INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

The United Nations University

SMR/774 - 14

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

CROSS-DEVLOPMENT OF EMBEDDED SYSTEMS

Chu Suan ANG
3 SS24/7
Taman Megah
47301 Petaling Jaya
MALAYSIA

These are preliminary lecture notes, intended only for distribution to participants.

MAIN BUILDING STRADA COSTIERA, 11 TEL. 22401 TELEFAX 224163 TELEX 460392 ADRIATICO GUEST HOUSE VIA GRIGNANO, 9 TEL. 224241 TELEFAX 224531 TELEX 460449
VIA BEIRUT, 7 TEL. 22401 TELEFAX 2240310 TELEX 460392

Cross-Development of Embedded Systems (2)

C S Ang

A Real-Time Kernel for Embedded Systems

- Recent surveys show that there are more then 40 real-time kernel manufacturers.
- Real-time kernels are available for 8, 16 and 32-bit processors, including proprietary and open market ones.
- The price ranges from $100 to $10,000.
- There are also a small number of real-time kernels appearing in journals, magazines and books, which are normally available in source code.
- We shall look at one designed by Jean J. Labrosse called μC/OS.

µC/OS

- Jean J. Labrosse published an early version of µC/OS in *Embedded Systems Programming* magazine in June 1992. It was written in C with the initial goal for creating a small but powerful kernel for 68HC11 microcontroller.
- It has since been extended to a portable system suitable for use with any microcontroller/microprocessor provided that it has a stack pointer and the processor status can be stacked and unstacked.
- Labrosse has written a book describing µC/OS.
  - Jean J. Labrosse, *µC/OS The Real-Time Kernel*, R & D Publications, Lawrence, Kansas. ISBN 0-13-031352-1
- The complete source listing of µC/OS is available in the book. It is also available in a companion disk which costs \$24.95 + \$15.00 for postage and handling.
- The code is protected by copyright. However, you do not need a license to use the code in your application if it is distributed in object format. You should indicate in you document that you are using µC/OS.

Main Features of μC/OS

- Portable
  - It is written in C, with a small processor specific code in assembly to *create task, start multitasking* and *perform context switching*.
  - For 80186/80188 the assemble language code is less than 4 pages.
- ROMable
- Priority driven
  - Always runs the highest priority task that is ready to run.
- Preemptive
  - When a task makes a higher priority task ready to run, the current task is preempted or suspended and the higher priority task is immediately given control of the processor.
  - Execution of the highest priority task is deterministic.
- Multitasking
  - Up to 63 tasks
- Interrupt feature
  - Interrupts can suspend the execution of a task.
  - If a higher priority task is awakened as a result of the interrupt, the highest priority task will run as soon as the interrupt completes.
  - Interrupts can be nested up to 255 levels deep.

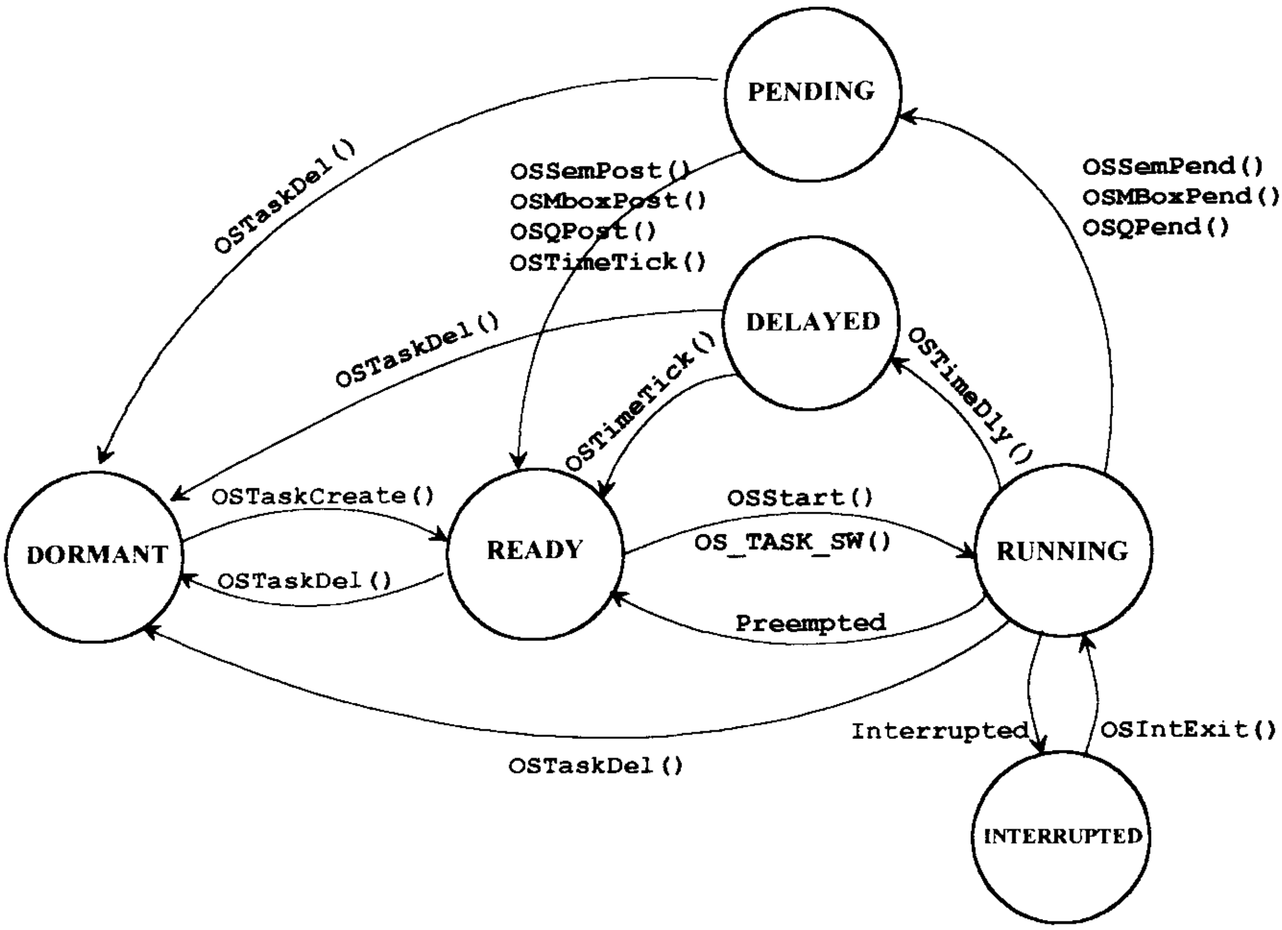
μC/OS Tasks

- A *task* is an infinite loop function or one that deletes itself when it is finished.
- The infinite loop can be preempted by an interrupt that can cause a higher priority task to run.
- A task can also call the following μC/OS services:
  - `OSTaskDel()`
  - `OSTimeDly()`
  - `OSSemPend()`
  - `OSMboxPend()`
  - `OSQPend()`
- Each task has a unique priority, ranging form 0 to 62. The lower the value the higher the task priority.

μC/OS Task States

- **DORMANT**
  - The state when a task has not been made available to μC/OS.
- **READY**
  - When a task is created by calling `OSTaskCreate()`, it is in the **READY** state.
  - Tasks may be created before multitasking starts or dynamically by a running task. If the created task has a higher priority than its creator, the created task is immediately given the control of the processor.
  - A task can return itself or another task to the **DORMANT** state by calling `OSTaskDel()`.
- **RUNNING**
  - The highest priority task created is in the **RUNNING** state when multitasking is started by calling `OSStart()`.
- **DELAYED**
  - The running task may call OSTimeDly() and enters the **DELAYED** state. The next highest priority task then runs.
  - The delayed task is made ready to run by `OSTimeTick()` when the desired delayed time expires.
- **PENDING**
  - The running may have to wait for an event by calling `OSSemPend()`, `OSMboxPend()` or `OSQPend()`. It then enters the **PENDING** state. The next highest priority task then runs. The task is made ready when the event occurs.
  - The occurrence of an event may be signalled by another task or by an interrupt service routine (ISR).
- **INTERRUPTED**
  - A task may be interrupted and enters the **INTERRUPTED** state. The ISR then runs. The ISR may make one or more tasks ready to run.
  - When all tasks are either waiting for events or delayed, an idle task `OSTaskIdle()` is executed.

μC/OS Task State Transition Diagram



Task Control Block

- Each task has a task control block, `OS_TCB`, which is used by μC/OS to maintain the state of the task when it is preempted. When the task regains control the `OS_TCB` allows it to resume execution properly.
- Each `OS_TCB` has the following field:
  - `OSTCBStkPtr` - points to the top of stack.
  - `OSTCBStat` - state of the task. 0 - ready to run
  - `OSTCBPrio` - task priority. 0 - 63
  - `OSTCBDly` - number of clock ticks the task is to wait for an event.
  - `OSTCBX`, `OSTCBY`, `OSTCBBitX`, `OSTCBBitY` - used for speeding up task handling by precomputing some parameters.

```
OSTCBX      =   priority & 0x07;
OSTCBBitX   =   OSMapTle[priority & 0x07];
OSTCBY      =   priority >> 3;
OSTCBBitY   =   OSMapTbl[Priority >>3];
```

  - `OSTCBNext`, `OSTCBPrev` - to doubly link `OS_TCB`s. `OSTimeTick()` use this link to update `OSTCBDly` field for each task.
  - `OSTCBEventPtr` - points to an event control block.
- All **OSTCB**s are placed in **OSTCBTbl[]**. The maximum number of task is declared in the user's code. An extra **OSTCB** is allocated for the idle task.

Creating a Task

- Tasks are created by calling `OSTaskCreate()` which is target processor specific.
- Tasks can either be created prior to the start of multitasking or by another task at run time.
- A task cannot be created by an interrupt service routine.
- `OSTaskCreate()` has four arguments:
  - *task* - points to the task code.
  - *data* - points to a user definable data area that is used to pass arguments to the task.
  - *pstk* - points to the task stack area for storing local variables and register contents during an interrupt.
  - *p* - task priority.
- `OSTaskCreate()` calls `OSTCBInit()` which obtains an `OS_TCB` from the list of free `OS_TCB`s. If all `OS_TCB`s have been used, an error code is returned. If an `OS_TCB` is available, it is initialised.
- A pointer the `OS_TCB` is place in the `OSTCBPrioTble[]` using the task priority as the index.
- The `OS_TCB` is then inserted in a doubly linked list with `OSTCBList` pointing to the most recently created `OS_TCB`.
- The task is then inserted in the ready list.
- Is a task is created by another task, the scheduler is called to determine if the created task has a higher priority than its creator. If so, the new task is executed immediately. Otherwise, control is returned to its caller.

Deleting a Task

- A task may return itself or another task to the DORMANT state by calling `OSTaskDel()`.
- The idle task cannot be deleted.
- The steps:
  - Removed from the ready list.
  - `OS_TCB` is unlinked and returned to the list of free `OS_TCB`.
  - If `OSTCBEventPtr` field in nonzero, the task must be removed from the event waiting list.

Task Scheduling

- Task scheduling is done by **OSSched()** which determines which task has the highest priority and thus will be the next to run.
- Each task has a unique priority number between 0 and 63. Priority 63, the lowest, is assigned to the idle task when µC/OS is initialised.
- Each task that is ready to run is placed in a ready list.
- The task scheduling time is constant irrespective of the number of tasks created.
- **OSSched()** looks for the highest priority test and verifies that it is not the current task to prevent unnecessary context switch.
- A context switch is then carried out by **OS_TASK_SW()**.
- **OSSched()** runs in a critical section to prevent ISR from changing the ready status of a task.

Interrupt Processing

- μC/OS requires an *interrupt service routine* (ISR) written in assembly language.
- Interrupts are enabled early in the ISR to allow other higher priority interrupts to enter.
- `OSIntEnter()` is called on entering and `OSIntExit()` on leaving the ISR to keep track of the interrupt nesting level. There may be 255 levels.
- μC/OS's worst case interrupt latency is 550 MPU clock cycles (80186/80188).
- μC/OS's worst case interrupt response time is 685 MPU clock cycles (80186/80188).

Clock Tick

- Time measurement in suspending execution and in waiting for an event is provided by `OSTimeTick()`, which supplies the *clock ticks* or the heartbeats.
- `OSTimeTick()` also decrements the `OSTCBDly` field for each `OS_TCB` that is not zero.
- The time between tick interrupts is application specific and is typically between 10 ms and 200 ms.
- `OSTimeTick()` increments a 32-bit variable OSTime since power up. This provides a system time.

Communication and Synchronisation

- μC/OS supports message *mailboxes* and *queues* for communication.
  - A task can deposit, through a kernel service, a message (the pointer) into the mailbox. Similarly, one or more tasks can received messages through a service provided by the kernel. Both the sending and receiving task have to agree as to what the pointer is pointing to.
  - A message queue is an array of mailboxes.
- μC/OS supports *semaphore* (0-32767) for synchronisation and coordination.
- These services are *events.*
- A task can signal the occurrence of an event (**POST**) or wait for an event to occur (**PEND**).
- ISR can **POST** an event but cannot **PEND** on an event.
- When an event occurs, the highest priority task waiting for the event is made ready to run.

Event Control Blocks

- The state of an event consists of:
  - the event itself - a counter for semaphores, a message for mailboxes, and a message queue for queues,
  - a waiting list for tasks waiting for the event to occur.
- Each event is assigned an Event Control Block which has the following data structure:
  - `OSEventGrp`
  - `OSEventTbl[8]`
  - `OSEventCnt` for semaphore count
  - `OSEventPtr` for mailbox or queue

Memory Requirements

- Program memory - less than 3150 (for 80186/80188)
- Data memory
  - 200
  - +((1 + OSMAX_TASK) * 16)
  - +(OS_MAX_EVENTS * 13)
  - +(OS_MAX_QS * 13)
  - +SUM(Storage requirements for each message queue)
  - +SUM(Storage requirements for each task stack)
  - +(OS_IDLE_TASK_STK_SIZE)
- Example: 20 tasks, 256 bytes for each task stack, 10 semaphores, 5 mailboxes and 5 queues of 10 entries would require 6337 bytes of RAM.

Kernel Services

#	Service	Description
1	`OSInit()`	Initialise µC/OS
2	`OSIntEnter()`	Signal ISR entry
3	`OSIntExit()`	Signal ISR exit
4	`OSMboxCreate()`	Create a mailbox
5	`OSMboxPend()`	Pend for message from mailbox
6	`OSMbox Post()`	Post a message to mailbox
7	`OSQCreate()`	Create a queue
8	`OSQPend()`	Pend for message from queue
9	`OSQPost()`	Post a message to queue
10	`OSSchedLock()`	Prevent rescheduling
11	`OSSchedUnlock()`	Allow rescheduling
12	`OSSemCreate()`	Create a semaphore
13	`OSSemPend()`	Wait for a semaphore
14	`OSSemPost()`	Signal a semaphore
15	`OSStart()`	Start multitasking
16	`OSTaskChangePrio()`	Change a task's priority
17	`OSTaskCreate()`	Create a task
18	`OSTaskDel()`	Delete a task
19	`OSTimeDly()`	Delay a task for n system ticks
20	`OSTimeGet()`	Get current system time
21	`OSTimeSet()`	Set system time
22	`OSTimeTick()`	Process a system tick

A µC/OS Programming Example

- An example to show a number of µC/OS features.
- 6 tasks are created.
- `TaskStat()` - First task to execute. It creates the other five which have higher priorities. It then displays statistics on the screen.
- `TaskKey()` - Monitors the keyboard. A message is sent to `Task1()` through a mailbox if key `1` is pressed.
- `Task1()` - Waits for message from TaskKey(). If not received within 36 system ticks, a timeout counter is incremented. Otherwise, a message counter is incremented.
- `Task2()` - Like Task1() except it waits for messages from a queue.
- `Task3()` - Displays one of four characters at random positions on the upper right hand side of the screen.
- `TaskClk()` - Displays date and time on the lower right hand corner.