();
        if (iTemp0 != iTemp1)
             !! Set off howling alarm;
   }
}
```

Figure 4.8 Disabling Interrupts in Assembly Language

```
.
.
    DI                 ; disable interrupts while we use the array
    MOVE          R1,  (iTemperatures[0])
    MOVE          R2,  (iTemperatures[1])
    EI                 ; enable interrupts again

    SUBTRACT   R1,  R2
    JCOND         ZERO,  TEMPERATURES_OK
    .
    .
    ;  Code goes here to set off the alarm
    .
    .
TEMPERATURES_OK:
    .
    .
```

4. INTERRUPTS

- 4.3 The Shared-Data Problem - 3
- Atomic/Critical Section – segment/block of code whose statements must be executed, without interruption because common/shared data is being accessed, in a fixed microprocessor cycles
- Needed in task code when variables/data are shared. (Non-shared data can be accessed or processed anywhere else in the task code.)
- Fig 4.9 shows an example task code, which can return wrong results if the timer asserts an interrupt during the calculations in the if()-statement
- Fig 4.10 is a solution, such that even if the code is called in the critical section of some part of the task code, the enable/disable protections will avoid inadvertent 'enabling' of the interrupt in the middle of that critical section
- Fig 4.11 lists a solution that works when the assembly code for the ***return*** statement is a long-MOVE. It doesn't if it takes multiple short-MOVE operations
- Fig 4.12 lists a solution that reads/re-reads time value without using explicit enable/disable. It works best if compiler optimization is in check to avoid skipping the ***re-read*** or ***while*** statement by using the **<u>volatile</u>** keyword to declare the shared data/variable

---

Figure 4.9 Interrupts with a Timer

```
Static int   iSeconds, iMinutes, iHours;
Void interrupt    vUpdateTime (void)
{
        ++iSeconds;
        if (iSeconds >= 60)
        {
                iSeconds = 0;
                ++iMinutes;
                if (iMinutes >= 60)
                {
                        iMinutes = 0;
                        ++iHours;
                        if (iHours >= 24)
                                iHours = 0;
                }
        }
        !! Do whatever needs to be done to the hardware
}
```

(continued)

Figure 4.9 *(continued)*

```
long lSecondsSinceMidnight (void)
{
        return ( (((iHours * 60) + iMinutes) * 60) + iSeconds);
}
```

Figure 4.10 Disabling and Restoring Interrupts

```
long    lSecondsSinceMidnight (void)
{
    long    lReturnVal;
    BOOL fInterruptStateOld; /* Interrupts already disabled? */

    fInterruptStateOld = disable ();

    lReturnVal = (((iHours * 60) + iMinutes) * 60) + iSeconds;

    /* Restore interrupts to previous state */
    if (fInterruptStateOld)
        enable ();

    return (lReturnVal);
}
```

Figure 4.11 Another Shared-Data Problem Solution

```
Static long int  lSecondsToday;

Void interrupt  vUpdateTime (void)
{
        .
        .
        ++ lSecondsToday;
        if (lSecondsToday = = 60 * 60 * 24)
                lSecondsToday = 0L;
        .
        .
}

long lSecondsSinceMidnight (void)
{
        return (lSecondsToday);
}
```

Figure 4.12 A Program That Needs the `volatile` Keyword

```
Static long int lSecondsToday;

Void interrupt vUpdateTime (void)
{
      .
      .
      ++lSecondsToday;
      if (lSecondsToday = = 60 * 60 * 24)
        lSecondsToday = 0L;
      .
      .
}
```

(continue)

Figure 4.12 *(continued)*

```
long  lSecondsSinceMidnight (void)
{
      long   lReturn;

      /* When we read the same value twice, it must be good. */
      lReturn = lSecondsToday;
      while (lReturn != lSecondsToday)
          lReturn = lSecondsToday;

      return (lReturn);
}
```

---

4. INTERRUPTS

- 4.4 Interrupt Latency
- How long does it take for my embedded system to respond to external stimulus (or interrupt), when the signal is asserted?
- Depends on:
  - 1. How long is the interrupt disabled (service time or handling time)
  - 2. Time it takes to execute/handle the higher priority interrupt (than the current one)
  - 3. Time it takes the microprocessor to save context and jump to the handler
  - 4. Time it takes the handler to save the context and start ‘responsive’ work

- Measuring each of the time periods
  - 1,2,4:
    - (i) Write short and efficient code and measure how long it takes to run (system time), eliminating unrelated/auxiliary code (that can be handled differently) from the handler itself
    - (ii) Look-up and add-up the instruction cycle times for individual instructions
  - 3: Look-up from the microprocessor manufacturer’s manuals

4. INTERRUPTS

- 4.4 Interrupt Latency – 1
- Latency as a function of the time an ***interrupt is disabled***
- E.g., given (the following parameters of a system):
  - Disable time: 125 usec for accessing shared variables in task code
  - Disable time: 250 usec for accessing time variables/values from a timer interrupt
  - Disable time: 625 usec for responding to interprocessor signals
- Will the system work under these constraints?
  - Yes, because after the first 125 usec (task code), the timer and processor interrupt requests will be asserted: the next 250 usec the timer is handled, at which point the clock value will be 375 usec. The processor is then handled (after the 250 usec time), for the next 375 usec – plenty of time to finish before the 625 usec deadline.
  - (See Fig 4.13)
  - If the microprocessor speed is cut in half, all handling and disabled times will double, and under the same constraints, the system will not work.
  - Adding a network handler with higher priority (than the processor), will cause latency problems and won't work (See Fig 4.14)

Figure 4.13 Worst Case Interrupt Latency

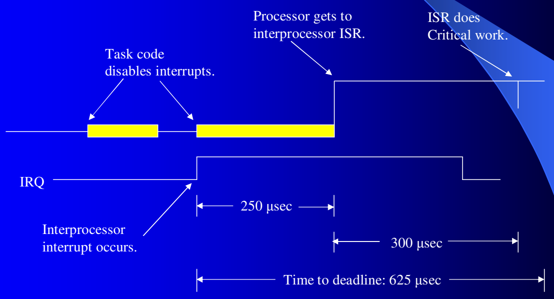
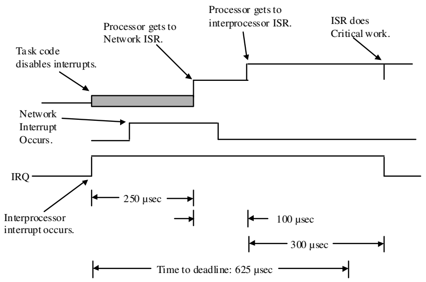


Figure 4.14 Worst Case Interrupt Latency



4. INTERRUPTS

- 4.4 Interrupt Latency - 2
- Avoiding the Disabling of Interrupts
  - Write task and handler code so that both code segments write to, or read from, different parts (buffers) of a shared data structure
  - Fig 4.15 – Arrays A and B, shared between both codes but never accessed at same time
  - Fig 4.16 – A queue structure is shared, but task code read from previously written temp values (different cells in the queue), while the handler writes ahead of the task code

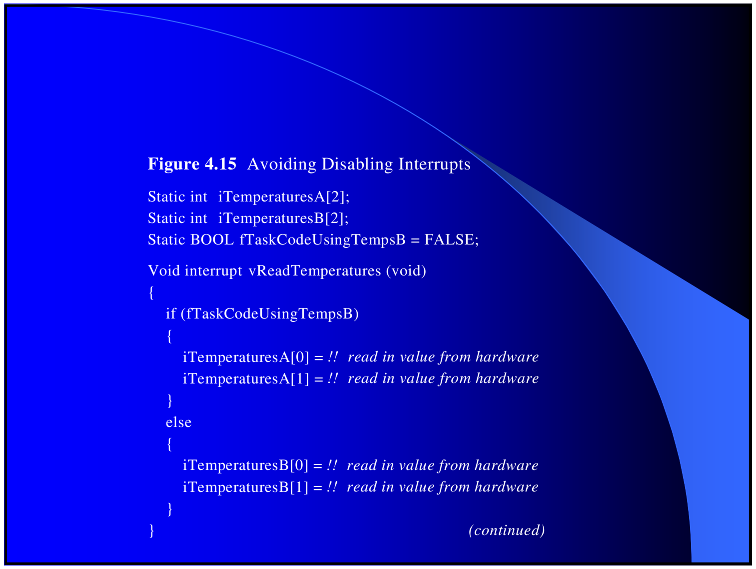


Figure 4.15 Avoiding Disabling Interrupts

```
Static int  iTemperaturesA[2];
Static int  iTemperaturesB[2];
Static BOOL fTaskCodeUsingTempsB = FALSE;

Void interrupt  vReadTemperatures (void)
{
    if (fTaskCodeUsingTempsB)
    {
        iTemperaturesA[0] = !!  read in value from hardware
        iTemperaturesA[1] = !!  read in value from hardware
    }
    else
    {
        iTemperaturesB[0] = !!  read in value from hardware
        iTemperaturesB[1] = !!  read in value from hardware
    }
}
```

(continued)

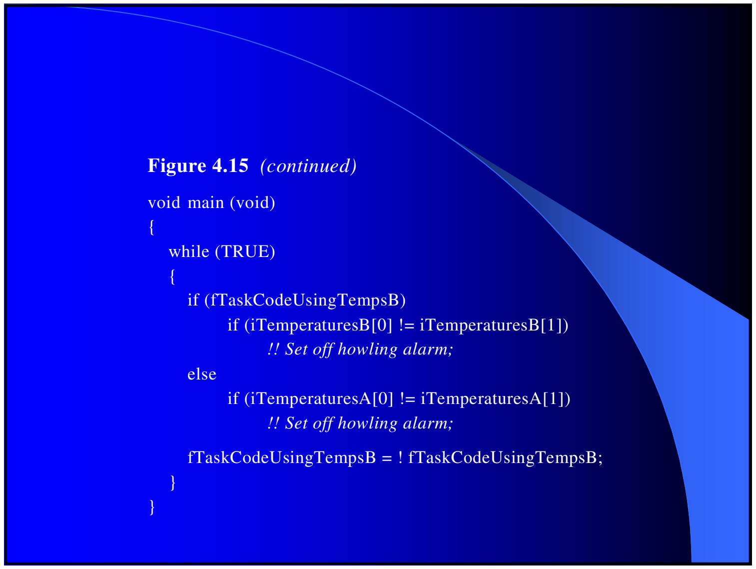


Figure 4.15 *(continued)*

```
void  main (void)
{
    while (TRUE)
    {
        if (fTaskCodeUsingTempsB)
                if (iTemperaturesB[0] != iTemperaturesB[1])
                        !! Set off howling alarm;
        else
                if (iTemperaturesA[0] != iTemperaturesA[1])
                        !! Set off howling alarm;

        fTaskCodeUsingTempsB = ! fTaskCodeUsingTempsB;
    }
}
```

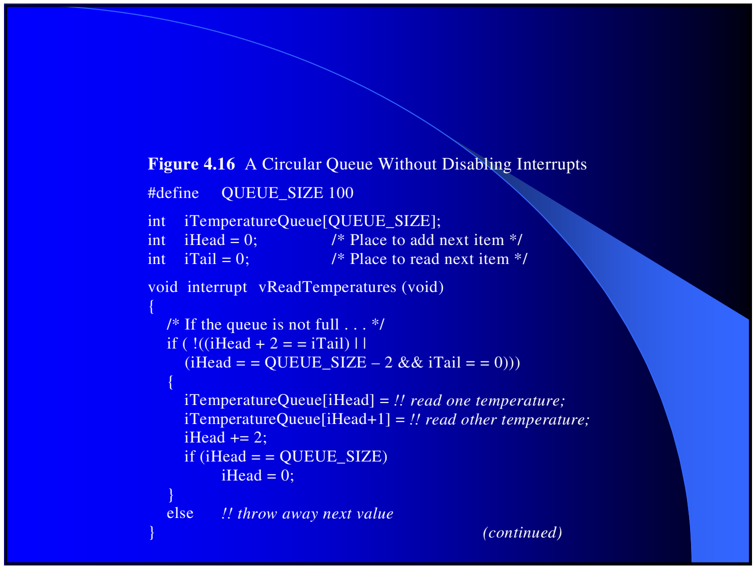


Figure 4.16 A Circular Queue Without Disabling Interrupts

```
#define     QUEUE_SIZE 100

int    iTemperatureQueue[QUEUE_SIZE];
int    iHead = 0;               /* Place to add next item */
int    iTail = 0;               /* Place to read next item */

void  interrupt  vReadTemperatures (void)
{
    /* If the queue is not full . . . */
    if ( !((iHead + 2 = = iTail) | |
        (iHead = = QUEUE_SIZE – 2 && iTail = = 0)))
    {
        iTemperatureQueue[iHead] = !! read one temperature;
        iTemperatureQueue[iHead+1] = !! read other temperature;
        iHead += 2;
        if (iHead = = QUEUE_SIZE)
                iHead = 0;
    }
    else      !! throw away next value
}
```

(continued)

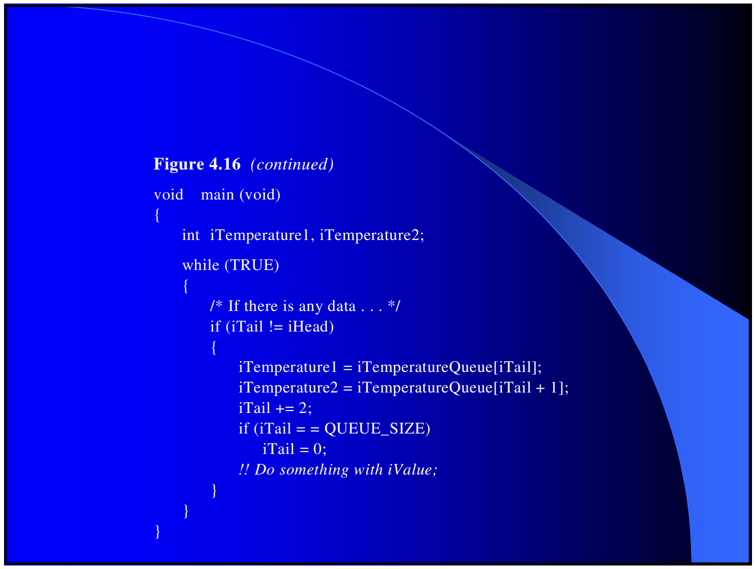


Figure 4.16 *(continued)*

```
void    main (void)
{
      int  iTemperature1, iTemperature2;

      while (TRUE)
      {
            /* If there is any data . . . */
            if (iTail != iHead)
            {
                  iTemperature1 = iTemperatureQueue[iTail];
                  iTemperature2 = iTemperatureQueue[iTail + 1];
                  iTail += 2;
                  if (iTail = = QUEUE_SIZE)
                       iTail = 0;
                  !! Do something with iValue;
            }
      }
}
```

SURVEY OF SOFTWARE ARCHITECTURES

CSE466

5.0 SURVEY OF SOFTWARE ARCHITECTURES

- 5.0 Overview
  - The basic ‘computational model’ that helps structuring or organizing the components of your embedded software
  - The underlying criterion is: *how much (logical) control is needed to satisfy the required system ‘response time.’*
  - Other factors affecting ‘response’ are: processor speed, system overhead
  - Guides:
    - A simple architecture: if response time is not a major issue
    - A complex architecture: if there are multiple, rapid deadline and priority requirements
  - Topics:

Round-robin	- simple
Round-robin with interrupts	- fairly complex
Function-queue-scheduling	- complex
Real-time operating system	- very complex

Round-Robin

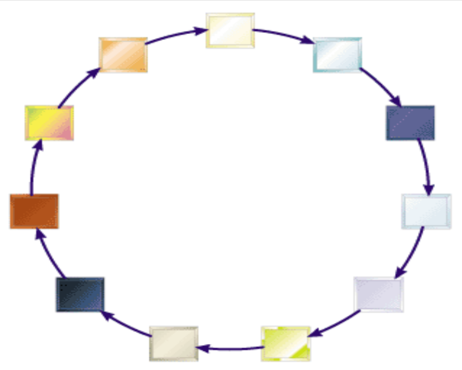


Figure 1: Basic cyclic executive

5.0 SURVEY OF SOFTWARE ARCHITECTURES

- 5.1 Round-Robin Architecture
- Advantages:
  - Simple
  - No interrupts and no shared data
  - No response latency (and no overhead)
  - More suitable for systems that require one-at-a-time operation (e.g., digital watches, microwave ovens with simple functionality)
- Disadvantages:
  - If any device or service/processing needs *response* in less time than it takes the microprocessor to complete processing any system component (e.g., a loop, a module, a basic functionality) – then RR won’t work
  - Adding more functionality, devices, or service/processing introduces potential ‘timing’ or ‘response time’ problems, which weakens the RR arch

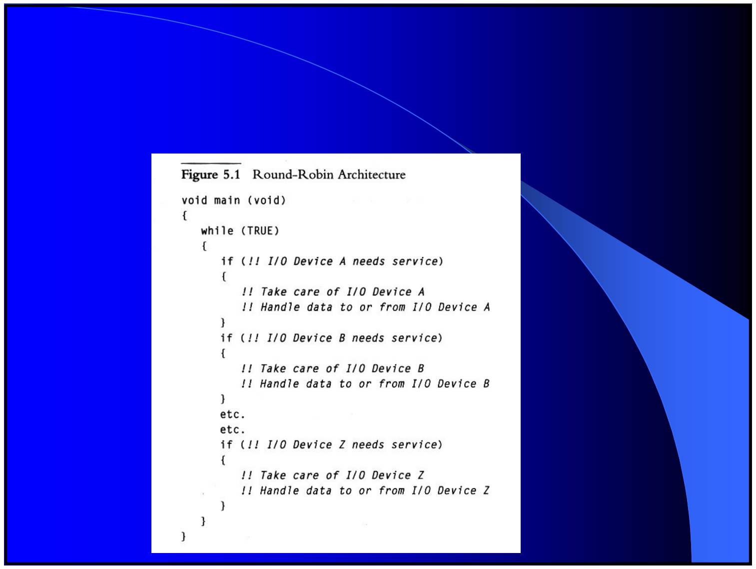


Figure 5.1 Round-Robin Architecture

```
void main (void)
{
   while (TRUE)
   {
      if (!! I/O Device A needs service)
      {
         !! Take care of I/O Device A
         !! Handle data to or from I/O Device A
      }
      if (!! I/O Device B needs service)
      {
         !! Take care of I/O Device B
         !! Handle data to or from I/O Device B
      }
      etc.
      etc.
      if (!! I/O Device Z needs service)
      {
         !! Take care of I/O Device Z
         !! Handle data to or from I/O Device Z
      }
   }
}
```

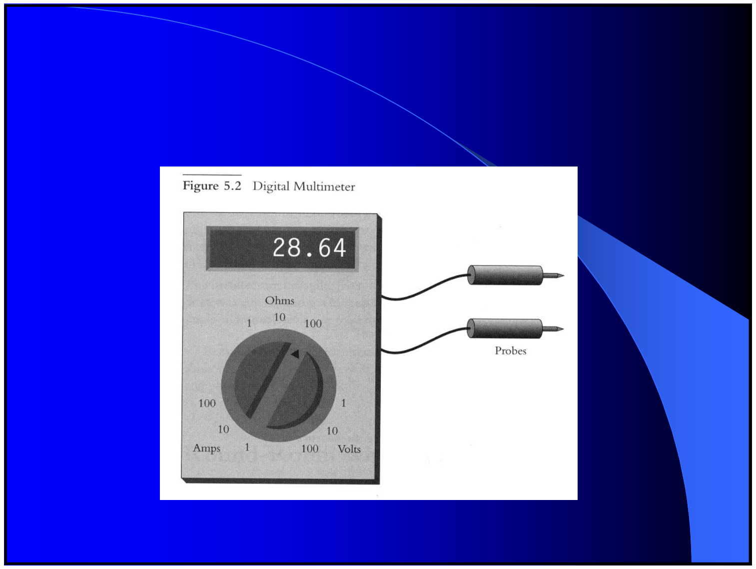


Figure 5.2 Digital Multimeter

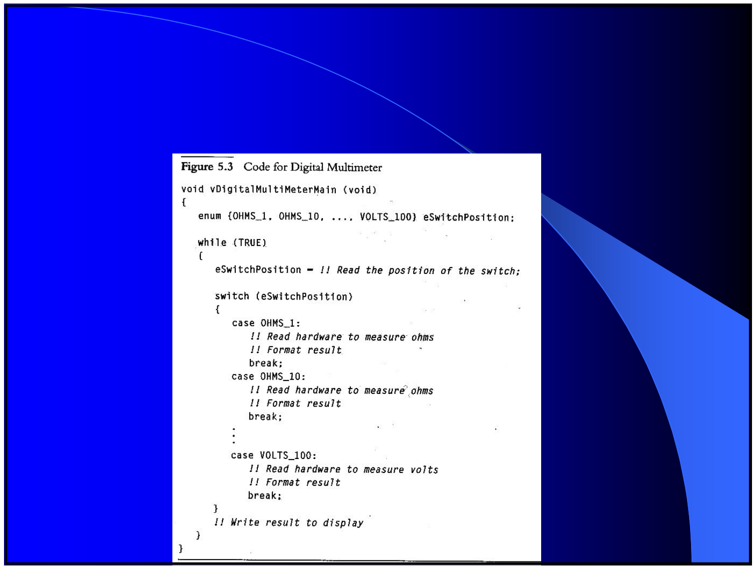
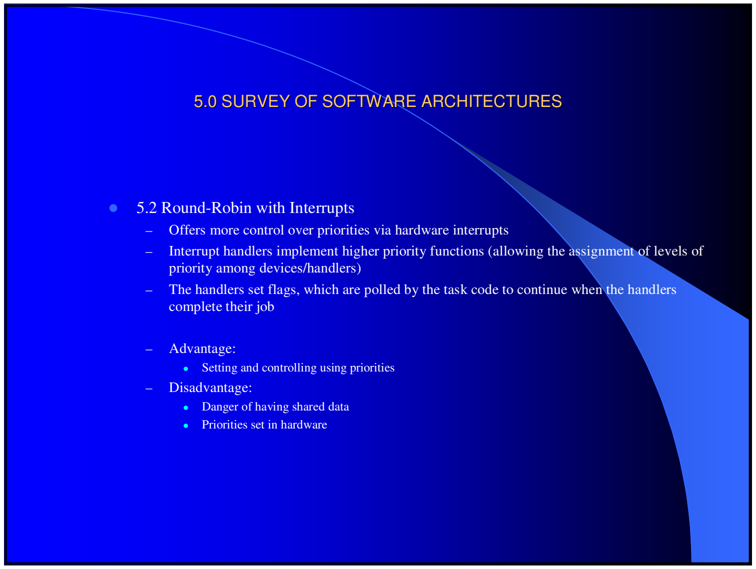


Figure 5.3 Code for Digital Multimeter

```
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
   }
}
```



5.0 SURVEY OF SOFTWARE ARCHITECTURES

- 5.2 Round-Robin with Interrupts
  - Offers more control over priorities via hardware interrupts
  - Interrupt handlers implement higher priority functions (allowing the assignment of levels of priority among devices/handlers)
  - The handlers set flags, which are polled by the task code to continue when the handlers complete their job
  - Advantage:
    - Setting and controlling using priorities
  - Disadvantage:
    - Danger of having shared data
    - Priorities set in hardware

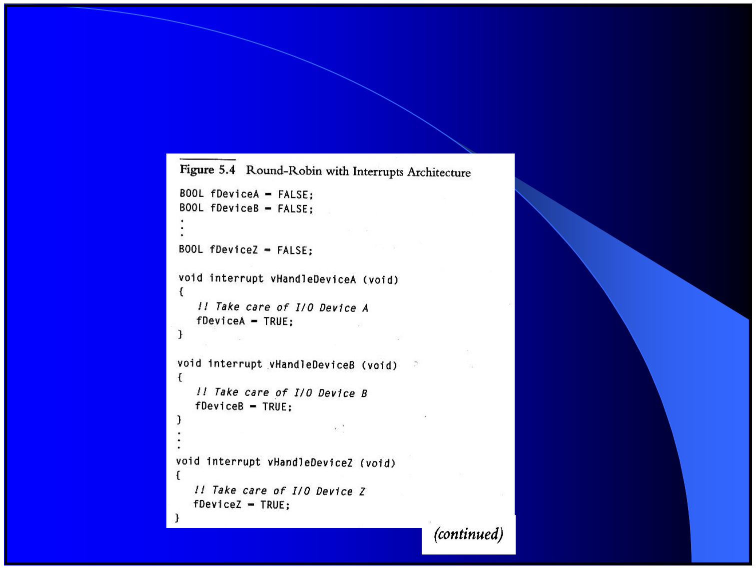


Figure 5.4 Round-Robin with Interrupts Architecture

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
:
BOOL fDeviceZ = FALSE;

void interrupt vHandleDeviceA (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;
}
```

(continued)

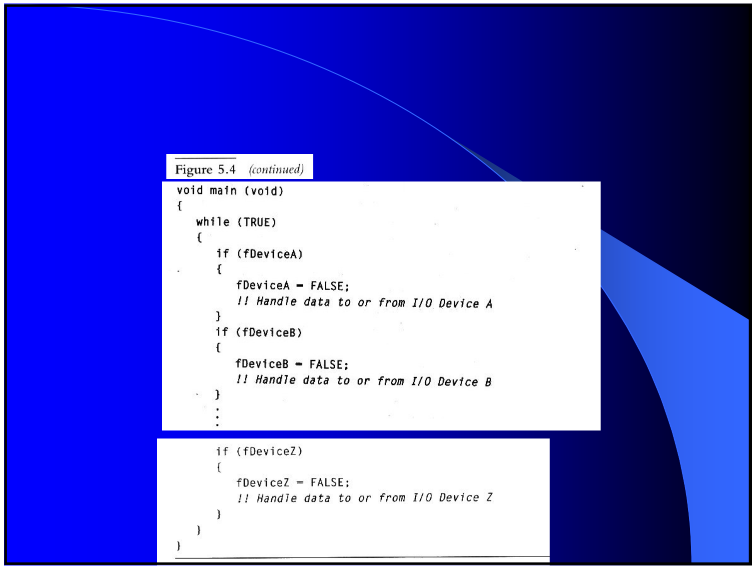


Figure 5.4 *(continued)*

```
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
      }
   }
}
```

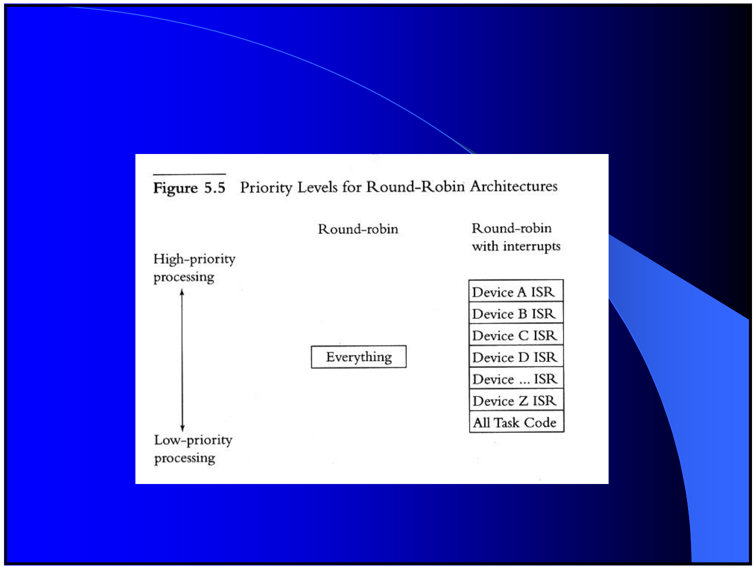
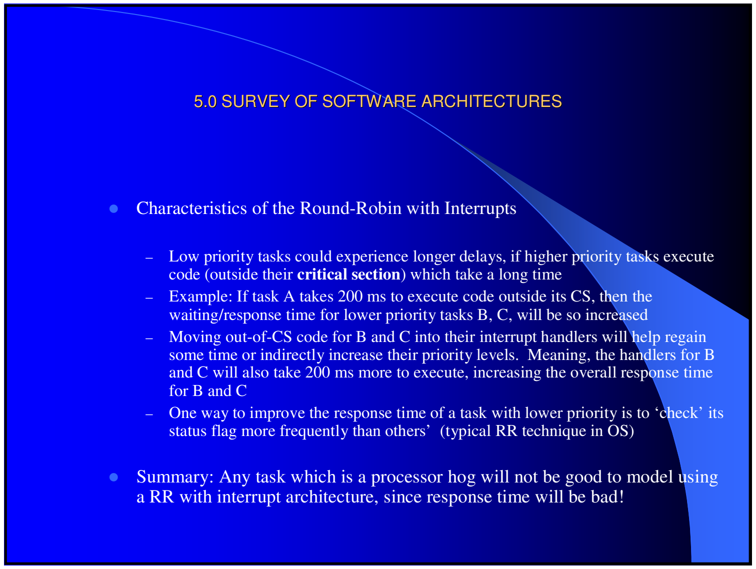


Figure 5.5 Priority Levels for Round-Robin Architectures



5.0 SURVEY OF SOFTWARE ARCHITECTURES

- Characteristics of the Round-Robin with Interrupts
  - Low priority tasks could experience longer delays, if higher priority tasks execute code (outside their **critical section**) which take a long time
  - Example: If task A takes 200 ms to execute code outside its CS, then the waiting/response time for lower priority tasks B, C, will be so increased
  - Moving out-of-CS code for B and C into their interrupt handlers will help regain some time or indirectly increase their priority levels. Meaning, the handlers for B and C will also take 200 ms more to execute, increasing the overall response time for B and C
  - One way to improve the response time of a task with lower priority is to 'check' its status flag more frequently than others' (typical RR technique in OS)
- Summary: Any task which is a processor hog will not be good to model using a RR with interrupt architecture, since response time will be bad!

5.0 SURVEY OF SOFTWARE ARCHITECTURES

- 5.3 Function-Queue-Scheduling Architecture
  - Improves the response time of higher priority tasks.
  - Interrupt routines add function-pointers to a queue for the main task to execute
  - The main task continuously scans the queue and executes the corresponding function
  - Allows placing function-pointers in the queue based on preferred priority scheme (placement is done by a supporting/auxiliary routine invoked by the handlers)

  - Characteristics:
    - The worst-case response time for highest-priority tasks is: sum(longest task code, any interrupts this code generates), and not the sum of the response times of all the handlers
    - The response time for lowest-priority tasks could be long when their code segments are long

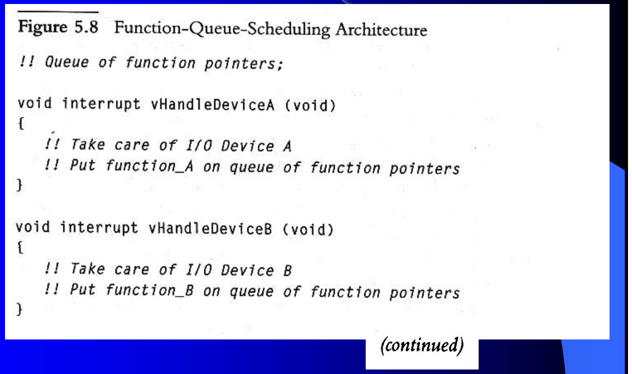


Figure 5.8 Function-Queue-Scheduling Architecture

```
!! 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
}
```

(continued)

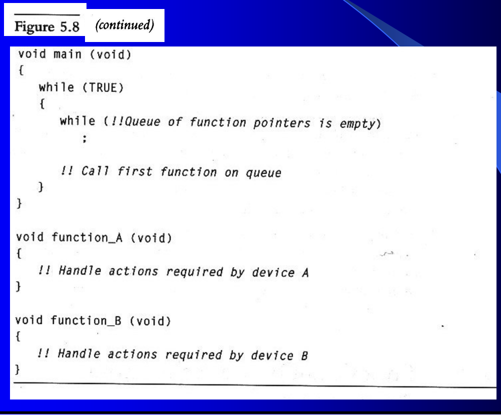


Figure 5.8 *(continued)*

```
void main (void)
{
   while (TRUE)
   {
      while (!!Queue of function pointers is empty)
         ;

      !! Call first function on queue
   }
}

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

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

Reference:

An Embedded Software Primer
By David E. Simon
(two copies in lab for checkout)