

Survey of software architectures: round-robin

Reference: Simon chapter 5

Survey of software architectures

- We are going to discuss the four basic software approaches we can implement in an embedded systems.
- A software architecture is just how our source code is arranged.
- The most important factor that determines which architecture will be the most appropriate is how much control you need to have over system response.
- A system that must respond rapidly to many different events ought to be implemented very differently from a system with just a single event and very small response time.
- Four architectures: round-robin, round-robin with interrupts, function-queue-scheduling, and real-time operating system.

Round robin

Round robin

- Round-robin is the simplest imaginable architecture.
- There are no interrupts.
- The main loop simply checks each of the I/O devices in turn and services any that need service.
- No interrupts, no shared data, no latency concerns

```
void main ()
{
    while (TRUE)
    {
        if (!! I/O Device A needs service) {
            !! Take care of I/O Device A
        }

        if (!! I/O Device B needs service) {
            !! Take care of I/O Device B
        }
        (...)

        if (!! I/O Device Z needs service) {
            !! Take care of I/O Device Z
        }
    } //end while (TRUE)
} //end void main()
```

Round robin usage example

- I want to create a software program for an embedded system that runs a digital multimeter.
- A digital multimeter measures resistance or current or voltage in several different ranges.
- A multimeter has two measuring probes, a digital display, and a big rotary switch that selects measurement and range.



Round robin inside a digital multimeter



- The system makes continuous measurements and changes the display to reflect the most recent measurement.
- Each time around its loop, the software checks the position of the rotary switch, branches to the appropriate code selection and writes the results to the display.

Digital multimeter code

```
void vDigitalMultiMeterMain() {
enum {OHMS1, OHMS10, ...VOLTS100} eSwitchPosition;
while (TRUE)
{
    eSwitchPosition = !! Read switch position
    switch (eSwitchPosition) {
        case OHMS_1:
            !! Read hardware to measure ohms
            break;
        case OHMS_10:
            !! Read hardware to measure ohms
            break;
        (...)
        case VOLTS_100:
            !! Read hardware to measure volts
            break;
    }
    !! Write result to display
}
}
```

Problems with round robin

- Round-robin architecture has only one advantage over other architectures: simplicity!
- It has some problems that make it inadequate for many systems:
 1. If any one device needs response in a limited time, the system may not work.
 2. The system can respond really slowly.
 3. The architecture is very susceptible to code changes.
- Because of these shortcomings, a round-robin architecture is probably suitable only for very simple devices .

Issue #1: If you need fast response time round robin is not the way to go

- If device Z can wait no longer than 7 milli-seconds for service
- If device A and B each take 5 milli-seconds to run.
- If all three devices need service, and the processor starts with device A, then the processor will not have time to reach device C quickly enough.
- ... Sure you can squeeze some msec by changing device order.

```
void main ()
{
    while (TRUE)
    {
        if (!! I/O Device A needs service){
            !! Take care of I/O Device A
        }

        if (!! I/O Device B needs service) {
            !! Take care of I/O Device B
        }
        (...)

        if (!! I/O Device Z needs service){
            !! Take care of I/O Device Z
        }
    } //end while (TRUE)
} //end void main()
```

Issue #2: Slow system response

- If it takes 3 seconds for a particular function to run...
- ... then the system response to the rotary switch could be as bad as 3 seconds.
- Sure, it works but its a lousy product.



```
void vDigitalMultiMeterMain() {
enum {OHMS1,OHMS10,...VOLTS100) eSwitchPosition;
while (TRUE)
{
    eSwitchPosition = !! Read switch position
    switch (eSwitchPosition) {
        case OHMS_1:
            !! Read hardware to measure ohms
            break;
        case OHMS_10:
            !! Read hardware to measure ohms
            break;
        (...)
        case VOLTS_100:
            !! Read hardware to measure volts
            break;
    }
    !! Write result to display
}
}
```

Issue #3: Round robin is a fragile architecture

- Even if we manage to tune up the system so that the microprocessor gets around the loop quickly enough to satisfy all requirements...
- Adding a single additional device may break everything.

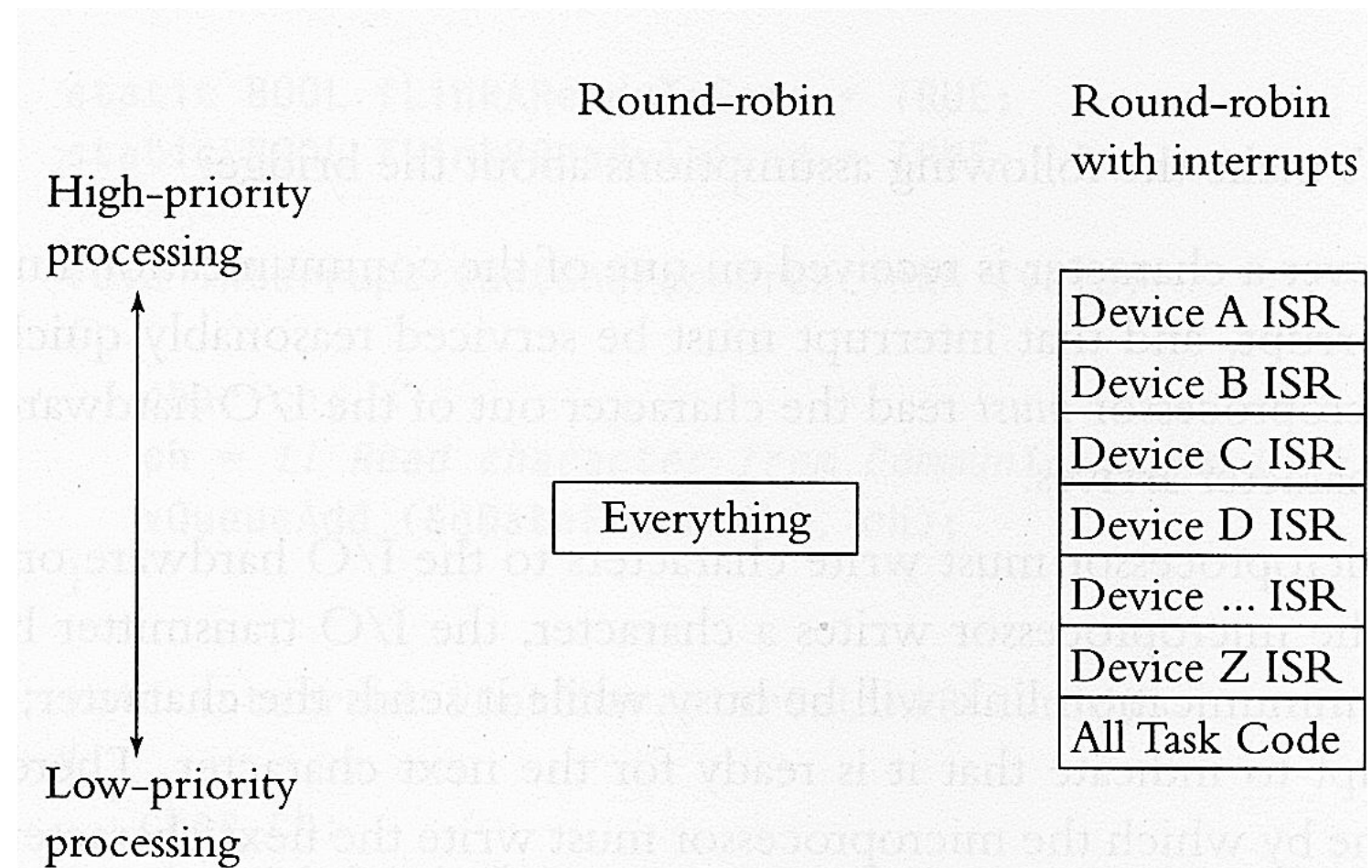
Round robin with interrupts

Round robin with interrupts

- Interrupt routines deal with the very urgent needs of the hardware and then set flags
- The main loop polls the flags and does any follow-up processing required by the interrupts.
- This architecture gives you more control over priorities, since the processor can now stop the main loop and resolve the interrupts.

Comparing round robin with interrupts and without interrupts

- Advantages: Same advantages of use interrupts over polling (priority control)
- Disadvantages: a lot of data is shared amongst many different interrupt routines which will potentially create shared data problems.



Round robin with interrupts example

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
(...)
BOOL fDeviceZ = FALSE;

void interrupt vHandleDeviceA ()
{
    !! Take care of I/O Device A
    fDeviceA = TRUE;
}

void interrupt vHandleDeviceB ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}

void interrupt vHandleDeviceZ ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}
```

```
void main ()
{
    while (TRUE) {
        if (fDeviceA) {
            fDeviceA = FALSE;
            !! Handle I/O Device A
        }
        if (fDeviceB) {
            fDeviceB = FALSE;
            !! Handle I/O Device B
        }
        (...)
        if (fDeviceZ) {
            fDeviceZ = FALSE;
            !! Handle I/O Device Z
        }
    } //end of while (TRUE)
}
```

Round robin with interrupts example

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
(...)
BOOL fDeviceZ = FALSE;

void interrupt vHandleDeviceA ()
{
    !! Take care of I/O Device A
    fDeviceA = TRUE;
}

void interrupt vHandleDeviceB ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}

void interrupt vHandleDeviceZ ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}
```

```
void main ()
{
    while (TRUE) {
        if (fDeviceA) {
            fDeviceA = FALSE;
```

- Whenever an I/O is ready, an interrupt will occur.
- This interrupt will set a boolean variable which mentions that some I/O operation needs to be done.

```
fDeviceZ = FALSE;
    !! Handle I/O Device Z
}

//end of while (TRUE)
}
```

Round robin with interrupts example

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
(...)
BOOL fDeviceZ = FALSE;

void interrupt vHandleDeviceA ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}

void interrupt vHandleDeviceZ ()
{
    !! Take care of I/O Device B
    fDeviceB = TRUE;
}
```

- The main loop checks the boolean variable to see if there is some I/O operation that needs to be done.

```
void main ()
{
    while (TRUE) {
        if (fDeviceA) {
            fDeviceA = FALSE;
            !! Handle I/O Device A
        }
        if (fDeviceB) {
            fDeviceB = FALSE;
            !! Handle I/O Device B
        }
        (...)
        if (fDeviceZ) {
            fDeviceZ = FALSE;
            !! Handle I/O Device Z
        }
    } //end of while (TRUE)
}
```

Isn't this the same as normal round-robin? Whats the point

- The point is that I can add “important” task code into the interrupt service routine.
- This will guarantee that “important devices” are dealt first.
- The problem is that, the lower priority devices will suffer increased response times.

Another round-robin-with-interrupts example: cordless bar-code scanner

- The bar-code scanner is essentially a device that gets the data from the laser that reads the bar codes and sends that data out on the radio.
- In this system the only real response requirements are to service the hardware quickly enough. Reading the data from the scanner is the priority...The rest may take its time.
- The task code processing will get done quickly enough in a round-robin loop.

Timing example

- If A,B and Z all take 200msec each
- If A,B and Z all interrupt at the same time when the micro-processor is here, the task code for Z may have to wait 400msec until it can be executed.
- The only way to avoid this is by putting the task code for device Z into an interrupt routine with an higher priority.

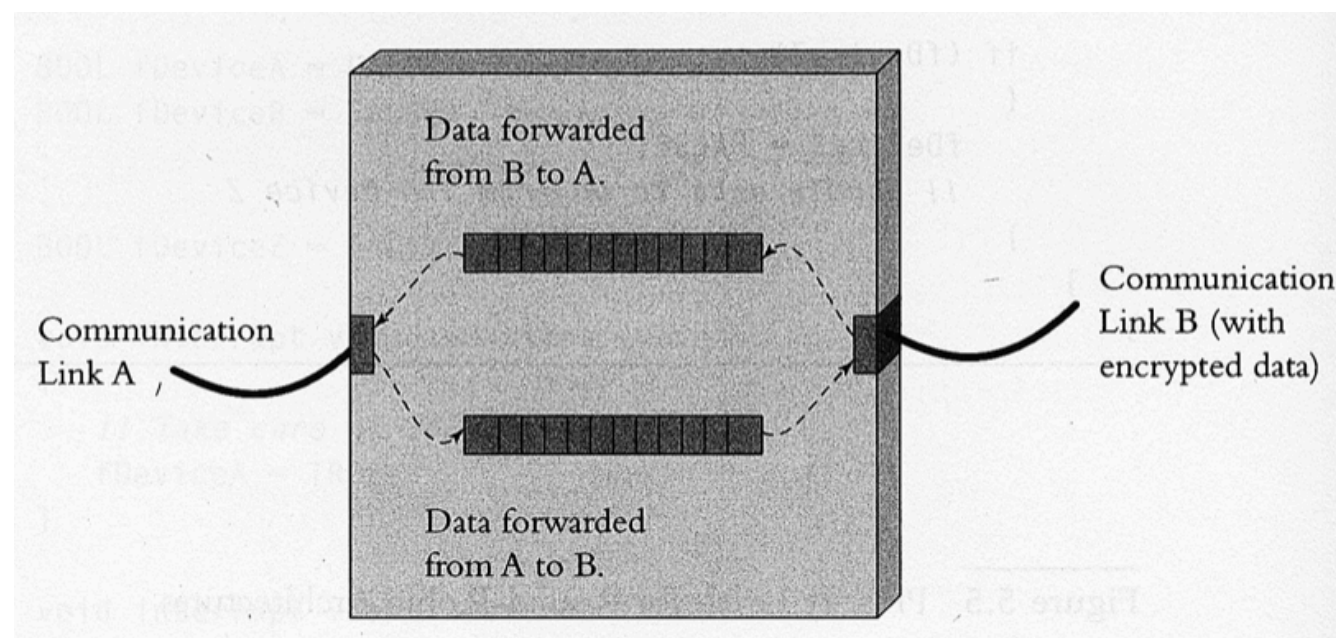
```
void main  ()
{
  while  (TRUE) {
    if  (fDeviceA) {
      fDeviceA = FALSE;
      !! Handle I/O Device A
    }
    if  (fDeviceB) {
      fDeviceB =  FALSE;
      !! Handle I/O Device B
    }
    (...)
    if  (fDeviceZ) {
      fDeviceZ = FALSE;
      !! Handle I/O Device Z
    }
  } //end of while (TRUE)
}
```

Round-robin-with-interrupts architecture is not perfect

- Round-robin-with-interrupts architecture does not work well in the following systems:
- A laser printers; since calculating the locations where the black dots go is very time-consuming. Also, laser printers have many other processing requirements, so it is impossible to make sure that low-priority interrupts are serviced quickly enough.
- A tank-monitoring system; one way to calculate how much water is in the tanks is to put all code inside interrupt service routines. This is not a good approach and a more sophisticated architecture is required for this system as well.

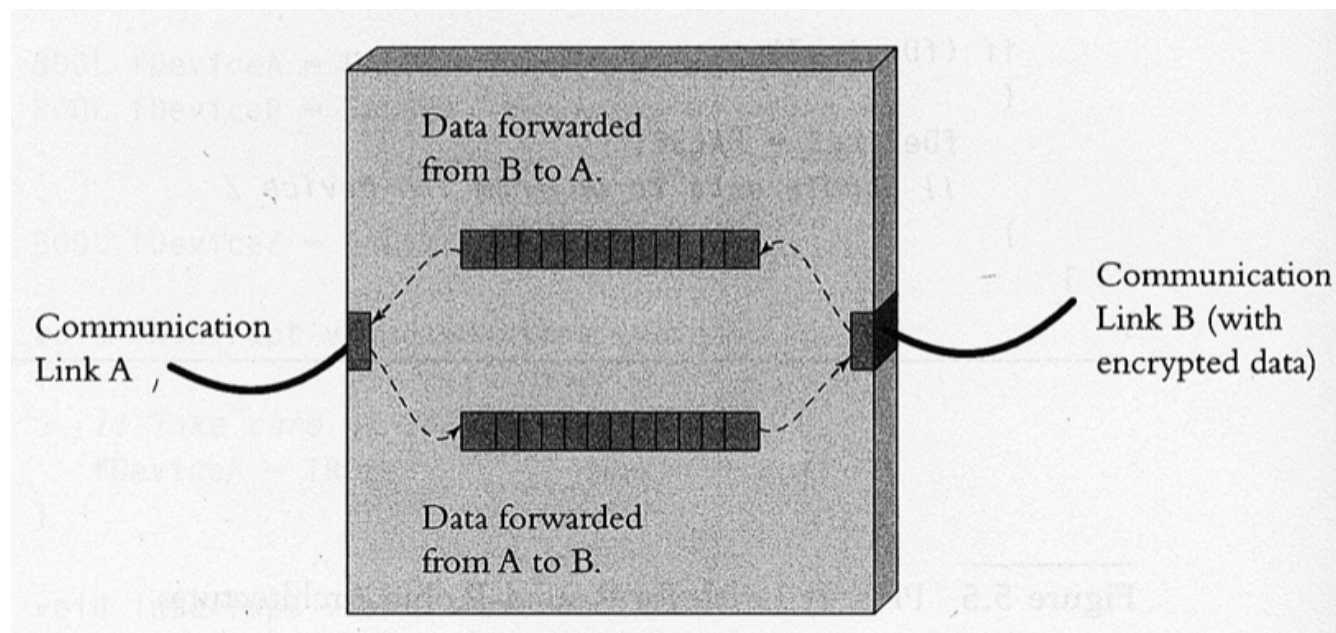
Round robin with interrupts: a system example

- The round-robin-with-interrupts architecture is suitable for many systems, ranging from the fairly simple to the surprisingly complex.
- One example is the communications bridge, a device with two ports that forwards data traffic received on the first port to the second and vice versa.
- Assume the data on one of the ports is encrypted and that it is the job of the bridge to encrypt and decrypt the data as it passes through it.



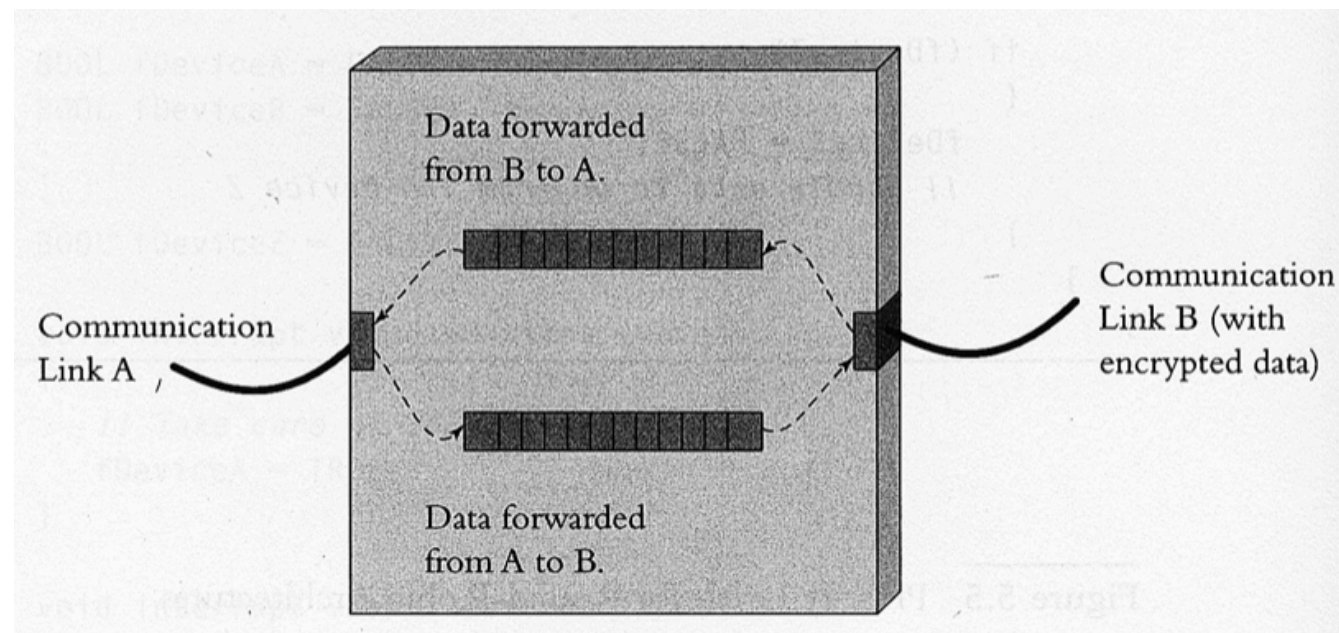
Bridge assumption #1

- Whenever a character is received at one end, it causes an interrupt.
- That interrupt must be serviced reasonably quickly to read the character out of the I/O hardware before the next character arrives.



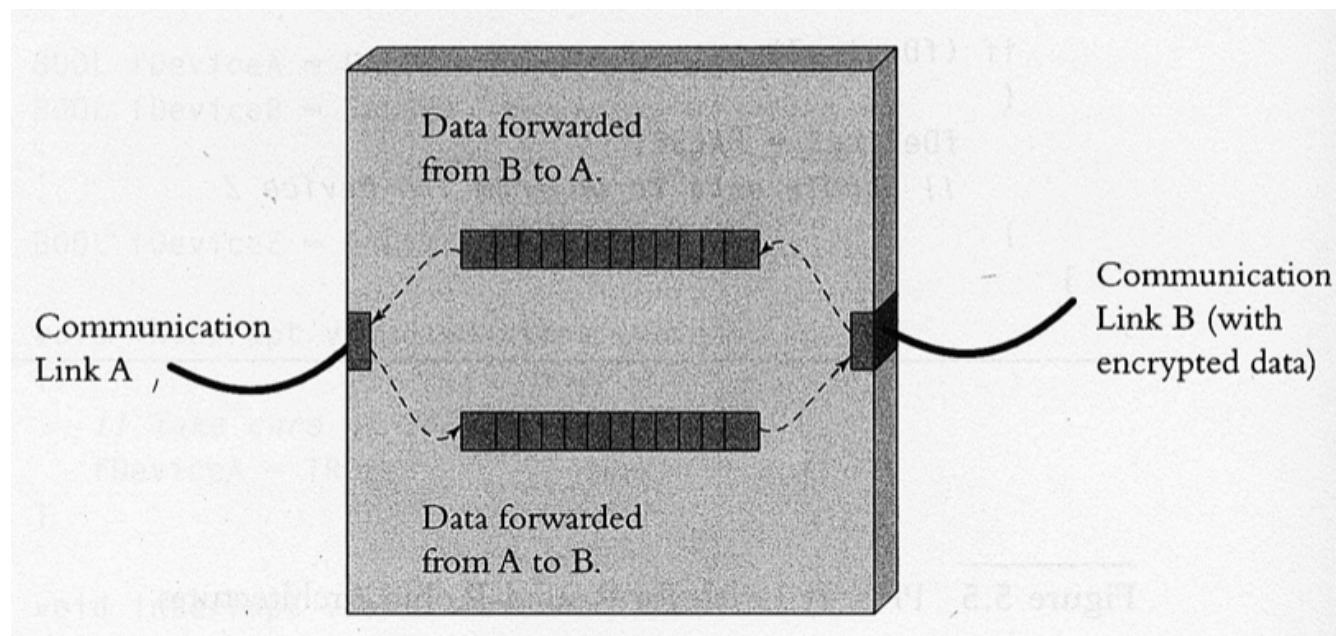
Bridge assumption #2

- The microprocessor must write to the I/O one character at a time.
- When a character is being written, the communication link is busy.
- An interrupt will indicate that the character is done transmitting (and the link is no longer busy).
- There is no hard deadline by which the character must be written in hardware.



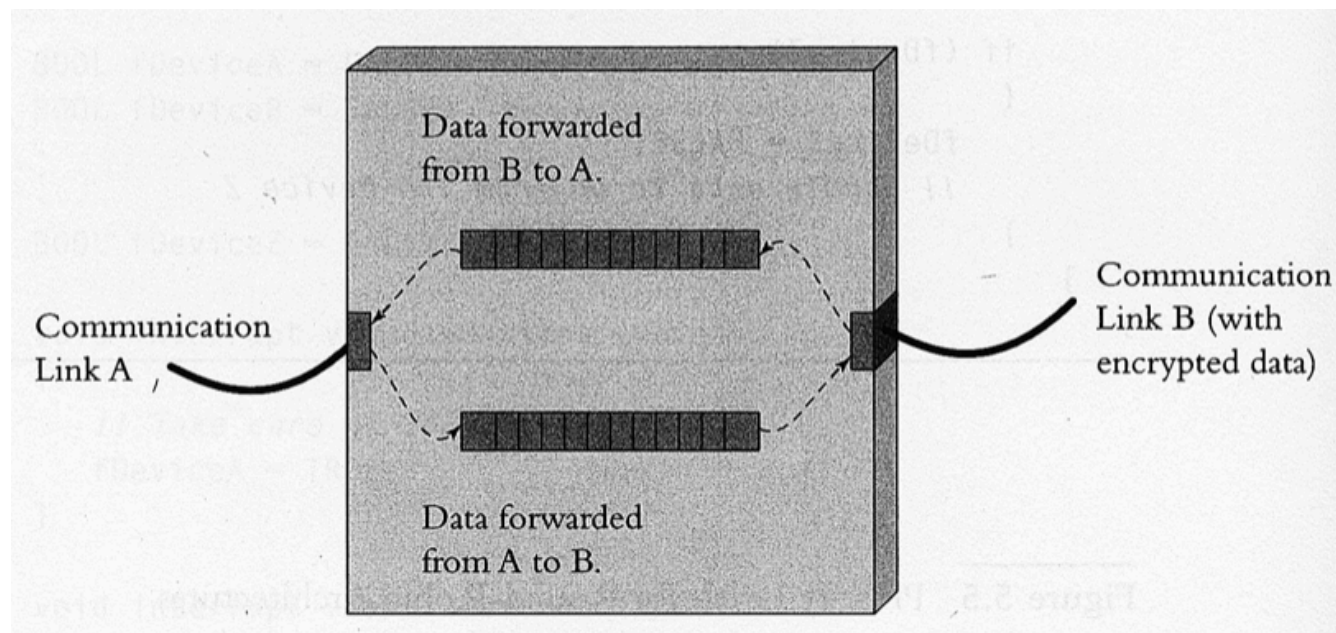
Bridge assumption #3

- We have routines that will read and write characters to queues, and to test if a queue is empty.
- We call these routines from the task code as well as from the interrupt routines.
- These routines deal with shared data problems appropriately.



Bridge assumption #4

- The encryption routine can encrypt just a single character at a time.
- The decryption routine can decrypt just a single character at a time.



Code (Part #1)

```
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;
```

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

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

void interrupt vSentCharacterOnLinkA () {
    fLinkAReadyToSend = TRUE;
}

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


Code (Part #1)

- The main loop checks the boolean variable to see if there is some I/O operation that needs to be done.

```
static QUEUE qDataFromLinkA;  
static QUEUE qDataFromLinkB;
```

- The interrupt routine puts the received character on the appropriate queue.

```
TRUE;  
TRUE;
```

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

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

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

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



Code (Part #2)

```
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 character to Link A
            fLinkAReadyToSend = FALSE;
            enable ();
        }
        if (fLinkBReadyToSend &&
            fQueueHasData (&qDataToLinkB))
        {
            ch = chQueueGetData (&qDataToLinkB);
            disable ();
            !!Send ch    to Link B
            fLinkBReadyToSend = FALSE;
            enable ();
        }
    } //end of while(TRUE)
} //end of of main(void)
```

Code (Part #2)

```
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 character to Link A
            fLinkAReadyToSend = FALSE;
            enable ();
        }
        if (fLinkBReadyToSend &&
            fQueueHasData (&qDataToLinkB))
        {
            ch = chQueueGetData (&qDataToLinkB);
            disable ();
            !!Send character to Link B
            fLinkBReadyToSend = FALSE;
            enable ();
        }
    } //end of while(TRUE)
} //end of of main(void)
```



•Task code calls vEncrypt() and vDecrypt() which reads queues, encrypt/decrypt data and updates destination queues.

Code (Part #2)

```
void main (void)
{
```

- `fLinkAReadyToSend` keep track of whether the I/O is ready to send characters over the two communication links.
- `FALSE` means that the I/O hardware is now busy.

```
vDecrypt ();
```

- When the character is ready to be sent, an interrupt will set `fLinkAReadyToSend` to `TRUE`.

```
    if (fLinkAReadyToSend &&
        fQueueHasData (&qDataToLinkA))
    {
        ch = chQueueGetData (&qDataToLinkA);
        disable ();
        !!Send character to Link A
        fLinkAReadyToSend = FALSE;
        enable ();
    }
    if (fLinkBReadyToSend &&
        fQueueHasData (&qDataToLinkB))
    {
        ch = chQueueGetData (&qDataToLinkB);
        disable ();
        !!Send ch to Link B
        fLinkBReadyToSend = FALSE;
        enable ();
    }
} //end of while(TRUE)
} //end of of main(void)
```

Code (Part #2)

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

- **Bottom line:** What is really important is that the arriving data (on either communication node) is stored in a queue.
- This is done through interrupts.
- Everything else is secondary.

```
        if (fLinkAReadyToSend &&
            fQueueHasData (&qDataToLinkA))
        {
            ch = chQueueGetData (&qDataToLinkA);
            disable ();
            !!Send character to Link A
            fLinkAReadyToSend = FALSE;
            enable ();
        }
        if (fLinkBReadyToSend &&
            fQueueHasData (&qDataToLinkB))
        {
            ch = chQueueGetData (&qDataToLinkB);
            disable ();
            !!Send ch    to Link B
            fLinkBReadyToSend = FALSE;
            enable ();
        }
    } //end of while(TRUE)
} //end of of main(void)
```

Code (Part #3)

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

    // While there are chars from port A
    while (fQueueHasData(&qDataFromLinkA))
    {
        //Encrypt them and put them on
        //queue for port B
        chClear = chQueueGetData
            (&qDataFromLinkA);

        chCryptic = !! Do encryption
        vQueueAdd (&qDataToLinkB, chCryptic);
    }
}
```

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

    // While there are chars from port B
    while (fQueueHasData
        (&qDataFromLinkB))
    {
        //decrypt them and put them on
        //queue for port A
        chCryptic = chQueueGetData
            (&qDataFromLinkB);
        chClear = !! Do decryption
        vQueueAdd (&qDataToLinkA, chClear);
    }
}
```