4.3: The shared data problem

- **Inconsistency in data used by a task and updated by an ISR; arises because ISR runs at just the wrong time.**
- Data is often shared because it is undesirable to have ISRs do all the work – they would take too long to run.
  - ISRs typically "hand off" some of the processing to task code.
  - This implies shared variables or communication between the ISR and the related task.
- Lab 3 simpler (in part) because we don't have to worry about this: the task and interrupt code are unrelated.

Figure 4.4: code example

```
static int iTemperatures[2];

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

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

What does this code do?

Fig 4.4: observations

- Note keyword "interrupt" in first function. (It is an ISR written in C; our tools don't support this.)
  - It is never called from task code; when will it run?
  - How do we connect this ISR with its interrupt?
- The main routine is an infinite loop.
  - Rare in conventional code, common in embedded systems.
  - Compares two temperatures and raises alarm if they ever differ.
- The ISR updates the temperature variables.
  - Assume interrupt asserted at
    - regular intervals, based on timer, or
    - when either temperature changes

Figure 4.4: analysis

```
static int iTemperatures[2];

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

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

What can go wrong?

Suppose interrupt occurs here

The shared-data problem

- Imagine this scenario:
  - Temperature rising, both values identical at each reading.
    - Say, 80 at one reading, 81 at the next.
  - Interrupt occurs between reads in task code.
  - Test in main() compares old value with new value.
  - Result: (false) alarm set off, evacuations begin.
- To prevent, programmer must carefully analyze all code
  - **Is there a point in code where an interrupt can mess things up?**

Figure 4.5: Does this fix problem?

```
static int iTemperatures[2];

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

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

Only change to code:
- global array values tested directly

Consider 8086 instruction sequence

```
static int iTemperatures[2];

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

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

```
        ...
        mov     ax,[iTemperatures+0]
        cmp     ax,[iTemperatures+2]
        je      okay
        ; set off alarm
okay:   ...
```


Ensuring correctness

- Key issue: will *single* machine instruction access the two values?
- If not (the case for almost all CPUs),
  - An interrupt *can* occur between the two memory reads. (Will it?)
  - The code *can* trigger a false alarm.
- If yes, the code may work for this CPU, but not others.
  - Best if the code we write will work on all target platforms.

The big picture

- When does shared data problem arise?
  - When data is shared between an ISR and task code it interrupts, and when the data can reach an inconsistent state through the actions of the ISR.
- The hard part:
  - Does the bug appear consistently?
  - Would it turn up during testing?
- Only real solution: **write bug-free code.**
  - Think long and hard about correctness of code at all levels.
  - Stick with basic principles that work.
  - But still do lots of testing!

Figure 4.5
One solution: disable interrupts

```
static int iTemperatures[2];

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

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

```
        ...
        cli
        mov     ax,[iTemperatures+0]
        cmp     ax,[iTemperatures+2]
        sti
        je      okay
        ; set off alarm
okay:   ...
```

Why does this work?

Implementation options

inline assembly

```
_asm {
    cli
}
iTemp0 = ...;
iTemp1 = ...;
_asm {
    sti
}
```

function call

```
disable();
iTemp0 = ...;
iTemp1 = ...;
enable();

; assembly code
disable:  cli
          ret

enable:   sti
          ret
```

What are tradeoffs?

Comparison

- Overhead for function call method
  - **call**, **cli**, **ret**
- Overhead for inline assembly method
  - **cli**
- Which method gives best performance?
  - Is the difference significant?
- Which method results in more portable code?

Discussion

- Why lock out *all* interrupts, and not just mask the one with the ISR that accesses the shared data?
  - Selective masking would reduce disruption to rest of system.
- Considerations:
  - Interrupts are disabled only briefly.
  - Increasing response time by 1-2 instructions is not a big deal.
  - The overhead of disabling single interrupt is generally higher; details are platform dependent.
- Disabling all interrupts is a simple, one-size-fits-all solution.
  - BUT you must ensure that interrupts are not disabled for too long!

Compiler limitations

- Why can't compilers handle this automatically?
  - In general, compilers cannot identify (truly) shared data, let alone analyze dynamic access patterns to that data.
  - It's plenty hard for humans to do – even for developer who understands the code.
- No existing tools are clever enough to determine automatically when interrupts need to be disabled.

Terminology

- **Atomic**: a section of code is atomic if it cannot be interrupted, i.e., if it can be guaranteed to execute as an unbreakable unit.
- **Critical Section**: a section of code that must be atomic for correct operation.

Atomicity

- Shared data problem arises when task code accesses shared data non-atomically.
- What are the natural atomic units of execution?
  - Single machine instructions only.
  - A line of C-code rarely maps to a single instruction. (If a line of your C-code must be atomic, then you have a critical section.)
- How can we make portion of code atomic?
  - Principal approach: disabling interrupts at start, enable interrupts at end.
  - We'll consider alternative approaches later.

Figure 4.9: What can go wrong here?

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

How far off can return value be?

Figure 4.9: Making it atomic

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

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

A very bad "solution"!

Figure 4.9: Making it atomic

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

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

A better solution

Figure 4.9: Making it atomic

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

```
long lSecondsSinceMidnight(void)
{
    long lReturnVal;
    BOOL fInterruptStateOld;
    fInterruptStateOld = disable();
    lReturnVal = (((iHours*60)+iMinutes)*60)+iSeconds;
    if (fInterruptStateOld) enable();
    return lReturnVal;
}
```

The best solution

A subtle point

- What can go wrong with "better" solution?
  - Consider scenario: function called within critical section in another function: interrupts will be re-enabled and should not be.
  - Some of you will experience this in your code this semester.
- How is "best" solution an improvement?
  - Re-enables interrupts only if they were on in first place.
  - Allows function with critical section to be called from normal code and from other critical sections.

Figure 4.11: Another approach: What was changed, and does it work?

```
static long int lSecondsToday;

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

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


Fig 4.11: Discussion

- Just counts seconds, only one shared variable.
  - ISR, task functions share a single variable.
- Does the problem go away?
  - No, just more subtle: accessing a single variable is not necessarily atomic.
  - Example: accessing a long on 8086 takes multiple instructions; can be interrupted between 16-bit accesses. (How far off can it be?)
- Bottom line: even with code accessing a single shared variable, you're usually better off disabling interrupts.
  - Code more portable to new target platforms.

Figure 4.12: Yet another approach

```
static long int lSecondsToday;
void interrupt vUpdateTime (void)
{
    ...
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    ...
}
long lSecondsSinceMidnight(void)
{
    long lReturn;
    lReturn = lSecondsToday;
    while (lReturn != lSecondsToday)
        lReturn = lSecondsToday;
    return lReturn;
}
```

Fig 4.12: Discussion

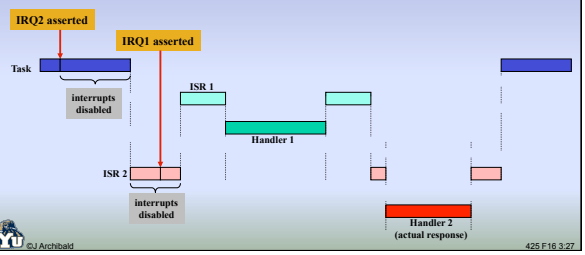
- Basic idea: read value repeatedly until you get two identical readings
  - An alternative to disabling interrupts.
- But what will a good optimizing compiler do with this code?
  - Read from memory just once, keep the value in a register.
  - Compiler sees nothing in code to modify value between the two reads.
- Solution?
  - Use **volatile** keyword: forces compiler to read memory every time variable is accessed and to avoid "obvious" optimizations.
  - Tells compiler that variable can be changed by something unseen.

Figure 4.12: Modified version

```
static volatile long int lSecondsToday;
void interrupt vUpdateTime (void)
{
    ...
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    ...
}
long lSecondsSinceMidnight(void)
{
    long lReturn;
    lReturn = lSecondsToday;
    while (lReturn != lSecondsToday)
        lReturn = lSecondsToday;
    return lReturn;
}
```


Response time revisited

- How long does it take for the system to respond to an interrupt?



Worst-case interrupt latency: components

1. The longest period of time that interrupts are disabled
2. The total time required to execute all ISRs + handlers of higher priority
3. The time for hardware to stop what it is doing, save critical state, and start executing the ISR for that interrupt
4. The time for the ISR+handler to save the context and then do the work that we consider to be the "response"

What can designer control?

1. Max length of critical sections?
   - Keep them short!
2. Execution time of higher-priority ISRs?
   - Assign priorities carefully.
   - Keep all ISRs lean and mean.
3. Overhead of hardware response?
   - Fixed when you select the processor.
4. Time to save context, run handler?
   - Size of context depends on number of registers – fixed for CPU
   - Handler efficiency: good coding

Measuring time

- In simulator, time unit is time to execute one instruction
  - Simple model: all instructions take same time to execute
  - Unlikely to be true in any implementation, but added realism buys little.
- CPU respond to asserted, enabled interrupt before starting next instruction
- Overhead of hardware response on 8086:
  - Finish current instruction
  - Push 3 words on stack, read 2 words from interrupt vector table

Meeting design specifications

- What do we need to know to ensure that response time will be less than, say, 625 μs?
  - Identify all critical sections, max length of each
    - Only **longest** critical section need concern us: no way to transition to another without hardware responding to pending interrupt.
  - Hardware priority level assigned to relevant interrupt
  - Run length of higher-priority ISRs + handlers
    - Just one time through each, or multiple runs?
  - Run length of this ISR + handler to point of "response"
- How important is such a guarantee?
  - Critical in real world, less so in 425 labs

Fig. 4.15: Another alternative to disabling interrupts

```
static int iTemperaturesA[2], iTemperaturesB[2];
static BOOL fTaskCodeUsingTempsB = FALSE;
void interrupt vReadTemperatures (void)
{
      if (fTaskCodeUsingTempsB)
      {
            iTemperaturesA[0] = !! read in value from HW
            iTemperaturesA[1] = !! read in value from HW
      }
      else
      {
            iTemperaturesB[0] = !! read in value from HW
            iTemperaturesB[1] = !! read in value from HW
      }
}

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;
      }
}
```


Fig. 4.15: Discussion

- Key idea: use *double buffering* with a global flag to ensure that the reader and writer access separate arrays.
- Does this work?
  - Global flag does not change while temperatures are being read in task code, especially at critical point between the two reads.
  - Values tested in task code are always corresponding pair – no way for ISR to change them at wrong time while reading.
- What are disadvantages?

Figure 4.16: Yet another alternative

```
#define Q_SIZE 100
int iTemperatureQ[Q_SIZE];
int iHead = 0;
int iTail = 0;

void interrupt vReadTemperatures (void)
{
    if ( !(( iHead+2==iTail) ||
          (iHead==Q_SIZE-2 && iTail==0)))
    {
        iTemperatureQ[iHead] =
            !! read one temperature
        iTemperatureQ[iHead+1] =
            !! read other temperature
        iHead += 2;
        if (iHead==Q_SIZE)
            iHead = 0;
    }
    else
        !! throw away next value
}
```

```
void main (void)
{
    int iTemp1, iTemp2;

    while (TRUE)
    {
        if (iTail != iHead)
        {
            iTemp1 = iTemperatureQ[iTail];
            iTemp2 = iTemperatureQ[iTail+1];
            iTail += 2;
            if (iTail == Q_SIZE)
                iTail = 0;
            !! Compare values
        }
    }
}
```


Figure 4.16: Discussion

- Key idea: use circular queues.
  - Queue buffers data between ISR and task that processes it.
  - Buffering with queues is a commonly used technique.
- Queue management:
  - Queue full: head+2 == tail (2 slots used/sample)
  - Queue empty: head == tail
- Advantage: queue decouples the data arrival rate (possibly bursty) from the data processing rate.
  - Processing rate must be at least as great as the average arrival rate.

Figure 4.16: Discussion

- How fragile is this code? How easy to get it wrong?
  - Task must read the data, then revise tail variable
    - Reversing order would allow ISR to overwrite data before it is read.
  - When tail is incremented, the *write* (not necessarily the increment) to tail must be atomic.
    - Otherwise reader and writer could see different pictures of shared array.
    - The operation is *generally* atomic, but not on all platforms.
- Overall assessment:
  - Queue approach is tricky to get right
  - Makes sense only if disabling interrupts is really not an option

Problem 4.1: Does this approach avoid a shared data problem?

```
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;
        }
    }
    !! Deal with HW
}
```

```
void vSetTimeZone (int iZoneOld, int iZoneNew)
{
    int iHoursTemp;

    /* Get current hours */
    disable();
    iHoursTemp = iHours;
    enable();

    !! adjust iHoursTemp for new time zone
    !! adjust for daylight savings time also

    /* save the new hours value */
    disable();
    iHours = iHoursTemp;
    enable();
}
```

Code based on Figure 4.17

Problem 4.2: The code below has a shared data bug.

```
static long int lSecondsToday;

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

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

(a) How far off can return value of function be if sizeof(long) is 32 and word size is 16 bits?

(b) How far off can return value of function be if sizeof(long) is 32 and word size is 8 bits?

Problem 4.3: What additional bug lurks in this code, even if registers are 32 bits in length?

```
static long int lSecondsToday;

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

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

What can happen if system has another interrupt that is higher priority than timer interrupt for vUpdateTime and that calls lSecondsSinceMidnight?


```
static int iTemperaturesA[2], iTemperaturesB[2];
static BOOL fTaskCodeUsingTempsB = FALSE;
void interrupt vReadTemperatures (void)
{
    if (fTaskCodeUsingTempsB)
    {
        iTemperaturesA[0] = !! read in value from HW
        iTemperaturesA[1] = !! read in value from HW
    }
    else
    {
        iTemperaturesB[0] = !! read in value from HW
        iTemperaturesB[1] = !! read in value from HW
    }
}

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;
    }
}
```

Problem 4.5: The task and interrupt code share the fTaskCodeUsingTempsB variable.

Is the task's use of this variable (fTaskCodeUsingTempsB) atomic? Does it need to be atomic for the code to work correctly?


```
int iQueue[100];
int iHead = 0;  /* place to add next item */
int iTail = 0;   /* place to read next item */
void interrupt SourceInterrupt(void)
{
    if ((iHead+1 == Tail) || (iHead == 99 && iTail == 0))
    {   /* if queue is full, overwrite oldest */
        ++iTail;
        if (iTail == 100)
            iTail = 0;
    }
    iQueue[iHead] = !!next value;
    ++iHead;
    if (iHead==100)
        iHead = 0;
}

void SinkTask(void)
{
    int iValue;
    while (TRUE)
      if (iTail != iHead)
      {   /* if queue has entry, process it */
          iValue = iQueue[iTail];
          ++iTail;
          if (iTail == 100)
              iTail = 0;
          !! Do something with iValue;
      }
}
```

Problem 4.6: where is "very nasty bug"?

Code from Figure 4.18


```
int iQueue[100];
int iHead = 0;  /* place to add next item */
int iTail = 0;   /* place to read next item */
void interrupt SourceInterrupt(void)
{
    if ((iHead+1 == Tail) || (iHead == 99 && iTail == 0))
    {   /* if queue is full, overwrite oldest */
        ++iTail;
        if (iTail == 100)
            iTail = 0;
    }
    iQueue[iHead] = !!next value;
    ++iHead;
    if (iHead==100)
        iHead = 0;
}

void SinkTask(void)
{
    int iValue;
    while (TRUE)
      if (iTail != iHead)
      {   /* if queue has entry, process it */
          iValue = iQueue[iTail];
          ++iTail;
          if (iTail == 100)
              iTail = 0;
          !! Do something with iValue;
      }
}
```

Problem 4.6: where is "very nasty bug"?

Code from Figure 4.18

Scenario 1.
Queue is full, say, iHead=20,iTail=21
Task about to read iQueue[iTail], value 21 already in register
Interrupt occurs: code sets iHead to 21, iTail to 22
Task reads iQueue[21] which is newest (rather than oldest) entry

```
int iQueue[100];
int iHead = 0;  /* place to add next item */
int iTail = 0;   /* place to read next item */
void interrupt SourceInterrupt(void)
{
    if ((iHead+1 == Tail) || (iHead == 99 && iTail == 0))
    {   /* if queue is full, overwrite oldest */
        ++iTail;
        if (iTail == 100)
            iTail = 0;
    }
    iQueue[iHead] = !!next value;
    ++iHead;
    if (iHead==100)
        iHead = 0;
}

void SinkTask(void)
{
    int iValue;
    while (TRUE)
      if (iTail != iHead)
      {   /* if queue has entry, process it */
          iValue = iQueue[iTail];
          ++iTail;
          if (iTail == 100)
              iTail = 0;
          !! Do something with iValue;
      }
}
```

Problem 4.6: where is "very nasty bug"?

Code from Figure 4.18

Scenario 2.
Queue is full, iHead=98,iTail=99
Task executes ++iTail (so iTail=100)
Back-to-back interrupts are executed.
Start of first: iHead=98, iTail=100
End of first: iHead=99, iTail=100.
End of second: iHead=0, iTail=101

iTail is never reset, increases w/o limit

Chapter 5: Software architectures

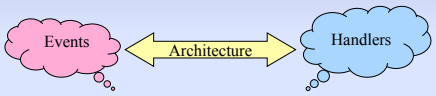
- Recap: important ideas in real-time code
  - ISRs: scheduled by hardware
  - Task code: scheduled by software
    - Similar to Linux "process" in this regard
  - Response time constraints
  - Simplicity vs. complexity
- For any given application, how should code be organized?
  - What alternative organizations exist?

Choosing a software architecture: key factors

- How much control you need over system response time
  - Absolute response time requirements
  - Other processing requirements, including lengthy computations
- How many different events you must respond to
  - Each with possibly different deadlines and priorities
- **In short: what does the system need to do?**

Software architectures

- Event handlers are procedures (typically written in C) that do the "work" to respond to events.



- The architecture determines
  1. how the event is detected, and
  2. how the event handler is called.

Architecture 1: Round-robin

No interrupts involved

One Event

```
while(1)
{
   if (event)
        handle_event();
}
```

Multiple Events

```
while(1)
{
   if (event1)
        handle_event1();
   if (event2)
        handle_event2();
   ...
   if (eventn)
        handle_eventn();
}
```

This approach is typically called polling.

Characteristics of round-robin

- Priorities available:
  - None: actions are all equal; each handler must wait its turn.
- Disadvantages:
  - Worst-case response time one full iteration of loop (possibly handling all other events first).
  - Worst-case response time for *every* event is bad if *any* single event requires lengthy processing.
  - System is fragile: adding a single new event handler may cause deadlines to be missed for other events.
- Advantage:
  - Simplicity: really just a single task, no shared data, no ISRs

How to decrease response time?

```
while(1)
{
   if (eventA)
        handle_eventA();
   if (eventB)
        handle_eventB();
   if (eventC)
        handle_eventC();
   if (eventD)
        handle_eventD();
}
```

How can I reduce the response time for event A?

```
while(1)
{
   if (eventA)
        handle_eventA();
   if (eventB)
        handle_eventB();
   if (eventA)
        handle_eventA();
   if (eventC)
        handle_eventC();
   if (eventA)
        handle_eventA();
   if (eventD)
        handle_eventD();
}
```


Applicability of round-robin

- Example from text: digital multimeter
  - Few input devices, few events to respond to
  - Response time constraints not demanding
  - No lengthy processing required
- Author's conclusion (page 119):

> "Because of these shortcomings, a round-robin architecture is probably suitable only for very simple devices such as digital watches and microwave ovens and possibly not even for these."

Architecture 2: Round-robin with interrupts

- To single polling loop, add interrupts.
  - ISRs complete initial response.
  - Remainder done by functions called in loop.
  - ISR sets flag to indicate that processing is required.
- Offers greater flexibility:
  - Time-critical processing can be in ISR.
  - Longer-running code can be in handlers.

Round-robin with interrupts

```
while(1)
{
   if (flagA) {
        flagA = 0;
        handle_eventA();
   }
   if (flagB){
        flagB = 0;
        handle_eventB();
   }
   if (flagC){
        flagC = 0;
        handle_eventC();
   }
}
```

```
ISR_A {
    !! do some A stuff
    flagA = 1;
}
ISR_B {
    !! do some B stuff
    flagB = 1;
}
ISR_C {
    !! do some C stuff
    flagC = 1;
}
```

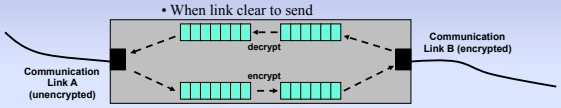
Work is split between task code and ISRs.

Example: communications bridge

Constraints

What is time critical?
- Not losing data
- Maintaining good throughput

Assume interrupts occur:
- When data arrives
- When link clear to send



Design

ISR actions:
- Buffer data on arrival
- Set flag when clear to send

Operations within main loop:
- Encrypt buffered data from Link A
- Decrypt buffered data from Link B
- Send data on Link A
- Send data on Link B

Characteristics of round-robin with interrupts

- Priorities available:
  - Interrupts are serviced in priority order.
  - All handlers have equal priority: none more important than the others.
- Worst-case response time
  - For ISR: execution time of higher priority ISRs (if any)
  - For handler: sum of execution of all other handlers + interrupts
- Advantages:
  - Work performed in ISRs has higher priority than code in main loop.
  - ISR response time stable through most code changes.
- Disadvantages:
  - ISRs and handlers will share data, shared data problems will appear!
  - Handler response time not stable when code changes.

Architecture 3: Function-queue scheduling

Work split between ISR and task code.

Order of tasks is dynamic.

Queue can be FIFO or sorted by priority.

```
ISR_A
{   !! do some work relating to A
    queue_put(handle_eventA);
}
ISR_B
{   !! do some work relating to B
    queue_put(handle_eventB);
}
while(1)
{
   while (queue_empty()); /* wait */
   task = get_queue();
   (*task);            /* = task() */
}
```

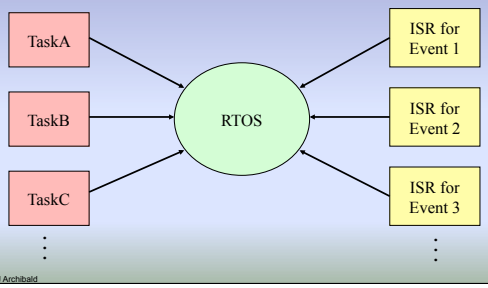

Characteristics of Function-queue scheduling

- Priorities available:
  - Interrupts are serviced in priority order
  - Tasks can be placed in queue and run in priority order
- Worst-case response time for highest-priority task
  - Scenario: just started executing another task, must wait for it to finish
  - Delay = longest task time + execution time for ISRs
- Advantages:
  - Improved response-time stability when code changes
- Disadvantages:
  - Some added complexity from function queue

Architecture 4: Real-time operating system (RTOS)

- Work is split between ISRs and *tasks*.
- Tasks are prioritized and run by a scheduler.
  - Scheduler always picks highest-priority ready task to run.
  - If higher-priority task becomes ready, lower-priority task is preempted.
- Tasks block when waiting for events, resources.
  - ISRs can cause tasks to become unblocked.
  - Tasks can delay themselves for fixed time intervals.
- RTOS contains code to
  - Create tasks, block and unblock tasks, schedule tasks, allow tasks and ISRs to communicate, etc.

RTOS architecture



RTOS characteristics

- Priorities available
  - Interrupts are serviced in priority order
  - Tasks are scheduled in priority order; lower priority tasks preempted
- Worst-case response time for highest-priority task
  - Sum of ISR execution times (since other tasks preempted)
- Advantages:
  - Stability when code changes (e.g. adding a lower-priority task)
  - Many choices of commercial RTOS available
- Disadvantages:
  - Runtime overhead of RTOS
  - Software complexity (some in RTOS, some in using RTOS correctly)

Non-trivial multi-threaded programs are incomprehensible to humans.
Edward A. Lee

Selecting an architecture

1. Select the simplest architecture that will meet current *and future* response time requirements.
2. If application has difficult response-time requirements, lean toward using an RTOS:
   - Many to choose from, debugging support, libraries, etc.
3. Consider constructing hybrid architecture – examples:
   - RTOS where one task does polling
   - Round robin with interrupts: main loop polls slower HW directly