SOFTWARE ARCHITECTURES

Embedded Software Design

熊博安
國立中正大學資訊工程研究所
pahsiung@cs.ccu.edu.tw

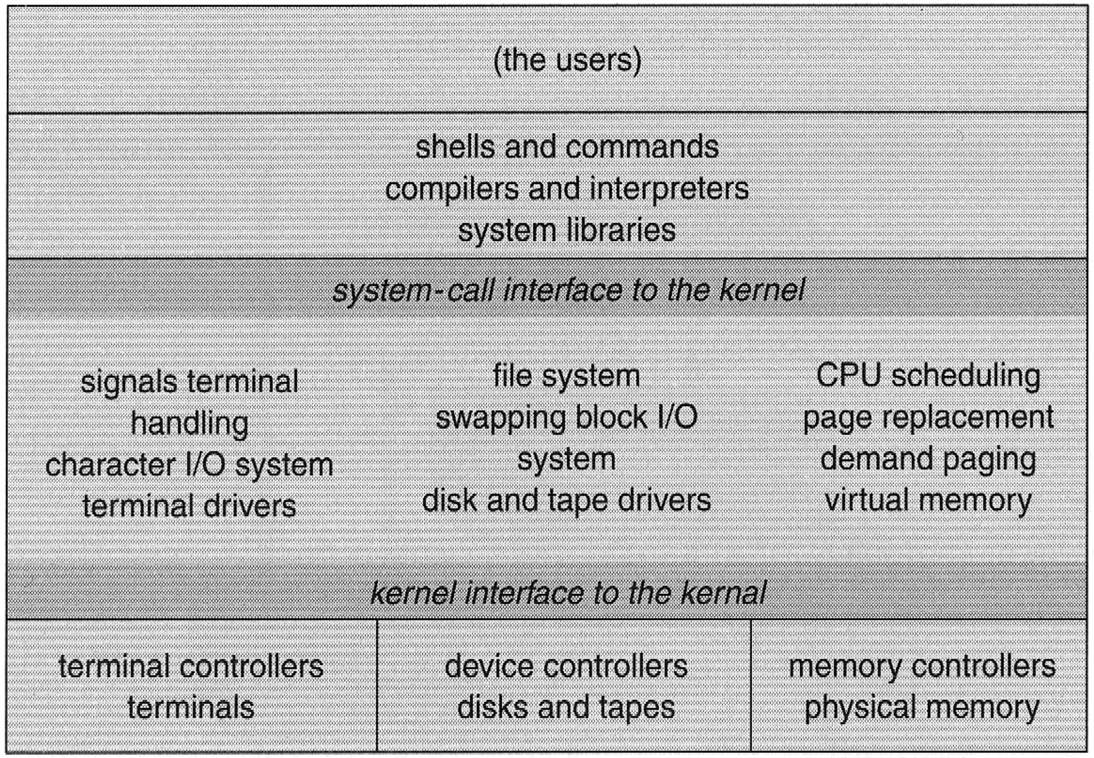
Textbook: An Embedded Software Primer,
David E. Simon, Addison Wesley

Contents

- Round-Robin
- Function-Queue Scheduling
- Real-Time Operating Systems
- Selecting an Architecture

Software Architectures

- When you are designing embedded software, what architecture will be the most appropriate for a given system?



Decision Factors

- The most important factor
  - how much control you need to have over system response.
- Good response
  - Absolute response time requirements
  - The speed of your microprocessor
  - and the other processing requirements
- Few, loose reqts → simple architecture
- Many, stringent reqts → complex architecture

Some Examples

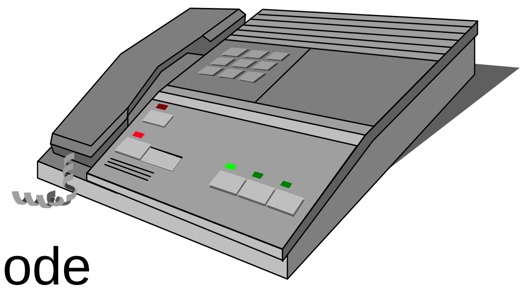
- The control of an air conditioner
  - This system can be written with a very simple software architecture.
  - The response time can be within a number of tens of seconds.
  - The major function is to monitor the temperature readings and turn on and off the air conditioner.
  - A timer may be needed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

Some Examples

- The software design of the control of an air conditioner
  - A simple assembly program for a low-end microprocessor
  - Inputs
    - Input buttons
    - Temperature readings
    - Timer readings
  - Output
    - The on-off control of the air conditioner
    - The power control

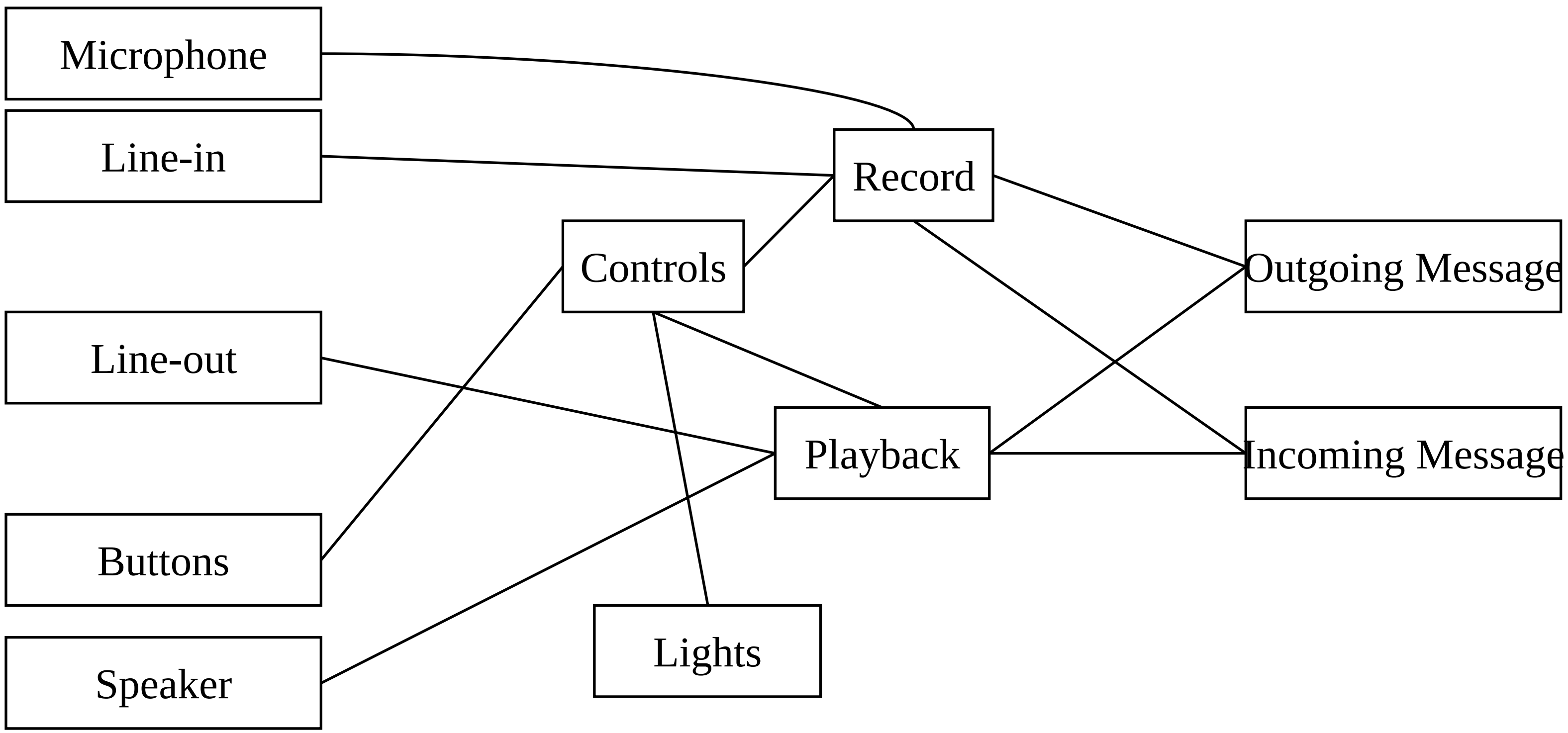
Some Examples

- Digital telephone answering machine
  - A telephone answering machine with digital memory, using speech compression.
  - The performance and functions
    - It should be able to record about 30 minutes of total voice.
    - Voice data are sampled at the standard telephone rate of 8kHz.
    - OGM of up to 10 seconds
    - Three basic modes
      - default/play back/OGM editing mode



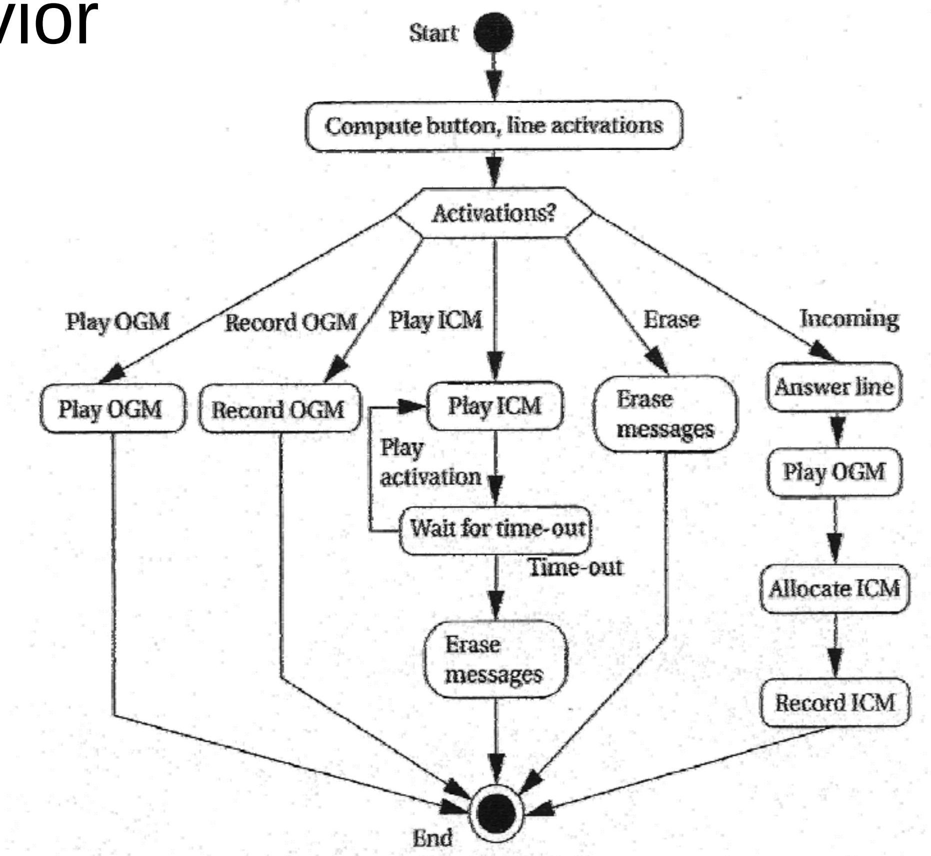
Some Examples

- The class diagram for the answering machine



Some Examples

- The state diagram for the controls activate behavior

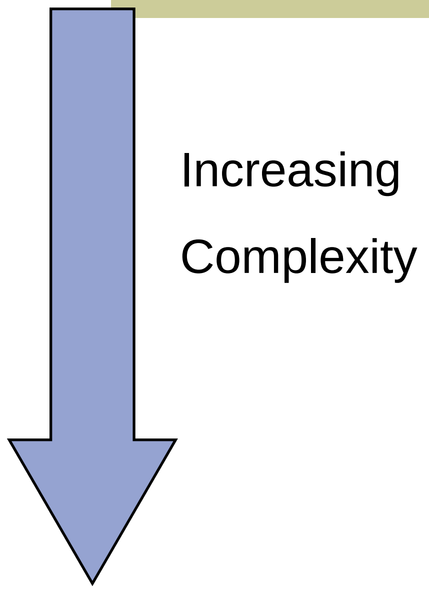


Some Examples

- The software design for the answering machine
  - It must respond rapidly to many different events.
  - It has various processing requirements.
  - It has different deadlines and different priorities.
- A more complex architecture

4 Basic SW Architectures

- Round-Robin
- Round-Robin with Interrupts
- Function-Queue Scheduling
- Real-Time Operating System



Round-Robin Architecture

- Very simple
- No interrupts
- No shared data
- No latency concerns
- Main loop:
  - checks each I/O device in turn
  - services any device requests
- E.g.: Digital Multimeter

Round-Robin Architecture

- The simplest 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
      }
   }
}
```

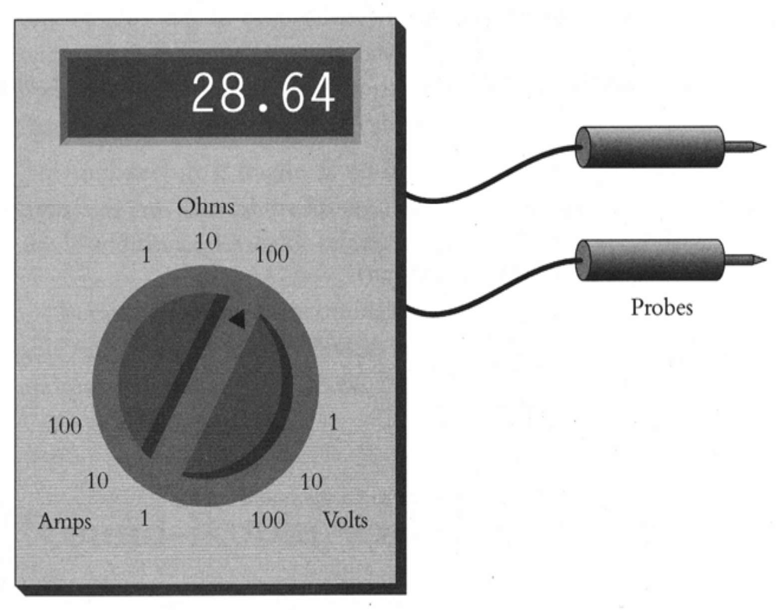
Device A

Device B

Device Z

An Application

- Digital multimeter
  - Measures
    - R, I, and V readings
  - I/O
    - Two probes
    - A digital display
    - A rotary switch
  - Function
    - Continuous measurements
    - Update display



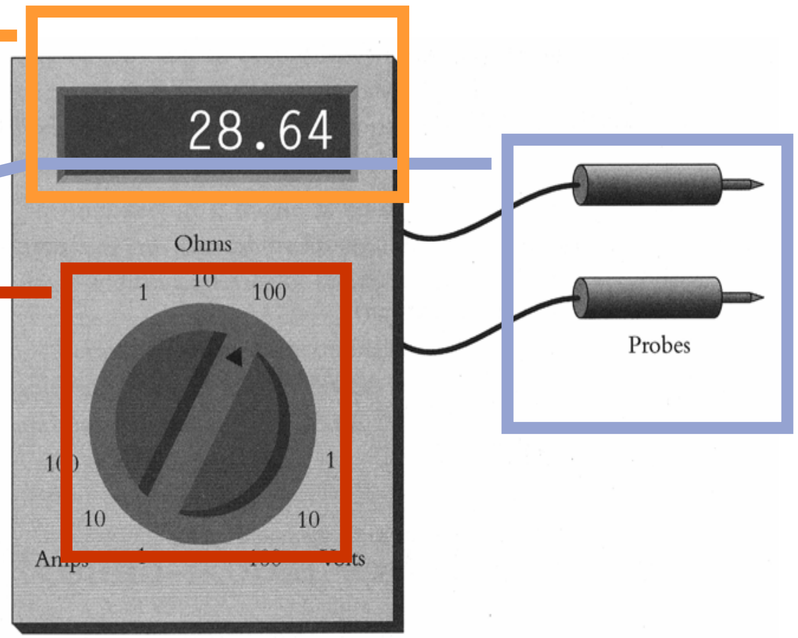
Digital Multimeter

- The possible pseudo-code

```
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
   }
}
```



Digital Multimeter

- Round-robin works well for this system because:
  - only 3 I/O devices
  - no lengthy processing
  - no tight response requirements
- 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!

Discussion

- Advantages
  - Simplicity
  - Low development cost
  - Short development cycle
- Shortcomings
  - This architecture cannot handle complex problems.

Shortcomings

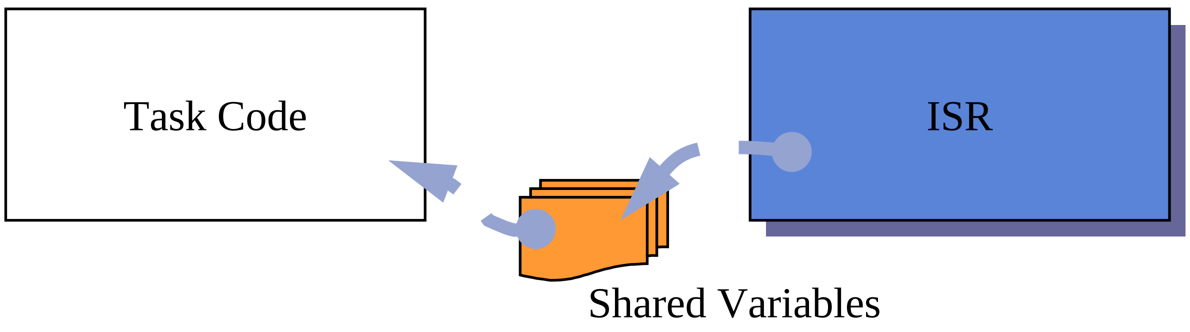
- If any one device needs response in less time
  - Two possible improvements for the RR architecture
    - Squeezing the loop
    - Carefully arranging the sequence (A,Z,B,Z,C,Z,D,Z,…)
- If there is any lengthy processing to do
  - Every other event is also postponed.
- This architecture is fragile
  - A single additional device or requirement may break everything.

Round-Robin with Interrupts

- A little bit more control
  - In this architecture,
    - ISRs deal with the very urgent needs of the hardware and set corresponding flags
    - the main loop polls the flags and does any follow-up processing
- ISR can get good response
- All of the processing that you put into the ISR has a higher priority than the task code

A Little Bit More Control

- You can control the priorities among the ISR as well.
- The software is more event-driven.



The Architecture

- Two main parts

Interrupt Service Routines

```
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;
}
```

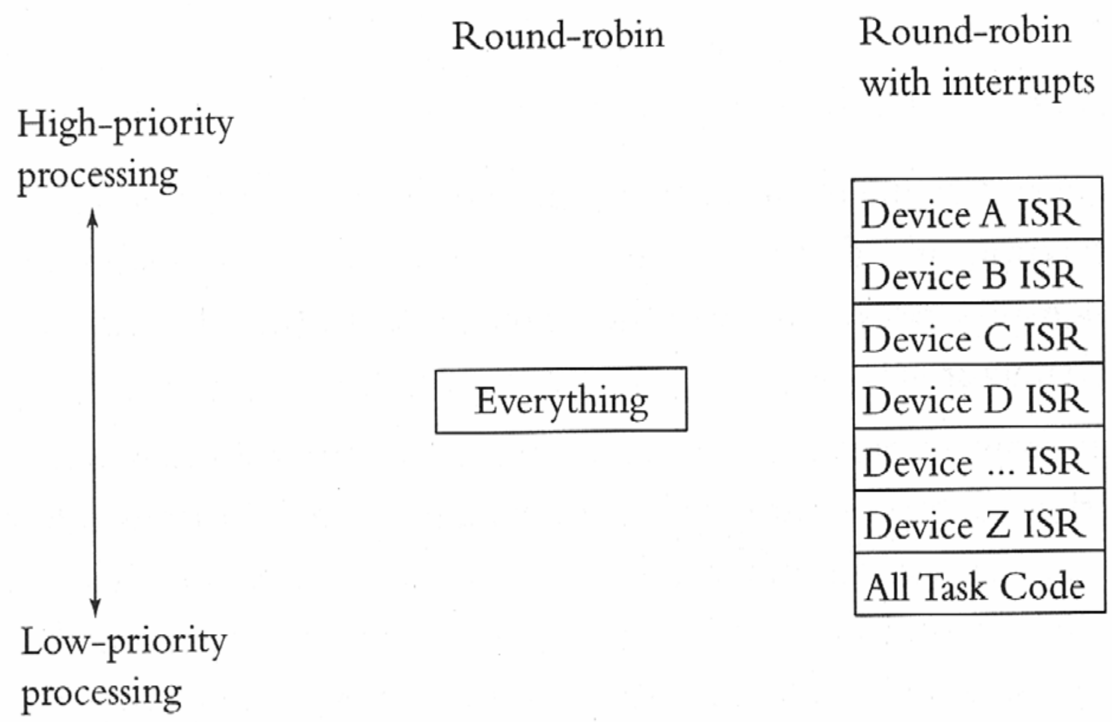
The main loop

```
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
      }
   }
}
```

RR vs. RR-INT

- Priority levels

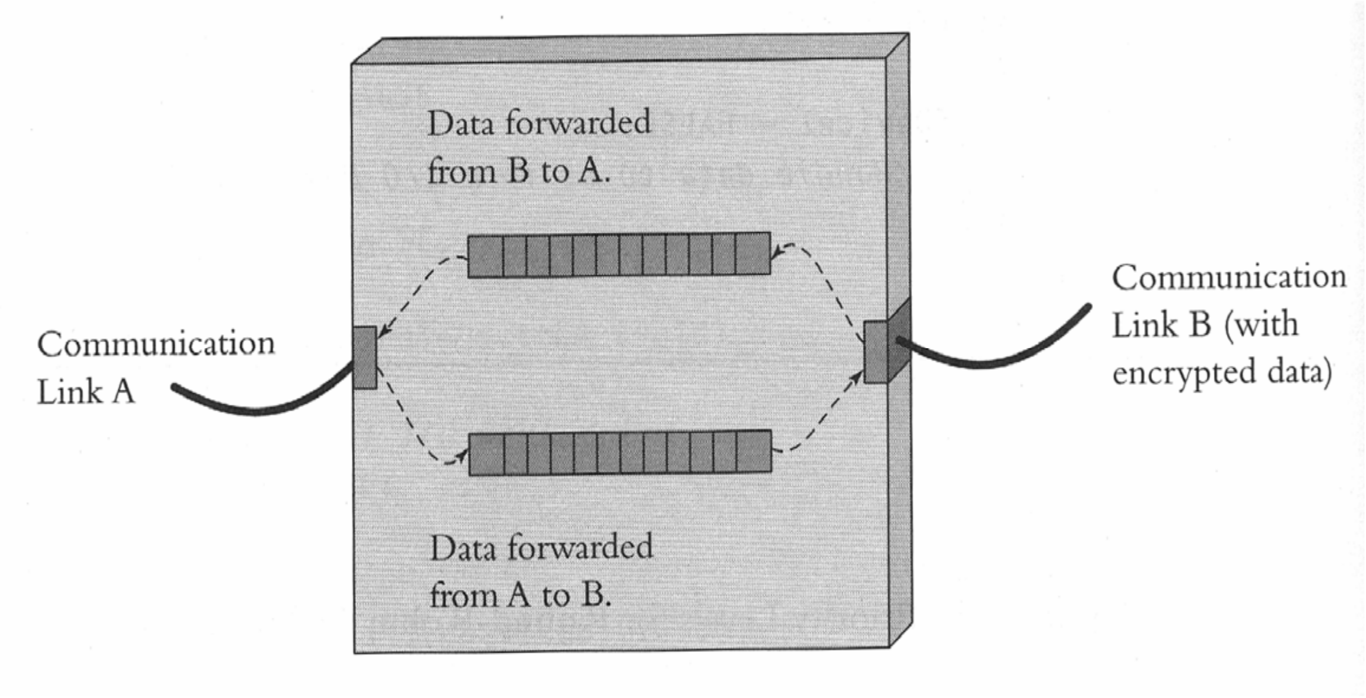


Discussion

- Advantage
  - The processing is more effectively.
- Disadvantage
  - All of the shared-data problems can potentially jump and bite you.

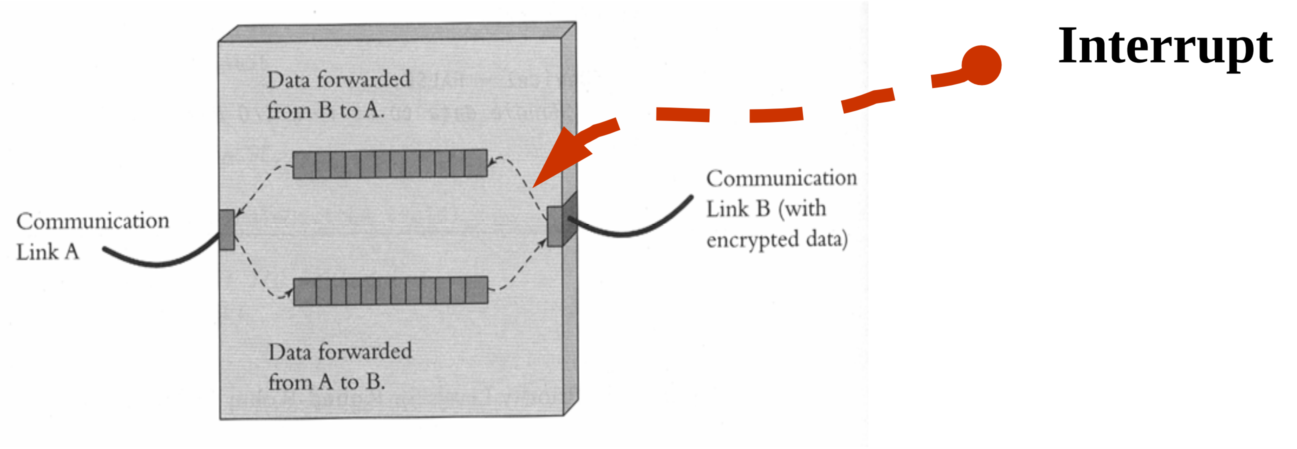
An Example of A Simple Bridge

- A device with two ports on it that forwards data traffic received on the first port to the second and vice versa.



Some Assumptions

- Whenever a character is received on one of the communication links, it causes an interrupt.
- The Interrupt must be serviced reasonably quickly.



Some Assumptions

- The microprocessor must write characters to the I/O hardware one at a time.
- The I/O transmitter hardware on that communication link will be busy while it sends the character.
- Then, it will interrupt to indicate that it is ready for the next character.

Some Assumptions

- We have routines that will
  - read characters from and write characters to queues and
  - test whether a queue is empty or not
- These routines can be called from ISRs as well as from the task code.
- They deal correctly with the shared-data problems.
- Encrypt / decrypt one character at a time

Possible Code

- Data structures

```
#define QUEUE_SIZE 100

typedef struct
{
   char chQueue[QUEUE_SIZE];
   int iHead;       /* Place to add next item */
   int iTail;       /* Place to read next item */
} QUEUE;

static QUEUE qDataFromLinkA;
static QUEUE qDataFromLinkB;
static QUEUE qDataToLinkA;
static QUEUE qDataToLinkB;

static BOOL fLinkAReadyToSend = TRUE;
static BOOL fLinkBReadyToSend = TRUE;
```

Possible Code

- Interrupt service routines

```
void interrupt vGotCharacterOnLinkA (void)
{
   char ch;
   ch = !! Read character from Communications Link A;
   vQueueAdd (&qDataFromLinkA, ch);
}

void interrupt vGotCharacterOnLinkB (void)
{
   char ch;
   ch = !! Read character from Communications Link B;
   vQueueAdd (&qDataFromLinkB, ch);
}

void interrupt vSentCharacterOnLinkA (void)
{
   fLinkAReadyToSend = TRUE;
}

void interrupt vSentCharacterOnLinkB (void)
{
   fLinkBReadyToSend = TRUE;
}
```

Interrupts upon receiving characters

Interrupts upon sending characters

Possible Code

- The main loop

```
void main (void)
{
   char ch;

   /* Initialize the queues */
   vQueueInitialize (&qDataFromLinkA);
   vQueueInitialize (&qDataFromLinkB);
   vQueueInitialize (&qDataToLinkA);
   vQueueInitialize (&qDataToLinkB);

   /* Enable the interrupts. */
   enable ();

   while (TRUE)
   {
      vEncrypt ();
      vDecrypt ();
      if (fLinkAReadyToSend && fQueueHasData (&qDataToLinkA))
      {
         ch = chQueueGetData (&qDataToLinkA);
         disable ();
         !! Send ch to Link A
         fLinkAReadyToSend = FALSE;
         enable ();
      }
      if (fLinkBReadyToSend && fQueueHasData (&qDataToLinkB))
      {
         ch = chQueueGetData (&qDataToLinkB);
         disable ();
         !! Send ch to Link B
         fLinkBReadyToSend = FALSE;
         enable ();
      }
   }
}
```

Possible Code

- encrypt() and decrypt()

```
void vEncrypt (void)
{
    char chClear;
    char chCryptic;

    /* While there are characters from port A . . .*/
    while (fQueueHasData (&qDataFromLinkA))
    {
        /* . . . Encrypt them and p
        chClear = chQueueGetData (&
        chCryptic = !! Do encryptio
        vQueueAdd (&qDataToLinkB, c
    }
}
```

```
void vDecrypt (void)
{
    char chClear;
    char chCryptic;

    /* While there are characters from port B . . .*/
    while (fQueueHasData (&qDataFromLinkB))
    {
        /* . . . Decrypt them and put them on queue for port A */
        chCryptic = chQueueGetData (&qDataFromLinkB);
        chClear = !! Do decryption (no one understands this code)
        vQueueAdd (&qDataToLinkA, chClear);
    }
}
```

Bridge code

- Interrupt routines:
  - read characters from hardware
  - put them into queues: qDataFromLink[AB]
- Main routine:
  - reads data from queues: qDataFromLink[AB]
  - encrypts and decrypts data
  - write data to queues: qDataToLink[AB]
- I/O Hardware:
  - 2 vars to keep track: fLink[AB]ReadyToSend

Bridge code

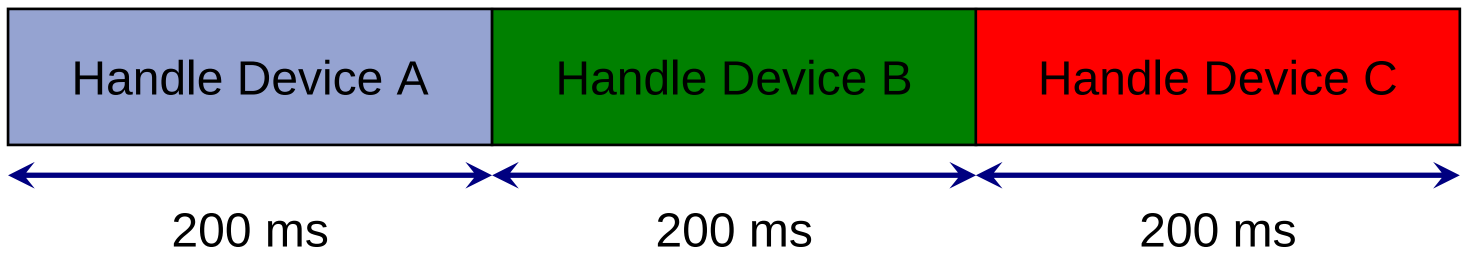
- Shared-Data Problem:
  - disable / enable interrupts
- Response Time:
  - Characters received from hardware by interrupt routines, thus HIGHER priority
  - moving characters among queues, encrypting, decrypting, sending them out, etc. are of LOWER priority
  - Burst of characters will not overrun system

Cordless Bar-Code Scanner

- Get data from laser reading bar codes
- Send data out on the radio
- Only real response requirements
  - Service hardware quickly enough
- Thus, round-robin-with-interrupts is sufficient

Characteristics of RR-with-Interrupts

- Shortcomings:
  - Not as simple as RR
  - All task code executes at the same priority



- C must wait 400 ms
- If C cannot wait that long → system wrong

Characteristics of RR-with-Interrupts

- Possible Solutions:
- Move task code for C into interrupt routine
  - ISR exec time will increase by 200 ms
  - Lower priority devices will have to wait
- Change sequence: A, C, B, C, D, E, C, …
  - Response time for C improves
  - Response times for other devices may be not acceptable
  - Tuning → Fragile

Characteristics of RR-with-Interrupts

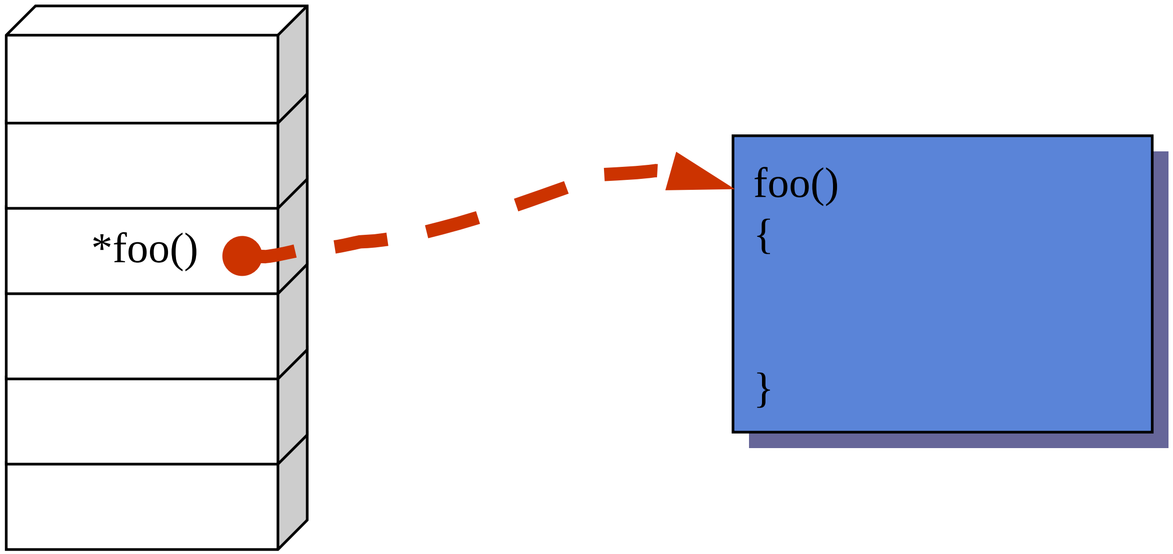
- Worst-case response time for task code for any given device
  - RR loop passes task for that device
  - Interrupt for that device occurs immediately after loop passes
- Worst-case response time = Sum of task code execution times of all other devices

Examples of Systems for which RR-with-Interrupts does not work well

- Laser printer
  - Calculating locations for black dots is very time consuming
- Underground tank-monitoring system
  - Calculating gasoline level in tank is very time consuming
- Processor hog → Task code gets stuck

Function Queue Scheduling Architecture

- In this architecture, the interrupt service routines add *function pointers* to a *queue of function pointers*.



Function-Queue Scheduling

- Interrupt routines:
  - add function pointers to a queue
- Main routine:
  - reads pointers from queue
  - calls the functions
- Main need not call functions in the order of occurrence
- A priority scheme can be used for ordering the function pointers

The Framework of FQS

- Three parts

```
!! 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 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
}
```

Worst-case Execution Time

- Worst wait for highest-priority task code function = length of longest task code function
- (Better than RR-with-Interrupts)
- Trade-off
  - Response for lower-priority task code functions may get worse
- Problem
  - Starvation: lower-priority task code may never get executed!

Real-Time Operating System

- Interrupt routines
  - take care of most urgent operations
  - “signal” that there is work for task code to do
- Differences with other architectures:
  - Signaling between interrupt routines and task code is handled by RTOS (no need of shared variables)
  - No main loop deciding what to do next, RTOS decides the scheduling
  - RTOS can suspend on task code subroutine to run another

A Paradigm

- The sample code

```
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 or from I/O Device A
    }
}

void Task2 (void)
{
    while (TRUE)
    {
        !! Wait for Signal Y
        !! Handle data to or from I/O Device B
    }
}
```

Worst case execution

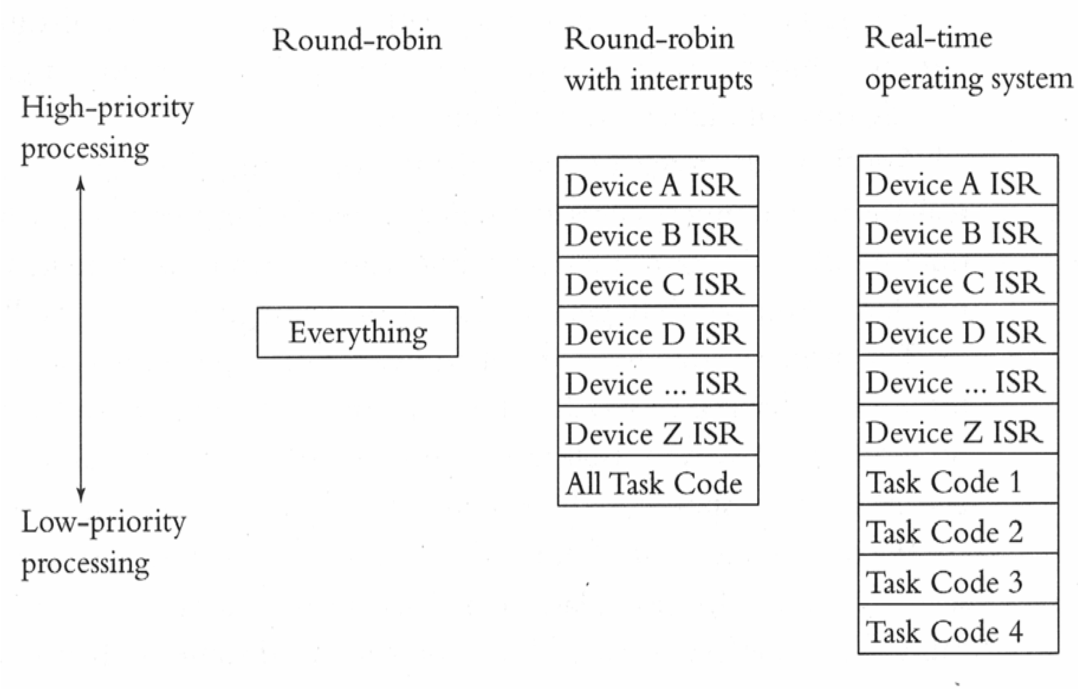
- Suppose Task1 has higher priority
- Suppose Task2 is running
- Interrupt occurs and vHandleDeviceA sets signal X
- Task2 is suspended
- Task1 is started
- Worst case execution time for the highest priority task code subroutine = 0 (+ ISR time)

Advantages / Disadvantages of RTOS

- Changes to any task code in the RR or function-queue scheduling schemes have a global effect: affects all tasks
- Changes to lower priority task code in RTOS does not affect response time of higher priority tasks
- RTOS are widely available, immediate solutions to your response problems
- Disadvantage: RTOS itself needs some processing time, throughput is affected

Priority Levels

- A comparison



Selecting an Architecture

- Select the simplest architecture that will meet your response requirements
- If your response constraints requires an RTOS, then buy one and use it because there are also several debugging tools for it
- You can create hybrids of the architectures. In RTOS or RR, main task code can poll slow hardware devices that do not need fast response (leaving interrupts for faster hardware)

Characteristics of Architectures

	Priorities Available	Worst Response Time for Task Code	Stability of Response When the Code Changes	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 priority	Total of execution time for all task code (plus execution time for interrupt routines)	Good for interrupt routines; poor for task code	Must deal with data shared between interrupt routines and task code
Function-queue scheduling	Interrupt routines in priority order, then task code in priority order	Execution time for the longest function (plus execution time for interrupt routines)	Relatively good	Must deal with shared data and must write function queue code
Real-time operating system	Interrupt routines in priority order, then task code in priority order	Zero (plus execution time for interrupt routines)	Very good	Most complex (although much of the complexity is inside the operating system itself)