# Events and memory management

Reference: Simon Chapter 7

# What is an event?

An event is:

- a boolean flag that tasks can be set or reset.

and

- a boolean flag that other tasks can wait for.

# Cordless bar-code scanner example

- When the user pulls the trigger, the task that starts scanning and recognizes the bar-code must start.

- Events make this easy.

- The interrupt routine that runs when the user pulls the trigger sets an event for which the scanning task is waiting.

# Some standard features of events

- More than one task can be blocked waiting for the same event.
- The RTOS will unblock all of them (and run them in priority order) when the event occurs.
- RTOSs typically form groups of events, and tasks can wait for any subset of events within the group.
  - Example: Playstation 3 must turn on when you press the console button or when you press the controller.

# What to do after an event?

- What to do with the tasks that have been unblocked?
- What will happen to the event boolean flag?
- ... Both Depends on the RTOS choice.
  - Some RTOSs reset events automatically.
  - Other RTOs require that your task software do this.

# Implementation example of events

- The next slides will describe how events can be implemented on the AMX RTOS.
- Their website is: http://www.kadak.com/rtos/rtos.htm

```
// 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 an event group with
   //the trigger and keyboard events reset
   ajevcre (&amxidTrigger, 0, "EVTR");
   (...)
}

void interrupt vTriggerlSR (void)
{
  //The user pulled the trigger.
  //Set the event.
  ajevsig (amxidTrigger, TRIGGER_MASK,
           TRIGGER_SET);
}

void interrupt vKeylSR (void)
{
  //The user pressed a key. Set the event.
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !!Figure out which key the user pressed
  !!store that value
}
```

```
void vScanTask (void)
{
  (...)
  while (TRUE)
  {
    // Wait for the 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 and look for a scan.
    (...)
    !!When the scan has been found,
    !!turn off the scanner.
  }
 }

void vRadioTask (void)
{
  (...)
   while (TRUE)
   {
   //Wait for the user to pull the trigger or
   //press a key.
   ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
            TRIGGER_SET | KEY_SET, WAIT_FOR_ANY,
            WAIT_FOREVER);

   // Reset the key event. The trigger event will
   //be reset by the ScanTask/
   ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);
   !!Turn on the radio.
   (...)
   !!When data has been sent, turn off the radio.
   }
}
```

```
// 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 an event group with
   //the trigger and keyboard events reset
   ajevcre (&amxidTrigger, 0, "EVTR");
   (...)
}

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

void interrupt vKeyISR (void)
{
  //The user pressed a key. Set the event.
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !!Figure out which key the user pressed
  !!store that value
}
```

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

The ajevcre function creates a group of 16 events, the handle for which is written into the location pointed to by p_amxidGroup (`&amxidTrigger`).

The initial values of those events, set and reset, are contained in the uValueInit parameter (which is `0`).

AMX assigns the group a four-character name pointed to by p_chTag (“`EVTR`”) which allows a task to find system objects by name.

```
// 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 an event group with
   //the trigger and keyboard events reset
   ajevcre (&amxidTrigger, 0, "EVTR");
   (...)
}

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

void interrupt vKeyISR (void)
{
  //The user pressed a key. Set the event.
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !!Figure out which key the user pressed
  !!store that value
}
```

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

The ajevsig function sets and resets the events in the group indicated by amxidGroup (`amxidTrigger`).

The uMask parameter (`TRIGGER_MASK`) indicates which events should be set or reset.

The uValueNew parameter (`TRIGGER_SET`) indicates the new values that the events should have.

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

The ajevwat function causes the task to wait for one or more events within the group indicated by amxidGroup (amxidTrigger).

The uMask (TRIGGER_MASK) parameter indicates which events the task wants to wait for.

```
void vScanTask (void)
{
  (...)
  while (TRUE)
  {
    // Wait for the 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 and look for a scan.
    (...)
    !!When the scan has been found,
    !!turn off the scanner.
  }
 }

void vRadioTask (void)
{
  (...)
   while (TRUE)
   {
   //Wait for the user to pull the trigger or
   //press a key.
   ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
            TRIGGER_SET | KEY_SET, WAIT_FOR_ANY,
            WAIT_FOREVER);

   // Reset the key event. The trigger event will
   //be reset by the ScanTask/
   ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);
   !!Turn on the radio.
   (...)
   !!When data has been sent, turn off the radio.
   }
}
```

The uValue (TRIGGER_SET) indicates whether the task wishes to wait for those events to be set or reset.

The iMatch (WAIT_FOR_ANY) parameter indicates whether the task wishes to unblock when all of the events specified by uMask (TRIGGER_MASK) have reached the values specified by uValue (TRIGGER_SET) or when any one of the events has reached the specified value.

```
void vScanTask (void)
{
  (...)
  while (TRUE)
  {
    // Wait for the 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 and look for a scan.
    (...)
    !!When the scan has been found,
    !!turn off the scanner.
  }
 }

void vRadioTask (void)
{
  (...)
   while (TRUE)
   {
   //Wait for the user to pull the trigger or
   //press a key.
   ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
            TRIGGER_SET | KEY_SET, WAIT_FOR_ANY,
            WAIT_FOREVER);

   // Reset the key event. The trigger event will
   //be reset by the ScanTask/
   ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);
   !!Turn on the radio.
   (...)
   !!When data has been sent, turn off the radio.
   }
}
```

```
// Handle for the trigger group of events.
AMXID amxidTrigger;
//Constants for use in the group.
#define TRIGGER_MASK 0x0001
#define TRIGGER_SET 0x0001
#
#
#
#
v
{
   //the trigger and keyboard events reset
   ajevcre (&amxidTrigger, 0, "EVTR");
   (...)
}

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

void interrupt vKeyISR (void)
{
  //The user pressed a key. Set the event.
  ajevsig (amxidTrigger, KEY_MASK, KEY_SET);
  !!Figure out which key the user pressed
  !!store that value
}
```

The ITimeout parameter (`WAIT_FOREVER`) indicates how long the task is willing to wait for the events.

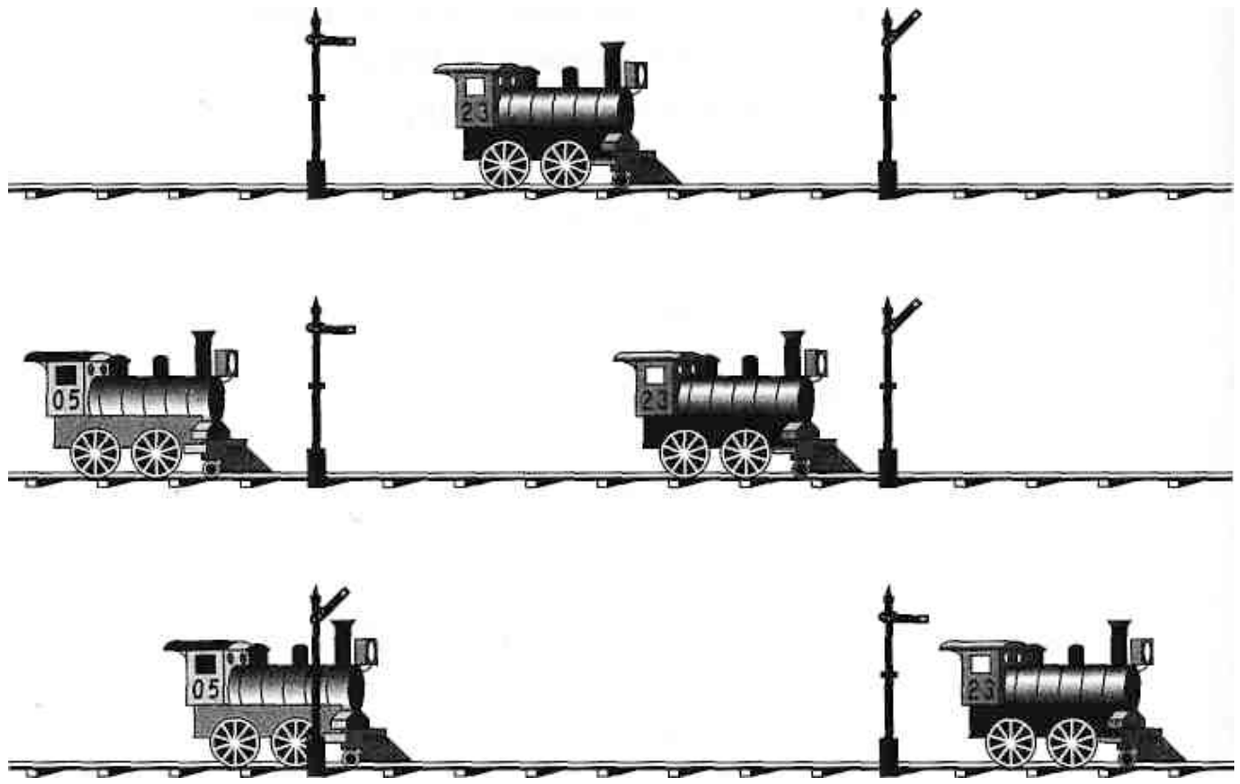
```
void vScanTask (void)
{
  (...)
  while (TRUE)
  {
    // Wait for the user to pull the trigger.
    ajevwat (amxidTrigger, TRIGGER_MASK, TRIGGER_SET,
             WAIT_FOR_ANY, WAIT_FOREVER);
    // Reset the trigger event.
    ajevsig (amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);
    !!Turn on the scanner hardware and look for a scan.
    (...)
    !!When the scan has been found,
    !!turn off the scanner.
  }
 }

void vRadioTask (void)
{
  (...)
   while (TRUE)
   {
   //Wait for the user to pull the trigger or
   //press a key.
   ajevwat (amxidTrigger, TRIGGER_MASK | KEY_MASK,
            TRIGGER_SET | KEY_SET, WAIT_FOR_ANY,
            WAIT_FOREVER);

   // Reset the key event. The trigger event will
   //be reset by the ScanTask/
   ajevsig (amxidTrigger, KEY_MASK, KEY_RESET);
   !!Turn on the radio.
   (...)
   !!When data has been sent, turn off the radio.
   }
}
```

# Comparing the methods for intertask communication

# Semaphores

- Semaphores are usually the fastest and simplest methods.

- However, they pass just a 1-bit message saying that it has been released.



# Events

- Events are a bit more complicated than semaphores.
- They take up slightly more microprocessor time than semaphores.
- The advantage of events over semaphores is that a task can wait for any one of several events at the same time.
- On a semaphore, you must wait for a single semaphore to be released.
- RTOS make it more convenient to use queues than semaphores.

# Queues, mailboxes and pipes

Queues, mailboxes and pipes allow you to send information from one task to another.

Even though the task can wait on only one queue (or mailbox or pipe) at a time, the fact that you can send data through a queue makes it even **more flexible than events**.

**Drawbacks**:

1. Putting messages into and taking messages out of queues is more microprocessor-intensive

2. Queues make it easier to insert bugs into your code.

# Memory Management


Nuno Alves (nalves@wne.edu), College of Engineering

# Memory management

- Most RTOSs have some kind of memory management subsystem.

- Two standard memory allocation C functions are malloc and free: real-time systems often avoid these two functions because they are slow their execution times are unpredictable.

- Most RTOSs offer fast and predictable functions that allocate and deallocate memory.

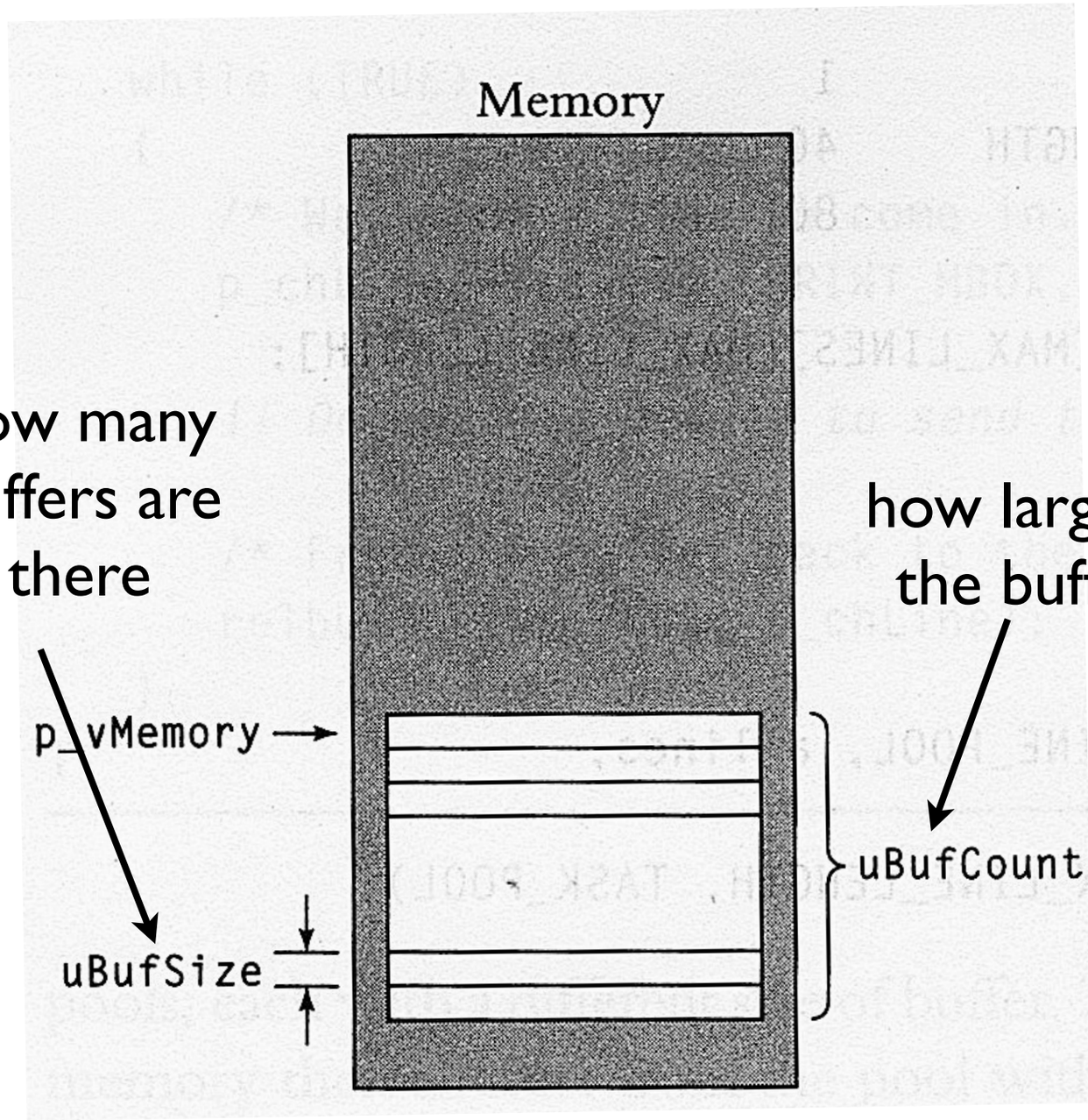
# MultiTask! RTOS

- The MultiTask! RTOS is a fairly typical RTOS.
- It set up pools, where each contains some number of memory buffers.
- In any given pool, all of the buffers are the same size.
- The **reqbuf** and **getbuf** functions allocate a memory buffer from a pool. The **relbuf** function frees a memory buffer.
- **reqbuf** & **getbuf** both returns a pointer to the allocated buffer.
- If no memory buffers are available, **getbuf** will block the task that calls it, whereas **reqbuf** will return a NULL pointer.

# Where is the memory?

- The MultiTask! system is also typical of many RTOSs in that it does not know where the memory on your system is.
- Most embedded systems get control of a machine first.
- When it starts, the RTOS has no way of knowing what memory is free and what memory your application is already using.
- MultiTask! will manage a pool of memory buffers for you, but you must tell it where the memory is.
- **init_mem_pool** function: allocates the pool of memory buffers.

# MultiTask! memory allocation




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

- The uPoolID parameter is that what will be used to call **reqbuf**,**getbuf** and **relbuf**
- **p_vMemory** points to the block of memory used as a pool.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- This code is the printing subsystem for a tank monitoring system, that reports the amount of water of each tank.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- I have data and I need to accurately print it.

- We need to format the report quickly so that the data in the report is consistent.

- Problem... I use a slow thermal printer that prints only a few lines each second.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- A higher-priority task formats the report, and a lower-priority task feeds the lines out to the printer one at a time
- A pool of buffers stores the formatted lines waiting to be printed.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- The code always allocates a full 40-character buffer, even if a given line has very little on it, obviously a waste of memory.
- A pool of buffers stores the formatted lines waiting to be printed.
- This waste of memory is the price you pay for the improved speed with fixed-size buffers.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- Common compromise: allocate three or four memory buffer pools, each with a different size of buffer.

- Then you retains the high-speed memory routines but efficiently uses memory.

```
#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_P00L, a_lines,
                  MAX_LINES, MAX_LINE_LENGTH,
                  TASK_P00L);
   (...)
}

void vPrintFormatTask (void)
{
   //Pointer to current line
   char *p_chLine;

   (...)

   //Format lines and send them to the
   //vPrintOutputTask
   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "INVENTORY REPORT");
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Date: %02/%02/%02",
            iMonth, iDay, iYear % 100);
   sndmsg (PRINT_MB0X. p_chLine, PRI0RITY_N0RMAL);

   p_chLine = getbuf (LINE_P00L, WAIT_FOREVER);
   sprintf (p_chLine, "Time: %02:%02",
            iHour,iMinute);
   sndmsg (PRINT_MB0X, p_chLine, PRI0RITY_N0RMAL);

   (...)
}
```

```
void vPrintOutputTask (void)
{
  char *p_chLine;

  while  (TRUE)
  {
    //Wait for a line to come in.
    p_chLine = rcvmsg (PRINT_MBOX, WAIT_FOREVER);
    !!Do what is needed
    !!Send the line to the printer
    //Free the buffer back to the pool
    relbuf (LINE_P00L, p_chLine);
  }

}
```

- Common compromise: allocate three or four memory buffer pools, each with a different size of buffer.

- Tasks that need little memory allocate them from the pool with the smallest buffers

- Tasks that need larger blocks of memory allocate from the pools with the larger buffers.

# Interrupt routines in an RTOS environment

# Interrupts

- It will be easier to understand the next couple of slides if you think of interrupts as the timer functions we discussed on our last class.

# Interrupts are NOT tasks

This means:

- There are no READY, BLOCKED or RUNNING states for interrupts.
- In ROTS, when an interrupt is triggered, it must run right now.
- If it can't get needed resource, the interrupt will stall there.

You also want the ISR to run until the completion... unless another higher priority interrupt shows up.

# Why do you care about interrupts running to completion?

- If not the interrupt flag will not be reset. (So... who cares?)
- The interrupt that was not properly terminated may not complete for a long time. (This is bad)
- This will ensure all lower-priority interrupts will not be allowed. (This is very bad)

# Rule #1

Interrupt routines in most RTOS environments must follow two rules that do not apply to task code.

**Rule #1 -** An interrupt routine must not call any RTOS function that might block the caller.

- So... interrupt routines must not get semaphores, read from queues or mailboxes that might be empty, wait for events, and so on.

- Most interrupt routines must run to completion in order to reset the hardware. This way they are ready for the next interrupt.

# Why not?

- An interrupt routine must not call any RTOS function that might block the caller. Why?

- If an interrupt routine calls an RTOS function and gets blocked, the task that was running when the interrupt occurred will be blocked, even if that task is the highest-priority task.

# Nuclear reactor temperature example

```
static int iTemperatures[2];
void interrupt vReadTemperatures (void)
{
  //This next line is **N0T ALLOWED**
  GetSemaphore (SEMAPHORE_TEMPERATURE);
  iTemperatures[0] = !!read in value from hardware;
  iTemperatures[1] = !!read in value from hardware;
  GiveSemaphore (SEMAPHORE_TEMPERATURE);
}

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

- Rule #1: No Blocking
- The task code and the interrupt routine share the temperature data with a semaphore.
- This code will not work. Why?

# Nuclear reactor temperature example

```
static int iTemperatures[2];
void interrupt vReadTemperatures (void)
{
  //This next line is **N0T ALLOWED**
  GetSemaphore (SEMAPHORE_TEMPERATURE);
  iTemperatures[0] = !!read in value from hardware;
  iTemperatures[1] = !!read in value from hardware;
  GiveSemaphore (SEMAPHORE_TEMPERATURE);
}

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

- If the interrupt routine happened to interrupt vTaskTestTemperatures while it had the semaphore, then when the interrupt routine called GetSemaphore, the RTOS would notice that the semaphore was already taken and block.

- This will stop both the interrupt routine and vTaskTestTemperatures

# Functions that never block

- Some RTOSs contain various functions that never block.
- For example, functions that returns the status of a semaphore.

```
int iQueueTemp; // Queue for temperatures.
void interrupt vReadTemperatures  (void) {
  int aTemperatures[2]; //16-bit temperatures.
  int iError;
  //Get a new set of temperatures.
  aTemperatures[0] = !!read in value from hardware;
  aTemperatures[1] = !!read in value from hardware;
  //Add the temperatures to a queue.

  sc_qpost (iQueueTemp,
            (char *) ((aTemperatures[0] « 16) |
            aTemperatures[1]), &iError);
}

void vMainTask (void) {
  long int ITemps;  //32 bits; same size as a ptr.
  int aTemperatures[2];
  int iError;
  while (TRUE)
  {
    ITemps = (long) sc_qpend (iQueueTemp,
                              WAIT_FOREVER,
                              sizeof(int), &iError);
    aTemperatures[0] = (int) (ITemps » 16);
    aTemperatures[1] = (int) (ITemps & 0x0000ffff);
    if (aTemperatures[0] != aTemperatures[1])
    !!Set off howling alarm;
  }
}
```

# Functions that never block

- This code shows an interrupt routine using another nonblocking RTOS function.
- That code is legal because the sc_qpost function (from the VRTX RTOS) will never block.
- Note that this code would violate rule 1 if sc_qpost might block

```
int iQueueTemp; // Queue for temperatures.
void interrupt vReadTemperatures  (void) {
  int aTemperatures[2]; //16-bit temperatures.
  int iError;
  //Get a new set of temperatures.
  aTemperatures[0] = !!read in value from hardware;
  aTemperatures[1] = !!read in value from hardware;
  //Add the temperatures to a queue.

  sc_qpost (iQueueTemp,
            (char *) ((aTemperatures[0] « 16) |
            aTemperatures[1]), &iError);
}

void vMainTask (void) {
  long int ITemps;  //32 bits; same size as a ptr.
  int aTemperatures[2];
  int iError;
  while (TRUE)
  {
    ITemps = (long) sc_qpend (iQueueTemp,
                              WAIT_FOREVER,
                              sizeof(int), &iError);
    aTemperatures[0] = (int) (ITemps » 16);
    aTemperatures[1] = (int) (ITemps & 0x0000ffff);
    if (aTemperatures[0] != aTemperatures[1])
    !!Set off howling alarm;
  }
}
```

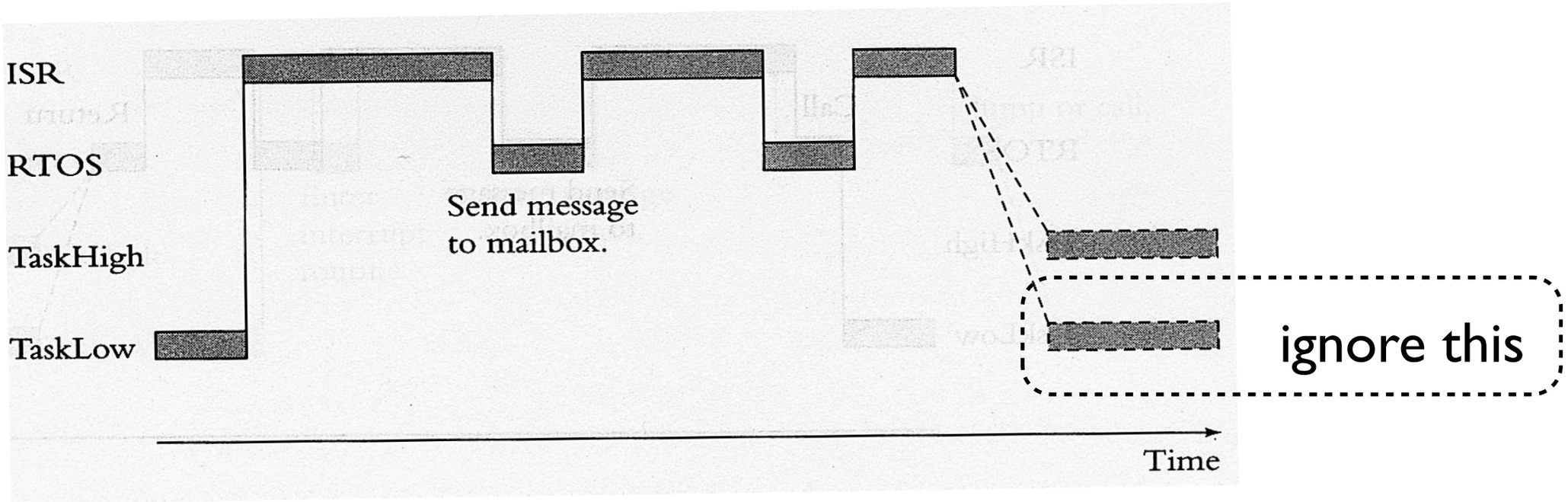
# Rule #2

**Rule #2 -** An interrupt routine may not call any RTOS function that might cause the RTOS to switch tasks unless the RTOS knows that an interrupt routine, and not a task, is executing.

- Interrupt routines may not write to mailboxes or queues on which tasks may be waiting, set events, release semaphores, and so on.

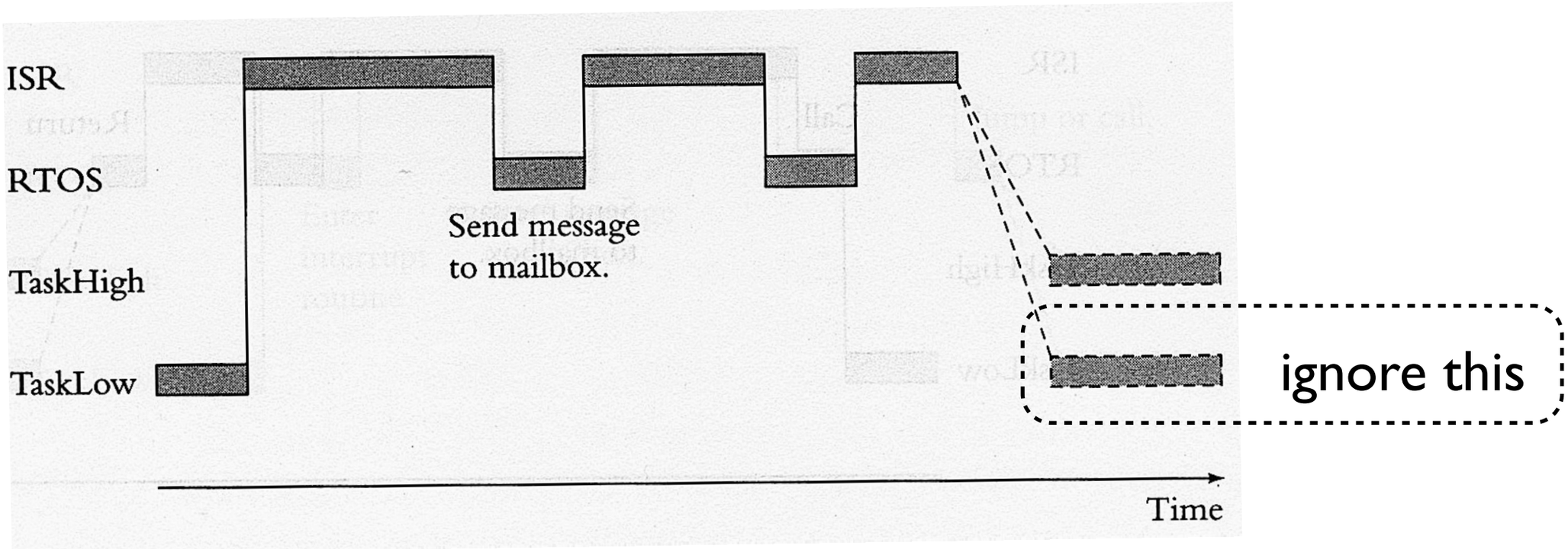
- If an interrupt routine breaks this rule, the RTOS might switch control away from the interrupt routine to run another task, and the interrupt routine may not complete for a long time, blocking at least all lower-priority interrupts and possibly all interrupts.

# Rule 2: No RTOS calls without fair warning




- This figure shows an ideal behavior: how a microprocessor attention gets shifted from one part of the code to another over time.
- The interrupt routine interrupts the lower-priority task, and calls the RTOS to write a message to a mailbox.
- In this example, the mailbox message is: execute a higher level task.

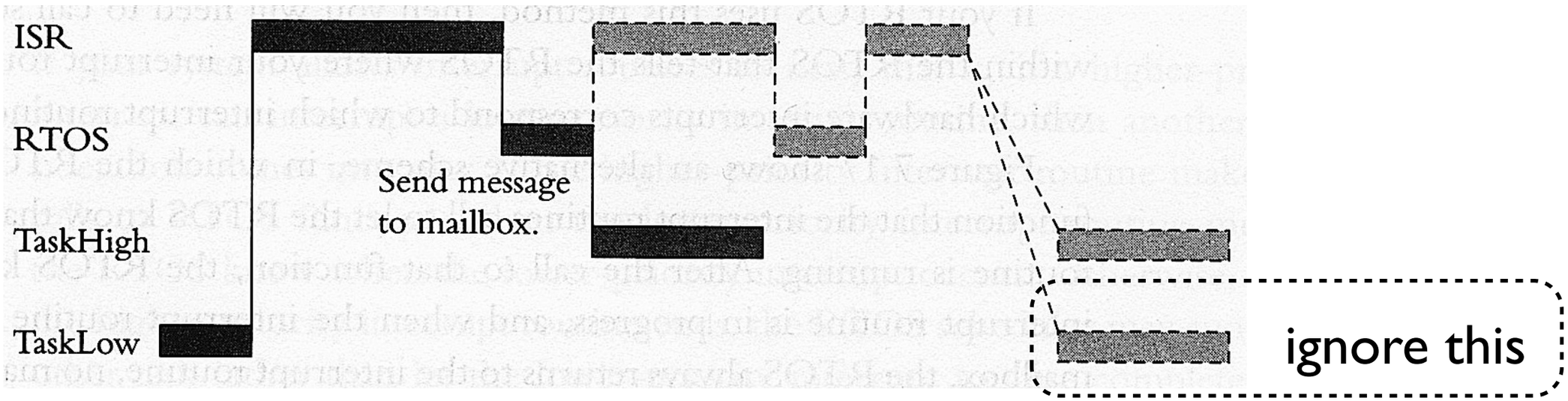
# How interrupts should work



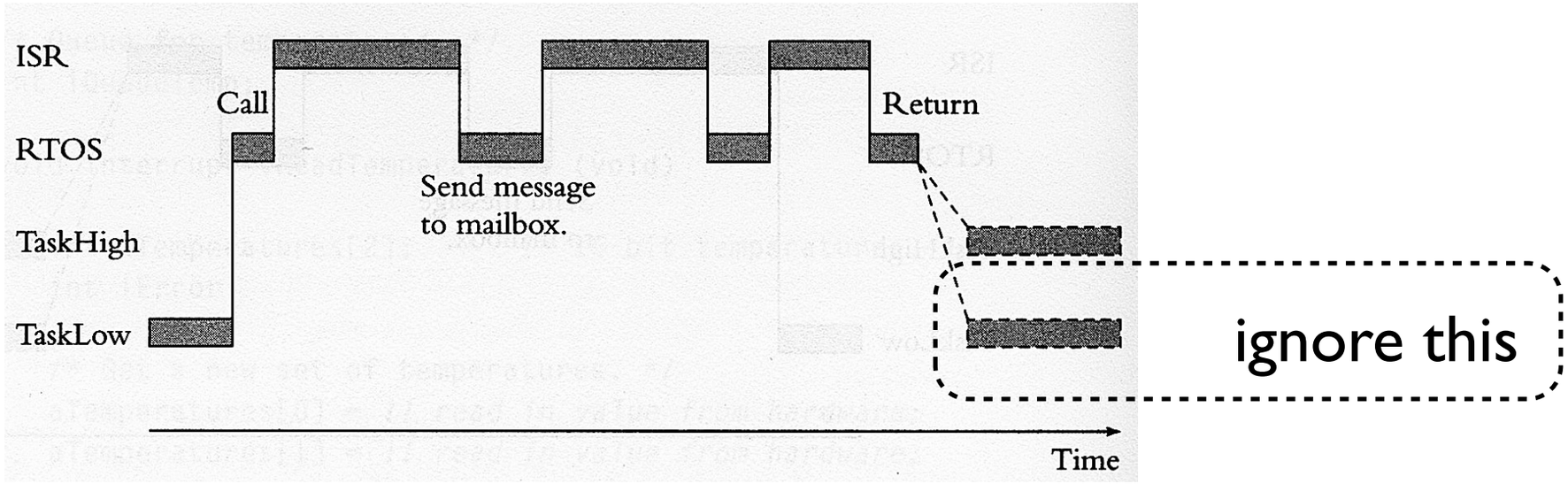

- Ideally, when the interrupt routine exits, the RTOS arranges from the microprocessor to execute an higher level task, which was waiting on the mailbox.

# What really happens (at least the worst case scenario)

- If the higher-priority task is blocked on the mailbox, then as soon as the interrupt routine writes to the mailbox, the RTOS unblocks the higher-priority task.
- Instead of returning to the interrupt routine, the RTOS switches to the higher-priority task.



# How to solve this problem?




- The RTOS intercepts all the interrupts and then calls your interrupt routine.

- When the interrupt routine writes to the mailbox, the RTOS knows to return to the interrupt routine and not to switch tasks, no matter what task is unblocked by the write to the mailbox.

- When the interrupt routine is over, it returns, and the RTOS gets control again.

# Rule 2 and nested interrupts

- Nested interrupts: when one interrupt can be interrupted by another higher priority interrupt.
- If an higher-priority interrupt, show up when a low-priority interrupt is is running... a RTOS function must be made.
- The RTOS scheduler should **NOT** run until all interrupts are complete!