Embedded Systems

Real Time Systems (Part III)

Outline

• Deadlock
• Task synchronization
  – Unilateral rendezvous
  – Bilateral rendezvous
• Event flags
  – Disjunctive synchronization
  – Conjunctive synchronization
• Intertask communication
  – Message mailboxes
  – Message queues
**Deadlock**

- A *deadlock* is a situation in which two tasks are each unknowingly waiting for resources held by the other.
- Assume Task T1 has exclusive access to Resource R1 and Task T2 has exclusive access to Resource R2.
- If T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue.

**Avoiding Deadlock**

- The simplest way to avoid a deadlock is for tasks to:
  - Acquire all resources before proceeding,
  - Acquire the resources in the same order, and
  - Release the resources in the reverse order.
- Most kernels allow you to specify a timeout when acquiring a semaphore
  - This feature allows a deadlock to be broken.
- If the semaphore is not available within a certain amount of time, the task requesting the resource resumes execution.
- Some form of error code must be returned to the task to notify it that a timeout occurred.
Task Synchronization

- A task can be synchronized with an ISR (or another task when no data is being exchanged) by using a semaphore
- The semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key)

Task Synchronization (continued)

- When used as a synchronization mechanism, the semaphore is initialized to 0
- Using a semaphore for this type of synchronization is called a *unilateral rendezvous*
- For example, a task can initiate an I/O operation and then wait for the semaphore
- When the I/O operation is complete, an ISR (or another task) signals the semaphore, and the task is resumed
Task Synchronization (continued)

- If the kernel supports counting semaphores, the semaphore accumulates events that have not yet been processed.
- Note that more than one task can be waiting for an event to occur.
- In this case, the kernel signals the occurrence of the event either to:
  - The highest priority task waiting for the event to occur or
  - The first task waiting for the event.
- Depending on the application, more than one ISR or task can signal the occurrence of the event.

Task Synchronization (continued)

- Two tasks can synchronize their activities by using two semaphores (bilateral rendezvous).
- A bilateral rendezvous is similar to a unilateral rendezvous, except both tasks must synchronize with one another before proceeding.
- A bilateral rendezvous cannot be performed between a task and an ISR because an ISR cannot wait on a semaphore.
**Bilateral Rendezvous Pseudocode**

```plaintext
Task1() {
    while(1) {
        Perform operation;
        Signal task #2;
        Wait for signal from task #2;
        Continue operation;
    }
}

Task2() {
    while(1) {
        Perform operation;
        Signal task #1;
        Wait for signal from task #1;
        Continue operation;
    }
}
```

**Event Flags**

- Event flags are used when a task needs to synchronize with the occurrence of multiple events.
- The task can be synchronized when any of the events have occurred, which is called *disjunctive synchronization* (logical OR).
- A task can also be synchronized when all events have occurred, which is called *conjunctive synchronization* (logical AND).
Event Flags (continued)

- Common events can be used to signal multiple tasks
- Depending on the kernel, a group consists of 8, 16, or 32 events, each represented by a bit
- Tasks and ISRs can set or clear any event in a group
- A task is resumed when all the events it requires are satisfied
- Kernels, like uC/OS-II, which support event flags offer services to SET event flags, CLEAR event flags, and WAIT for event flags (conjunctively or disjunctively)

Intertask Communication

- It is sometimes necessary for a task or an ISR to communicate information to another task
- This information transfer is called *intertask communication*
- Information can be communicated between tasks in two ways:
  - through global data or
  - by sending messages
**Intertask Communication (continued)**

- When using global variables, each task or ISR must ensure that it has exclusive access to the variables
  - If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts
- If two tasks are sharing data, each can gain exclusive access to the variables either by disabling and enabling interrupts or with the use of a semaphore
- A task can only communicate information to an ISR by using global variables
- A task is not aware when a global variable is changed by an ISR, unless the ISR signals the task by using a semaphore or unless the task polls the contents of the variable periodically
- To correct this situation, use either
  - A *message mailbox* or
  - A *message queue*

---

**Message Mailboxes**

- Messages can be sent to a task through kernel services
- A *message mailbox*, also called a *message exchange*, is typically a pointer-size variable
- Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox
- Similarly, one or more tasks can receive messages through a service provided by the kernel
- Both the sender and receiving task agree on what the pointer is actually pointing to
Message Mailboxes (continued)

- A waiting list is associated with each mailbox in case more than one task wants to receive messages through the mailbox.
- A task desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received.
- Typically, the kernel allows the task waiting for a message to specify a timeout.
- If a message is not received before the timeout expires, the requesting task is made ready to run, and an error code (indicating that a timeout has occurred) is returned to it.

Message Mailboxes (continued)

- When a message is deposited into the mailbox, either
  - The highest priority task waiting for the message is given the message (priority-based, uC/OS-II supported), or
  - The first task to request a message is given the message (FIFO).
- The mailbox is represented by an I-beam and the timeout is represented by an hourglass.
- The number next to the hourglass represents the number of clock ticks the task will wait for a message to arrive.
Message Mailbox Kernel Services

- Kernels typically provide the following mailbox services
  - Initialize the contents of a mailbox
  - The mailbox initially might or might not contain a message
  - Deposit a message into the mailbox (POST)
  - Wait for a message to be deposited into the mailbox (PEND)
  - Get a message from a mailbox, if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox
- Message mailboxes can also simulate binary semaphores
- A message in the mailbox indicates that the resource is available, and an empty mailbox indicates that the resource is already in use by another task

Message Queues

- A message queue is used to send one or more messages to a task
  - A message queue is basically an array of mailboxes
- Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into a message queue
- Similarly, one or more tasks can receive messages through a service provided by the kernel
- Both the sender and receiving task or tasks have to agree as to what the pointer is actually pointing to
- Generally, the first message inserted in the queue is the first message extracted from the queue (FIFO)
Message Queues (continued)

- As with the mailbox, a waiting list is associated with each message queue, in case more than one task is to receive messages through the queue.
- A task desiring a message from an empty queue is suspended and placed on the waiting list until a message is received.
- Typically, the kernel allows the task waiting for a message to specify a timeout.
- If a message is not received before the timeout expires, the requesting task is made ready to run, and an error code (indicating a timeout has occurred) is returned to it.
- When a message is deposited into the queue, either the highest priority task (uC/OS-II), or the first task to wait for the message is given the message.

Message Queues (continued)

- The figure below shows an ISR depositing a message into a queue.
- Note that the queue is represented graphically by a double I-beam.
- The "10" indicates the number of messages that can accumulate in the queue.
- A "0" next to the hourglass indicates that the task will wait forever for a message to arrive.
Message Queue Kernel Services

- Kernels typically provide these message queue services:
  - Initialize the queue
    - The queue is always assumed to be empty after initialization
  - Deposit a message into the queue (POST)
  - Wait for a message to be deposited into the queue (PEND)
  - Get a message from a queue, if one is present, but do not suspend the caller if the queue is empty (ACCEPT)
    - If the queue contains a message, the message is extracted from the queue