7.1: Communication between tasks

- What other forms of communication does an RTOS usually offer besides global data protected by semaphores?
- Three message-based options are described in the text:
  - Queues
  - Mailboxes
  - Pipes
- Advantages, disadvantages of sending messages:
  - \+ Often easier than using semaphores and global data
  - – Creates new ways of inserting bugs into your system


```
/* RTOS queue function prototypes */
void AddToQueue (int iData);
void ReadFromQueue (int *p_iData);

void Task1 (void)
{
   ...
   if (!! problem arises)
      vLogError (ERROR_TYPE_X);

   !! other things that need to be done soon
   ...
}

void Task2 (void)
{
   ...
   if (!! problem arises)
      vLogError (ERROR_TYPE_Y);

   !! other things that need to be done soon
   ...
}
```

```
void vLogError (int iErrorType)
{
   AddToQueue (iErrorType);
}

static int cErrors;

void ErrorsTask (void)
{
   int iErrorType;
   while (FOREVER)
   {
      ReadFromQueue (&iErrorType);
      ++cErrors;
      !! Send cErrors, iErrorType out on network
   }
}
```

Simple queue example

Tasks 1 & 2 are high priority tasks. Sometimes they detect an error that needs to be reported over the network. A low priority task is assigned the job of network communication so that other tasks are not delayed. (From Figure 7.1)

Queue functions in example

- Two reentrant functions:
  - AddToQueue( )
    - Posts a message to a queue
  - ReadFromQueue( )
    - Gets message from a queue
- Review: what is significance of them being *reentrant*?
  - What is guaranteed?
  - What does it allow developer to do?

Queue functions: discussion

- Example is simplistic because it glosses over many important details.
- Consider these questions:
  - Which queue will be used?
  - Where is that queue located in memory?
  - How big is the queue?
  - When and how was the queue allocated?
  - What is C type of each message in queue?
  - What if queue is empty when code requests next entry?
  - What if queue is full when code tries to insert new message?

Why make queue functions part of RTOS, and not just application code?

Queue usage example: Simptris (Lab 8)

- You write code to place pieces in simplified version of Tetris
  - Appearance of each new piece is signaled by an interrupt
  - Your code must calculate how to move piece given current state
  - Single output port for movement commands; fixed communication delay
- Logical design approach:
  - Tasks decide how to move each piece
  - Separate task sends commands; blocks until communication channel clear
- Design questions:
  - How does information about new piece get from ISR to placement tasks?
  - How do commands get from placement task to communication task?

The pesky details

- Like semaphores, queues must be created and initialized before using
- The application code must specify which queue to use
  - As with semaphores, application may have several queues
- Tricky: we'd like RTOS queue functions that work for queues with different *sizes* and *types*. How can we do this?
  - RTOS records size of each queue, uses generic type for every entry
  - Application code determines size, allocates memory to be used
  - Each queue represented by a multilevel data structure
    - Part managed by RTOS, part managed by application code

The pesky details

- Let's revisit details not addressed in previous example:
  - Which queue will be used?
  - Where is that queue located in memory?
  - How big is the queue?
  - When and how was the queue allocated?
  - What is C type of each message in queue?
  - What if queue is empty when code requests next entry?
  - What if queue is full when code tries to insert new message?

The pesky details

- Which queue will be used?
  - Queue functions that create, pend, and post will refer to a specific queue via a unique variable in user code (similar to semaphores)
- Where is queue located? How big and how allocated?
  - User code must declare array for each queue; must be "big enough"
  - User code must inform RTOS about queue so it can be managed
- What is *type* of each queue entry?
  - For maximum flexibility: typically a generic pointer (**void ***)
    - Can be cast to point to anything user wishes
  - But then what code must allocate/deallocate objects pointed to?
    - As you might guess: not the RTOS. More on this in a moment…

The pesky details

- What happens on a read when queue is empty?
  - Most common: calling task is blocked (like pending on semaphore)
  - Many kernels offer two read function alternatives:
    - Read from queue, block if empty
    - Read from queue, return immediate error if empty
- What happens on write when the queue is full?
  - Most common: function returns error
    - User code must test for this return value. But what should it do if error returned?
  - Less common: block caller until space becomes available
    - This version must never be called from interrupt code
  - Neither approach an obvious winner
    - Only sure-fire solution: make sure queue is big enough!

Queue message type

- Across many applications using same RTOS, you may want to send integers, strings, floats, structs, etc.
- Solution: RTOS views all entries as the same generic type (void *)
  - Queue is an array of pointers that RTOS manages
  - Entries can be cast to anything (or point to anything) programmer wants
  - Provides consistency for RTOS, flexibility for application code
- Responsibilities of application code:
  - Correctly cast void pointers to actual types used in application
  - Manage any objects that void pointers actually point to

Two code examples

- The first is simple: the message content is a single integer. The message is passed "by value" – integer content is in pointer field.
- The second example is more complex: the message content is a short array. The message is passed "by reference" – pointer field contains address of array.


```
/* Figure 7.2 More realistic use of a queue  */

/* RTOS queue function prototypes */
OS_EVENT *OSQCreate(void **ppStart, BYTE bysize);
unsigned char OSQPost (OS_EVENT *pOse, void *pvMsg);
void *OSQPend (OS_EVENT *pOse, WORD wTimeout, BYTE *pByErr);
#define WAIT_FOREVER 0

/* Global handle for message queue */
static OS_EVENT *pOseQueue;

/* The data space for our queue (managed by RTOS) */
#define SIZEOF_QUEUE 25
void *apvQueue[SIZEOF_QUEUE];

void main(void)
{
   ...
   /* the queue gets initialized before tasks are started */
   pOseQueue = OSQCreate (apvQueue, SIZEOF_QUEUE);
   ...
   // Create Task1
   // Create Task2
   ...
}

void Task1(void)
{
   ...
   if (// problem arises)
      vLogError(ERROR_TYPE_X);
   // Other things that need to be done soon.
   ...
}
```

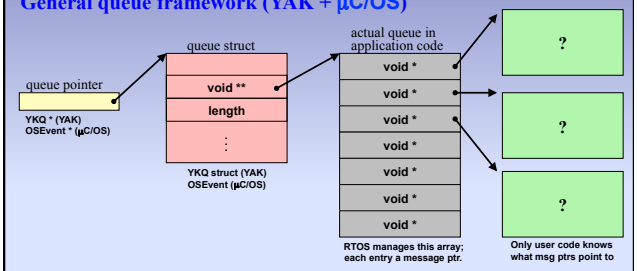
```
void Task2(void)
{
   ...
   if (// problem arises)
      vLogError(ERROR_TYPE_Y);
   // Other things that need to be done soon.
   ...
}

void VLogError(int iErrorType)
{
   BYTE byReturn;   /* Return code from writing to queue */
   /* Write to queue.  Cast error type as void pointer */
   byReturn = OSQPost (pOseQueue, (void *) iErrorType);
   if (byReturn != OS_NO_ERR)
      // Handle situation that arises when queue is full
}

static int cErrors;

void ErrorsTask(void)
{
   int iErrorType;
   BYTE byErr;

   while (FOREVER)
   {
      /* Cast value received back to int. No possible error,
         so ignore byErr. */
      iErrorType = (int) OSQPend(pOseQueue, WAIT_FOREVER, &byErr);
      ++cErrors;
      // Send cErrors and iErrorType out on network
   }
```

```
/* Figure 7.3 Passing pointers on queues */

/* Queue function prototypes */
OS_EVENT *OSQCreate (void **ppStart, BYTE bySize);
unsigned char OSQPost (OS_EVENT *pOse, void *pvMsg);
void *OSQPend (OS_EVENT *pOse, WORD wTimeout, BYTE *pByErr);
#define WAIT_FOREVER 0

static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask (void) {
  int *pTemperatures;
  while (TRUE) {
    // Wait until time to read next temperature
    /* get a new buffer for new temperature set */
    pTemperatures = (int *) malloc(2*sizeof *pTemperatures);
    pTemperatures[0] = // read in value from hardware
    pTemperatures[1] = // read in value from hardware
    /* add pointer to the new temperatures to the queue */
    OSQPost (pOseQueueTemp, (void *) pTemperatures);
  }
}

void vMainTask (void) {
  int *pTemperatures;
  BYTE byErr;

  while (TRUE) {
    pTemperatures = (int *) OSQPend(pOseQueueTemp, WAIT_FOREVER, &byErr);
    if (pTemperatures[0] != pTemperatures[1])
      // set off howling alarm
    free (pTemperatures);
  }
}
```

Not shown (but essential):
- allocation of queue
- call to OSQCreate

Dynamic memory allocation here is not ideal.
How else could this be done?

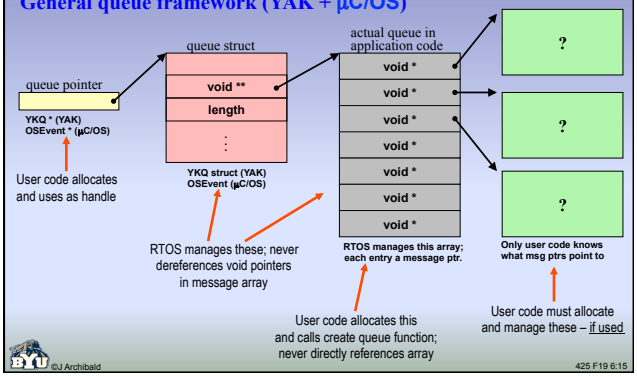
General queue framework (YAK + µC/OS)



Remember, **void *** is a wildcard type. Any pointer can be cast to a **void *** and back again without loss of information.

Why do it this way? What does RTOS deal with?
What does application code deal with?

General queue framework (YAK + µC/OS)

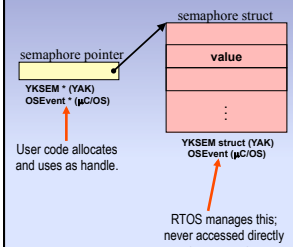


YAK: Division of responsibilities

- **The application code**
  - Defines the YKQ* "handle"
  - Allocates the actual queue (array of void *)
  - Initializes the handle with call to YKQCreate (passes queue address, size)
  - Calls YKQPend to get next entry in specified queue
  - Calls YKQPost to put next entry in specified queue
  - Allocates/frees whatever the queue entries point to if they are pointers
- **It does not**
  - Reference contents of YKQ struct
  - Know any management details of queue
  - Reference queue contents directly

- **The RTOS**
  - Has available pool of YKQ structs
  - Returns ptr to new (initialized) YKQ struct on each call to YKQCreate
  - Puts entry in the named queue for each call to YKQPost
  - Returns entry from named queue for each call to YKQPend, blocks caller if empty
- **It does not**
  - Define handle for queue
  - Allocate the queue (array of void *)
  - Know how to interpret entries in the queue

General semaphore framework (YAK + µC/OS)



YAK queue code (from Lab 6)

```
#include "clib.h"
#include "yakk.h"                  /* contains kernel definitions */
#include "lab6defs.h"              /* contains user's definitions */

#define TASK_STACK_SIZE  512       /* stack size in words */
#define MSGQSIZE         10

struct msg MsgArray[MSGARRAYSIZE]; /* buffers for message content */

int ATaskStk[TASK_STACK_SIZE];     /* a stack for each task */
int BTaskStk[TASK_STACK_SIZE];
int STaskStk[TASK_STACK_SIZE];

int GlobalFlag;

void *MsgQ[MSGQSIZE];              /* space for message queue */
YKQ *MsgQPtr;                      /* actual name of queue */
```

```
/* File: lab6defs.h */

#define MSGARRAYSIZE  20

struct msg
{
   int tick;
   int data;
};
```

```
void ATask(void)      /* processes data in msgs */
{
   struct msg *tmp;
   ...
   while (1)
   {
      /* get next msg */
      tmp = (struct msg *) YKQPend(MsgQPtr);
      ...
   }
}

void main(void)
{
   YKInitialize( );

   /* create queue, at least one user task, etc. */
   GlobalFlag = 0;
   MsgQPtr = YKQCreate(MsgQ, MSGQSIZE);
   YKNewTask(STask, (void *) &STaskStk[TASK_STACK_SIZE], 30);
   YKRun( );
}

/* tickhandler calls YKQPost(MsgQPtr, (void *) &(MsgArray[next])) */
```


Mailboxes and pipes

- Similar to queues
  - Tasks can use them to communicate with each other
  - Functions provided to create, write to, and read from
  - Both must be created before they are used
- Details of both are RTOS dependent

Typical mailboxes

- How do mailboxes differ from queues?
  - RTOS may restrict the number of entries
    - In some cases, a single entry per mailbox is allowed (µC/OS)
    - In some cases, a fixed number of total messages in system (across all mailboxes) cannot be exceeded at any point in time
  - RTOS may prioritize message order
    - Messages will come out in priority order, regardless of order in which they were inserted

Pipes

- How do pipes differ from queues?
  - Typically allow messages of varying length
    - In contrast, messages in queues and mailboxes have fixed length
  - Usually byte oriented
    - Writing task places some number of bytes into one end of the pipe
    - Reading task reads some number of bytes from other end of pipe
  - Writer and reader must agree how to parse variable-length messages

Which is best choice?

- For queues, mailboxes, and pipes the details vary, so developer must study RTOS documentation carefully
  - In YAK, we will implement queues
  - Mailboxes and pipes would not be hard to add
- Both functionality and performance are important
  - Vendor documentation usually gives information about memory requirements and runtime overhead
  - Observation: hard to get comparable information for Windows, Linux, etc.

Potential pitfalls

- Possible to use wrong queue, mailbox, or pipe
- Possible for reader and writer to interpret message content differently
  - Void pointers can be cast incorrectly or inconsistently; compiler won't catch it
- Possible to write code with multiple readers (tasks that empty the queue), but tricky to manage

Compiler will catch this error...

```
/* function that takes a ptr as parameter */
void vFunc (char *p_ch);

void main (void)
{
   int i;
   ...
   /* call function with an int */
   vFunc (i);
   ...
}
```

Compiler cannot help you here…

```
static OS_EVENT *pOseQueue;

void TaskA (void)
{
   int i;
   ...
   /* put an integer in the queue */
   OSQPost (pOseQueue, (void *) i);
   ...
}

void TaskB (void)
{
   char *p_ch;
   BYTE byErr;
   ...
   /* expect to get a char ptr */
   p_ch = (char *) OSQPend(pOseQueue,
      FOREVER, byErr);
   ...
}
```


Other pitfalls

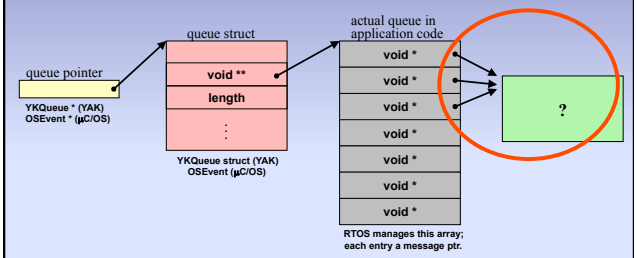
- Running out of space in the queue.
  - No good option if queue too small: lose data or block posting task
  - Designer should ensure that queue is big enough to handle the highest burst-rate of data
- Passing pointers can result in shared data problems that are more subtle than previous examples we've considered
  - Consider example on next slide

What's wrong with this code?

```
/* Queue function prototypes */
OS_EVENT *OSQCreate (void **ppStart,
      BYTE bySIZE);
unsigned char OSQPost (OS_EVENT *pOse,
      void *pvMsg);
void *OSQPend (OS_EVENT *pOse,
      WORD wTimeout, BYTE *pByErr);
#define WAIT_FOREVER 0
static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask (void)
{
   int iTemperatures[2];
   while (TRUE)
   {
      // Wait until time to read next temp;
      iTemperatures[0] = // read value from HW;
      iTemperatures[1] = // read value from HW;
      /* add to queue ptr to new temps */
      OSQPost (pOseQueueTemp,
            (void *) iTemperatures);
   }
}
```

```
void vMainTask (void)
{
   int *pTemperatures;
   BYTE byErr;

   while (TRUE)
   {
      pTemperatures = (int *)
            OSQPend(pOseQueueTemp,
            WAIT_FOREVER, &byErr);
      if (pTemperatures[0] != pTemperatures[1])
         // set off howling alarm;
   }
}
```


Queue details revisited



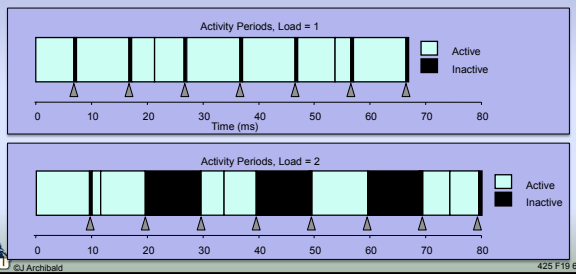
Problem: The void pointer for *every* queue entry points to *same* message buffer in user code!

7.2: Timer Functions

- Delaying a task is a typical RTOS service
  - As in YAK, parameter is usually number of system clock ticks
  - Usually not in seconds or other standard units of time
- Timer that triggers tick interrupts is often called the heartbeat timer
- Frequency of heartbeat timer is platform dependent
  - In many systems the frequency is programmable
  - Implementation details are usually encapsulated in system functions, simplifying task of application developer
- Use of timer like this is not unique to embedded systems

Example: Linux interval timer

- Consider program that spins in loop, reading system clock
  - Execution pattern can be detected: can you explain what we see below?

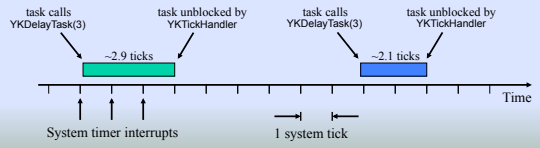


Timing accuracy

- How accurate can a delay mechanism be that is based on the heartbeat or interval timer?
- **Jitter:** variation and uncertainty in the actual interval from the time a task calls delay to when it actually runs again
- What bounds can we establish on the interval?
  - How long can it be?
  - How short can it be?
- Is jitter unavoidable?

Timing example

- Scenario:
  - Task calls YKDelayTask(3) each time through loop
  - Delay shown is from call to delay until task is unblocked
  - Clearly, the actual delay length can vary



Timing uncertainty

- Observation from previous slide:
  - We don't know when in tick interval that YKDelayTask is called
  - On 3rd clock tick, RTOS will change state of task to Ready
- If task calls YKDelayTask(*n*), what can we guarantee that RTOS will do?
  - Unblock task between *n* and *n*-1 tick intervals later
- When will task run?
  - This depends on the structure of the application code

Jitter

- How much delay from unblocking to running?
  - Best case: It runs immediately after it is unblocked
  - Worst case: It experiences arbitrary delay because ISRs and higher priority tasks are executing
  - Actual delay depends on interrupt behavior, relative task priorities, state of other tasks, etc.
- What can designer do if this timing is not accurate enough?
  - Start by reassigning task priorities, etc.
  - No surprise: the problem belongs to user code, not the RTOS

Increasing timing accuracy

- Options to consider:
  - Increase frequency of heartbeat timer
    - Downside: this increases total overhead of tick ISR and handler
    - You've seen this while testing code with short tick intervals
  - Use special hardware timers
    - Common in embedded systems
    - Most microcontrollers come with one or more built-in timers
    - How do they work?

Using a timeout timer

- First, set the timer to desired delay value
- Second, start timer
- When timer *expires* (counts down to zero), an interrupt is generated
  - Just one interrupt at the end; no other CPU overhead until then
  - You write ISR/handler for that interrupt that takes the actions you want
- This approach results in very precise timing
  - Intervals are essentially any desired number of *processor* clock cycles
  - Running hardware timer unaffected by processor utilization, interrupts, etc.

Timers

- What if desired interval (in processor cycles) exceeds range of counter?
  - Hardware often provides a programmable prescaler
  - If value set to *n*, counter decremented once for each *n* cycles
- What if you want more timers than hardware provides?
  - Can create multiple software timers, all based on a single hardware timer
  - Set hardware timer to expire on first deadline of any SW timer
  - ISR/handler triggers actions for expired SW timer, updates all SW timers, and sets HW timer to expire when next SW timer expires

Configuring timers

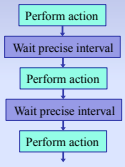
- An RTOS typically runs on multiple platforms
  - Part of job of porting RTOS is programming heartbeat timer since this is microprocessor dependent
  - Commercial RTOS will come set up for your processor
- If you use non-standard hardware timers, you may need to write:
  - Timer setup procedure
  - Timer ISR
- Often RTOS includes a board support package with
  - Drivers for common hardware components, and
  - Instructions and model code to help you create drivers for special HW

Other timing services

- Timeouts when blocking on semaphore, queue, or mailbox
  - My assessment: not easy to use
    - If pend call times out, what should code do? How to recover?
  - Alternative approach:
    - Use timeout as indicator of problem during design and testing; if it occurs, treat as design error and revise code
    - Example: if task can't wait any longer for a semaphore, then rewrite using messages in queue instead of semaphore

Timer callback functions

- A powerful and useful timing-related RTOS service
- Let's illustrate by considering the timing needs of one application: code controlling a radio
  - To turn radio off, just cut power
  - To turn radio on, multiple steps required:
    - Turn on power to basic radio hardware, then wait 12 ms
    - Set frequency of radio, then wait 3 ms
    - Turn on transmitter or receiver, and start using radio



- How could we do this with timing mechanisms already discussed?
  - Includes tasks, task delay functions, hardware timers, etc.

Timer callback functions, cont.

- Basic idea: specified function will be called after specified delay
- Call to timer callback function identifies:
  - Timer to use
  - Delay value to initialize timer with
  - Function to call, arguments to pass
- Very powerful and flexible; can simplify application code
  - Let's look at an example: source code for radio control

Figure 7.7a Using timer callback function

```
/* Message queue for radio task */
extern MSG_Q_ID queueRadio;

/* Timer for turning the radio on */
static WDOG_ID wdRadio;

static int iFrequency;   /* frequency to use */

void vSetFrequency (int i);
void vTurnOnTxorRx (int i);

void vRadioControlTask (void)
{
   #define MAX_MSG 20
   char a_chMsg[MAX_MSG + 1];
   enum
   {
      RADIO_OFF, RADIO_STARTING,
      RADIO_TX_ON, RADIO_RX_ON
   } eRadioState;     /* state of the radio */

   eRadioState = RADIO_OFF;

   /* create the radio timer */
   wdRadio = wdCreate( );
```

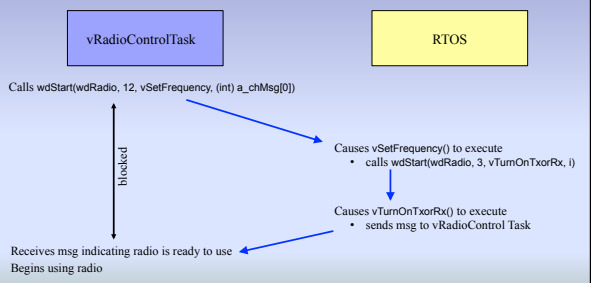
```
/* vRadioControlTask( ) continued */
while (TRUE)
{
   /* find out what to do next */
   msgQReceive (queueRadio, a_chMsg, MAX_MSG,
                WAIT_FOREVER);
   /* first char tells message type */
   switch (a_chMsg[0])
   {
      case 'T':
      case 'R':
         /* turn on transmitter or receiver */
         if (eRadioState == RADIO_OFF)
         {
            /* Turn on power to radio hardware
            eRadioState = RADIO_STARTING;
            /* get frequency from msg */
            iFrequency = * (int *) &(a_chMsg[1]);
            /* take next step in 12 ms */
            wdStart (wdRadio, 12, vSetFrequency,
                     (int) a_chMsg[0]);
         }
         else
            /* Handle error -- radio not off!
         break;
```

Figure 7.7b Using timer callback function, cont.

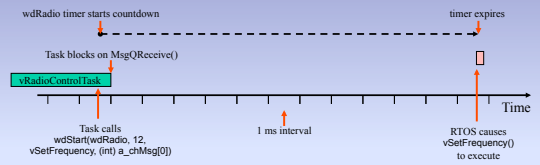
```
/* vRadioControlTask( ) continued */
case 'K':
   /* the radio is ready */
   eRadioState = RADIO_TX_ON;
   !! do whatever is desired with radio
   break;
case 'L':
   /* the radio is ready */
   eRadioState = RADIO_RX_ON;
   !! do whatever is desired with radio
   break;
case 'X':
   /* radio is to be turned off */
   if (eRadioState == RADIO_TX_ON ||
      eRadioState == RADIO_RX_ON)
   {
      !! Turn off power to radio
      eRadioState = RADIO_OFF;
   }
   else
      !! Handle error -- radio not on
   break;
...
default:
   !! Deal with the error of a bad message
   break;
}
}
}
```

```
void vSetFrequency (int i)
{
   !! Set radio frequency to iFrequency
   /* turn on the transmitter in 3 ms */
   wdStart (wdRadio, 3, vTurnOnTxorRx, i);
}

void vTurnOnTxorRx (int i)
{
   if (i == (int) 'T')
   {
      !! Turn on the transmitter
      /* tell the task that the radio is ready to go */
      msgQSend (queueRadio, "K", 1,
            WAIT_FOREVER, MSG_PRI_NORMAL);
   }
   else  /* i == (int) 'R' */
   {
      !! Turn on the receiver
      /* tell the task that the radio is ready to go */
      msgQSend (queueRadio, "L", 1,
            WAIT_FOREVER, MSG_PRI_NORMAL);
   }
}
```


Action and call sequence



Execution



- Implementation questions:
  - Does timer use a hardware timer or system tick?
    - Example above assumes hardware timer; VxWorks uses system tick
  - What code actually calls vSetFrequency()? (Whose stack will it run on?)
    - Book suggests two alternatives (*neither* of which is the calling task!):
      - May be called from ISR/handler for timer
      - May be called from special high priority task in RTOS
    - What are the tradeoffs? What jitter might be observed?

Discussion

- What are advantages of using timer callback functions rather than sequence of calls to delay task?
  - Increased timing accuracy
    - Tasks delayed using system tick may experience significant clock jitter
    - System tick may not have enough precision; hardware timer gives greater control
  - Simpler task structure
    - Code in vRadioControlTask blocks at just one place; easier to analyze task behavior
    - Would have multiple blocking points with calls to delay task

Discussion

- Are there disadvantages of using timer callback functions?
  - Compared with sequence of calls to delay task, which version of source code is easier to understand and modify?

7.3: Events

- A useful RTOS service you'll implement in lab 7
- An event is a Boolean flag that tasks can
  - create,
  - set,
  - reset, and
  - wait for (block on).
- Events generally handled in groups by RTOS
  - Task operations are on *sets of events*: dramatically increases power and flexibility of event construct
  - Pay particular attention to how events differ from semaphores

Events: standard features

- More than one task can be unblocked by same event
  - When event occurs, RTOS unblocks all tasks waiting for that event
  - Tasks then run in priority order (normal execution)
- Tasks can wait for any subset of events in event group
  - Wait until any *one* event occurs or until *all* events occur
- After event occurs and waiting tasks unblocked, event must be reset
  - Some kernels handle this, others leave it to task code
- Let's look at an example...

Event routines

These are used in example on next slide (from the AMX RTOS):

ajevcre(AMXID *p_amxidGroup, unsigned int uValueInit, char *p_chTag)

- Creates group of 16 events. First parameter points to location that will store the 16-bit event group. Initial value of all events in group is in uValueInit. char * is string name of object (unique to AMX).

ajevsig(AMXID amxidGroup, unsigned int uMask, unsigned int uValueNew)

- Sets or resets events in specified group. uMask specifies subset of events, and uValueNew specifies desired values of those events.

ajevwat(AMXID amxidGroup, unsigned int uMask, unsigned int uValue, int iMatch, long lTimeout)

- Causes task to wait for one or more events in group. uMask specifies subset, uValue specifies value desired, iMatch says to block until *all* events in mask occur or just *one*.

Figure 7.8: Using events in a cordless bar-code scanner

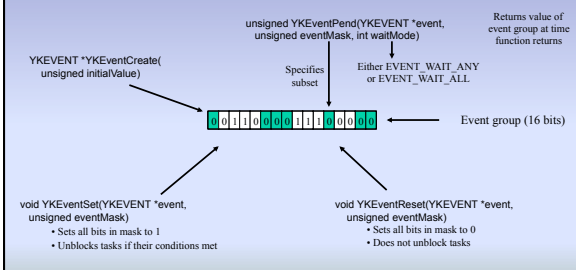
```
/* handle for the trigger group of events */
AMXID amxidTrigger;

/* constants for use in the group */
#define TRIGGER_MASK            0x0001
#define TRIGGER_SET             0x0001
#define TRIGGER_RESET           0x0000
#define KEY_MASK                0x0002
#define KEY_SET                 0x0002
#define KEY_RESET               0x0000

void main (void)
{
   ...
   /* create event group with trigger and keyboard
      events reset */
   ajevcre (&amxidTrigger, 0, "EVTR");
   ...
}

void interrupt vTriggerISR (void)
{
   /* trigger pulled.  Set event */
   ajevsig (amxidTrigger, TRIGGER_MASK,
            TRIGGER_SET);
}

void interrupt vKeyISR (void)
{
   /* key pressed.  Set event */
   ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
   // store value of key pressed
}
```

```
void vScanTask (void)
{
   ...
   while (TRUE)
   {
      /* wait for user to pull the trigger */
      ajevwat (amxidTrigger, TRIGGER_MASK, TRIGGER_SET,
               WAIT_FOR_ANY, WAIT_FOREVER);
      /* reset the trigger event */
      ajevsig (amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);

      // Turn on the scanner hardware, look for scan
      ...
      // When scan found, turn off scanner
   }
}

void vRadioTask (void)
{
   ...
   while (TRUE)
   {
      /* wait for trigger pull or key press */
      ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
               TRIGGER_SET | KEY_SET, WAIT_FOR_ANY, WAIT_FOREVER);
      /* reset key event.  (trigger will be reset by scantask) */
      ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);

      // turn on the radio
      ...
      // when data has been sent, turn off the radio
   }
}
```


Events: discussion

- What kind of bugs can occur in code using events?
  - Not resetting all events at appropriate point in code
    - Tricky since multiple tasks may be unblocked by event: which one resets?
    - Easier for application code if RTOS resets, but when should reset occur?
  - Waiting on wrong mask or wrong value
  - Resetting using wrong mask or wrong value
  - Understanding functionality: when waiting for **all** of three events, do they all have to be set at same time? (Are events "buffered"?)
- What is required to add support for events in YAK?
  - What new kernel data structures are required?
  - How complex are create(), pend(), and post() functions?

YAK event support



Comparing alternatives

- Semaphores
  - Usually faster and simpler than events and queues
  - Really just a one-bit message
  - A task can block on just one semaphore at a time
- Events
  - A little more complicated than semaphores, a little slower
  - A task can wait for any or all of several events at same time
  - Multiple tasks can be unblocked by a single event
- Queues (and mailboxes and pipes)
  - Message can consist of much more than one bit of information
  - A task can block on only one queue at a time
  - More system overhead, potential for bugs in application code

7.4: Memory management

- Embedded developers usually avoid using malloc() and free() because they are often slow, with unpredictable execution times
  - Why do these functions have high overhead?
- Alternative: simpler functions supported by RTOS
  - Typical functions allocate and free fixed size buffers
- Key questions:
  - Why would such functions be faster, more predictable than malloc() and free()?
  - Why are these functions part of the RTOS?

Memory pools

- Typical usage: application code sets up pools, each consisting of memory blocks or buffers of the same fixed size
- The RTOS manages pools, providing three key functions:
  - **Initialize pool**. Parameters include unique ID, base address, number of blocks, size of each block, etc.
  - **Obtain block**. Returns pointer to memory block that can be used. If none available, caller is blocked or null pointer is returned immediately
  - **Release block**. Caller passes pointer to memory block, RTOS returns that block to pool of available blocks

Example: memory management functions in *MultiTask!*

```
int init_mem_pool(
   unsigned int uPoolId,
   void *p_vMemory,
   unsigned int uBufSize,
   unsigned int uBufCount,
   unsigned int uPoolType);

void *getbuf(
   unsigned int uPoolId,
   unsigned int uTimeout);

void *reqbuf(
   unsigned int uPoolId);

void relbuf(
   unsigned int uPoolId,
   void *p_vBuffer);
```

uPoolType: used by task or interrupt code?

getbuf: blocks if no buffer available

reqbuf: returns NULL pointer right away if no buffer available

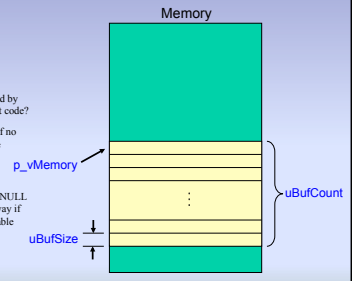


Figure 7.11: Using memory management functions

```
#define LINE_POOL          1
#define MAX_LINE_LENGTH   40
#define MAX_LINES         80

static char a_lines[MAX_LINES][MAX_LINE_LENGTH];

void main (void)
{
   ...
   init_mem_pool (LINE_POOL, a_lines, MAX_LINES,
         MAX_LINE_LENGTH, TASK_POOL);
   ...
}

void vPrintOutputTask (void)
{
   char *p_chLine;
   while (TRUE) {
      /* wait for a line to come in */
      p_chLine = rcvmsg(PRINT_MBOX, WAIT_FOREVER);

      !! Send line to printer

      /* free the buffer back to the pool */
      relbuf (LINE_POOL, p_chLine);
   }
}
```

```
void vPrintFormatTask (void)
{
   char *p_chLine;   /* pointer to current line */
   while (TRUE) {
      !! Wait for request for print job
      /* Format lines and send them to vPrintOutputTask */
      p_chLine = getbuf (LINE_POOL, WAIT_FOREVER);
      sprintf (p_chLine, "INVENTORY REPORT");
      sndmsg (PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
      p_chLine = getbuf (LINE_POOL, WAIT_FOREVER);
      sprintf (p_chLine, "Date: %02d/%02d/%02d", iMonth,
            iDay, iYear % 100);
      sndmsg (PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
      p_chLine = getbuf (LINE_POOL, WAIT_FOREVER);
      sprintf (p_chLine, "Time: %02d:%02d", iHour, iMinute);
      sndmsg (PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
      ...
   }
}
```

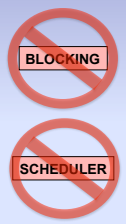
This code has a few problems. Can you spot any?

Discussion

- Why must application code set up pool?
  - RTOS does not know what memory to use, how big pool should be, or the size of blocks
- Common to use multiple pools, each with a different block size
  - What can go wrong as a result?
  - Compared with malloc() and free():
    - In what ways is this approach more efficient?
    - In what ways is this approach less efficient?

7.5: Rules for ISRs in an RTOS

1. Interrupt routine must not call any RTOS function that might block caller
   - Examples: pend on semaphore, queue, event, memory buffer, etc.
2. Interrupt code must not call any RTOS function that might cause a task switch *unless* RTOS knows that interrupt code, not a task, is running
   - This is called fair warning
   - Examples: post to semaphore, queue, mailbox, etc.



Critical to understand these in creating your RTOS

They aren't exactly new to us, but they deserve discussion

What's wrong with this code?

```
static int iTemperatures[2];

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

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

- What happens if interrupt happens while task is here?
  - "... the system would grind to a halt in a sort of one-armed deadly embrace."
- If we break rule 1, the attempt to block the interrupt routine will actually block the current (interrupted) task.

Alternative behavior

- In this scenario, some kernels would
  - assume (incorrectly) that current task is actually making call,
  - notice that the current task already has the semaphore, and then
  - let the ISR continue past the GetSemaphore() call
- Result:
  - Because RTOS functions are not used properly, the semaphore fails to protect the shared resource
  - Author: compared with the one-armed deadly embrace, this is "equally useless behavior"

Discussion

- Serious problems also arise if ISR interrupts a different task – not the one with semaphore – and "blocks"
  - RTOS will block the current task (the task that happened to be interrupted) until semaphore becomes available
  - ISR is also effectively "blocked" along with task
    - ISR context is saved on task stack
    - ISR/handler suspended at point of call to GetSemaphore()
    - Interrupts disabled at current and lower priority levels, hence ignored until release of semaphore
    - Dispatcher switches to different stack, causes another task to execute
    - When interrupted task is unblocked, execution will resume in ISR on stack

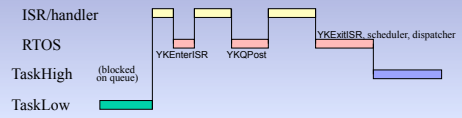
Is this code okay?

```
int iQueueTemp;              /* Fig. 7.13 */

void interrupt vReadTemperatures (void)
{
   int aTemp[2];     /* 16 bit values */
   int iError;

   aTemp[0] = !! read in value from HW
   aTemp[1] = !! read in value from HW
   sc_qpost (iQueueTemp, (char *)
      ((aTemp[0] << 16) | aTemp[1]), &iError);
}

void vMainTask (void)
{
   long int iTemps;   /* 32 bit value */
   int aTemp[2];
   int iError;

   while (TRUE)
   {
      iTemps = (long) sc_qpend (iQueueTemp,
         WAIT_FOREVER, sizeof(int), &iError);
      aTemp[0] = (int) (iTemps >> 16);
      aTemp[1] = (int) (iTemps & 0x0000ffff);
      if (aTemp[0] != aTemp[1])
         !! Set off howling alarm
   }
}
```

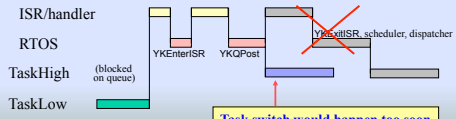
- What assumptions did programmer make?
- Is rule 1 violated?
  - What happens when the queue fills up?
- Is rule 2 violated?
  - What must be true about sc_qpost() to | avoid problems?

Violating rule 2

How YAK ISRs *should* work



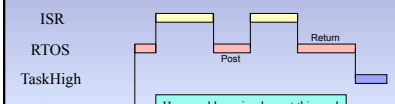
What would happen if RTOS didn't know YKQPost() was called by ISR



Task switch would happen too soon (on 8086, EOI would be missed!)

Methods of obeying rule 2

Method 1



How could you implement this, and what can go wrong?

- RTOS intercepts all interrupts, then calls appropriate ISR
- RTOS always knows that it is executing an ISR
- ISR returns to RTOS before returning to task

Method 2

What can go wrong here? What are tradeoffs with method 1?

- RTOS provides a routine that ISRs must call to inform the RTOS that an interrupt is running
- Near end, ISR calls RTOS routine that calls scheduler
- This is approach used in YAK

Methods of obeying rule 2

- Method 3
  - RTOS provides separate set of functions to be called exclusively by interrupt routines:
    - ISRSemPost(), ISRQPost(), etc.
  - Regular post functions can be called only from task code
    - SemPost(), QPost(), etc.
  - Scheduler called at end of task post routines, but not ISR post routines

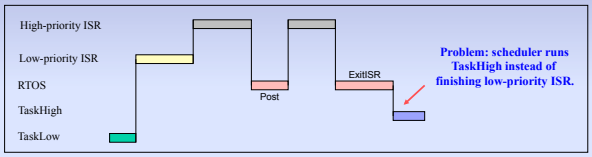
What could go wrong in this approach?
What are tradeoffs of three methods?

YAK implementation

- YAK uses method 2 to provide "fair warning"
  - What is purpose of YKEnterISR()?
  - What is purpose of YKExitISR()?
  - What test is required in every post function?
- What happens if
  - You forget to call YKEnterISR() or YKExitISR() in an ISR?
  - Interrupts are enabled before call to YKEnterISR(), or after call to YKExitISR()?
  - Post function doesn't correctly test for call from interrupt code?

Rule 2 and nested interrupts

- If a higher-priority interrupt can interrupt a lower-priority ISR, then another consideration comes into play.



- Solution:
  - ExitISR() needs to know if it is returning to a lower-priority ISR or to task code
- How is this addressed in YAK and what is runtime overhead?

Problem 7.1: What's wrong with this code?

Assumptions:
- Messages are void *
- sndmsg() puts void * in queue
- rcvmsg() returns void *

```
void vLookForInputTask (void)
{
  while (TRUE)
  {
    ...
    if (!! a key has been pressed on the keyboard)
      vGetKey( );
    ...
  }
}

void vGetKey (void)
{
  char ch;

  ch = !! get key from keyboard
  /* now send key to task that handles commands */
  sndmsg (KEY_MBOX, &ch, PRIORITY_NORMAL);
}

void vHandleKeyCommandsTask (void)
{
  char *p_chLine;
  char ch;

  while (TRUE)
  {   /* wait for key to be received */
    p_chLine = rcvmsg (KEY_MBOX, WAIT_FOREVER);
    ch = *p_chLine;
    !! do what is needed with ch
  }
}
```


Problem 7.2: Can you rewrite this code using semaphores in place of events?

```
/* handle for the trigger group of events */
AMXID amxidTrigger;

/* constants for use in the group */
#define TRIGGER_MASK          0x0001
#define TRIGGER_SET           0x0001
#define TRIGGER_RESET         0x0000

void main (void)
{
  ...
  /* create event group with all events reset */
  ajevcre (&amxidTrigger, 0, "EVTR");
  ...
}

void interrupt vTriggerISR (void)
{
  /* trigger pulled.  Set event */
  ajevsig (amxidTrigger, TRIGGER_MASK,
           TRIGGER_SET);
}
```

```
void vScanTask (void)
{
  ...
  while (TRUE)
  {
    /* wait for user to pull the trigger */
    ajevwat (amxidTrigger, TRIGGER_MASK,
             TRIGGER_SET,WAIT_FOR_ANY,
             WAIT_FOREVER);

    /* reset the trigger event */
    ajevsig (amxidTrigger, TRIGGER_MASK,
             TRIGGER_RESET);

    !! Turn on the scanner hardware, look for scan
    ...
    !! When scan found, turn off scanner
  }
}
```


Problem 7.2b: Rewrite with semaphores?

```
/* a handle for the trigger group of events */
AMXID amxidTrigger;

/* constants for use in the group */
#define TRIGGER_MASK          0x0001
#define TRIGGER_SET           0x0001
#define TRIGGER_RESET         0x0000
#define KEY_MASK              0x0002
#define KEY_SET               0x0002
#define KEY_RESET             0x0000

void main (void)
{
  ...
  /* create event group with trigger and keyboard
     events reset */
  ajevcre (&amxidTrigger, 0, "EVTR");
  ...
}

void interrupt vTriggerISR (void)
{
  /* trigger pulled.  Set event */
  ajevsig (amxidTrigger, TRIGGER_MASK,
           TRIGGER_SET);
}

void interrupt vKeyISR (void)
{
  /* key pressed.  Set event */
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !! store value of key pressed
}
```

```
void vScanTask (void)
{
  ...
  while (TRUE)
  {
    /* wait for user to pull the trigger */
    ajevwat (amxidTrigger, TRIGGER_MASK, TRIGGER_SET,
             WAIT_FOR_ANY, WAIT_FOREVER);
    /* reset the trigger event */
    ajevsig (amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);

    !! Turn on the scanner hardware, look for scan
    ...
    !! When scan found, turn off scanner
  }
}

void vRadioTask (void)
{
  ...
  while (TRUE)
  {
    /* wait for trigger pull or key press */
    ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
             TRIGGER_SET | KEY_SET, WAIT_FOR_ANY,
             WAIT_FOREVER);
    /* reset key event.  (trigger will be reset by scantask) */
    ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);

    !! turn on the radio
    ...
    !! when data has been sent, turn off the radio
  }
}
```

Problem 7.2c: Rewrite with semaphores?

```
/* a handle for the trigger group of events */
AMXID amxidTrigger;

/* constants for use in the group */
#define TRIGGER_MASK            0x0001
#define TRIGGER_SET             0x0001
#define TRIGGER_RESET           0x0000
#define KEY_MASK                0x0002
#define KEY_SET                 0x0002
#define KEY_RESET               0x0000

void main (void)
{
  ...
  /* create event group with trigger and keyboard
     events reset */
  ajevcre (&amxidTrigger, 0, "EVTR");
} ...

void interrupt vTriggerISR (void)
{
  /* trigger pulled.  Set event */
  ajevsig (amxidTrigger, TRIGGER_MASK,
           TRIGGER_SET);
}

void interrupt vKeyISR (void)
{
  /* key pressed.  Set event */
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !! store value of key pressed
}
```

```
void vScanTask (void)
{
  ...
  while (TRUE)
  {
    /* wait for user to pull the trigger */
    ajevwat (amxidTrigger, TRIGGER_MASK, TRIGGER_SET,
             WAIT_FOR_ANY, WAIT_FOREVER);
    /* reset the trigger event */
    ajevsig (amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);

    !! Turn on the scanner hardware, look for scan
    ...
    !! When scan found, turn off scanner
  }
}

void vRadioTask (void)
{
  ...
  while (TRUE)
  {
    /* wait for trigger pull or key press */
    ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
             TRIGGER_SET | KEY_SET, WAIT_FOR_ALL,
             WAIT_FOREVER);
    /* reset key event.  (trigger will be reset by scantask) */
    ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);

    !! turn on the radio
    ...
    !! when data has been sent, turn off the radio
  }
}
```


Problem 7.3: What's wrong with this code?

```
void vGetCharactersTask (void)
{
  while (FOREVER)
  {
    if (!! have urgent command char)
      OSQPost (URGENT_QUEUE, !!next urgent cmd char);
    if (!! have regular command char)
      OSQPost (REGULAR_QUEUE, !!next regular cmd char);
    ...
  }
}

void vUseCharactersTask (void)
{
  char chUrgent;
  char chNormal;
  while (FOREVER)
  {
    chUrgent = OSQPend (URGENT_QUEUE, WAIT_FOREVER);
    !! handle chUrgent

    chNormal = OSQPend (REGULAR_QUEUE, WAIT_FOREVER);
    !! handle chNormal
  }
}
```


Problem 7.4: Does this change fix problem in previous slide?

```
void vGetCharactersTask (void)
{
  while (FOREVER)
  {
    if (!! have urgent command char)
      OSQPost (URGENT_QUEUE, !!next urgent cmd char);
    if (!! have regular command char)
      OSQPost (REGULAR_QUEUE, !!next regular cmd char);
    ...
  }
}

void vUseCharactersTask (void)
{
  char chUrgent;
  char chNormal;
  while (FOREVER)
  {
    chUrgent = OSQPend (URGENT_QUEUE, WAIT_100_MSEC);
    !! handle chUrgent

    chNormal = OSQPend (REGULAR_QUEUE, WAIT_100_MSEC);
    !! handle chNormal
  }
}
```


Problem 7.5

In Section 7.4 it was suggested that one reasonable design for memory management is to allocate three or four memory buffer pools, each with a different size of buffer.

What drawbacks can you see to this design compared to using malloc() and free()?

Problem 7.6: What is wrong with this code?

```
void task1 (void)
{
  BUFFER *p_bufferA, *p_bufferA1;
  ...
  p_bufferA  = GetBuffer( );
  p_bufferA1 = GetBuffer( );

  !! put useful data into p_bufferA
  SendMsg(task2, p_bufferA);
  !! copy data from p_bufferA into p_bufferA1
  ...
  FreeBuffer(p_bufferA1);
}

void task2 (void)
{
  BUFFER *p_bufferB;
  ...
  p_bufferB = GetMsg( );
  !! use the data in p_bufferB

  FreeBuffer(p_bufferB);
  ...
}
```


Problem 7.7: Does this change fix problem in previous slide?

Can it be fixed using semaphores?

```
void task1 (void)
{
  BUFFER *p_bufferA, *p_bufferA1;
  ...
  GetSemaphore(SEM_OUR_MEMORY);
  p_bufferA  = GetBuffer ( );
  p_bufferA1 = GetBuffer ( );
  GiveSemaphore(SEM_OUR_MEMORY);

  !! put useful data into p_bufferA
  SendMsg (task2, p_bufferA);
  !! copy data from p_bufferA into p_bufferA1
  ...
  FreeBuffer (p_bufferA1);
}

void task2 (void)
{
  BUFFER *p_bufferB;
  ...
  p_bufferB = GetMsg ( );
  !! use the data in p_bufferB

  GetSemaphore(SEM_OUR_MEMORY);
  FreeBuffer (p_bufferB);
  GiveSemaphore(SEM_OUR_MEMORY);
  ...
}
```

Problem 7.8

The text outlines three different plans by which an RTOS finds out that an interrupt routine is executing. Compare these three plans. Which is likely to have the best interrupt response time, and which will be the easiest to create user code for? Are there differences in memory requirements?

Plan 1: RTOS intercepts all interrupts, then calls appropriate ISR for each. Control returns to RTOS at end of ISR.

Plan 2: RTOS provides function that must be called by each ISR at beginning, and another to be called at the end.

Plan 3: RTOS provides separate functions for ISRs and tasks.

Problem 7.9

On some RTOSs you can write two kinds of interrupt routines: *conforming routines*, which tell the RTOS when they enter and exit, and *nonconforming routines*, which do not.

What advantage does a nonconforming routine have?

What disadvantages?