**1. Introduction**

The OS used in the SDK of the AP80 series chips is based on FreeRTOS and modified to be a hard real-time preemptive kernel. It has functions such as task management, queue management, semaphore management, mutex lock management, and memory management.

**2. Task Management and Scheduling**

**2.1. Task States**

Tasks have four states:

Running state: The task is currently running;

Blocked state: If a task is waiting for an event;

Suspended state: A task in the suspended state is invisible to the scheduler;

Ready state: The task is in a non-running state, neither blocked nor suspended;

As shown in the figure below, after a task is created, it enters the Ready state. After the task is executed, it enters the Running state. After the task yields the CPU, it returns to the Ready state. After calling the vTaskSuspend function to suspend the task, the task enters the Suspended state. When the task is in the Suspended state, calling the vTaskResume function to resume the task causes the task to enter the Ready state. When a task is in the Running state and attempts to acquire resources such as messages, semaphores, or mutexes, if the task cannot acquire these resources, it enters the Blocked state. When the resources become available, the task returns to the Ready state.

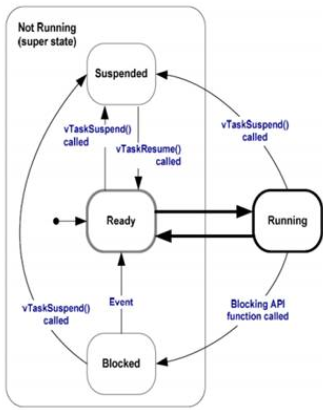


Figure 1 Task status transition diagram

**2.2. Task Scheduling**

The task scheduler in the OS uses priority-based preemptive scheduling. In the system, except for interrupt handling functions, the code for locking the scheduler, and code that disables interrupts, which are non-preemptive, all other parts of the system are preemptive, including the task scheduler itself. The system supports a total of CFG\_MAX\_PRIORITIES priority levels (0 to CFG\_MAX\_PRIORITIES-1, with higher values indicating higher priority, and 0 being the lowest priority). When a task with a higher priority than the current task becomes ready, the current task is immediately preempted, and the high-priority task takes over the processor for execution.

As shown in the figure below, there are currently four tasks. Task1 and Task2 have the same priority, which is the lowest among the four tasks; Task3 has a higher priority than Task1 and Task2 but lower than Task4; Task4 has the highest priority. t1, t2, t3, t4, t5, t6, and t7 are time slice scheduling moments. At time t1, Task1 begins execution; by time t2, since Task1 and Task2 have the same priority, these two tasks share the CPU, and Task2 begins execution; by time t3, Task1 resumes execution; by time a, Task3 enters the ready state. Since Task3 has a higher priority than Task1, Task3 preempts the CPU and begins execution; At time b, Task 4 enters the ready state. Since Task 4 has a higher priority than Task 3, Task 4 preempts the CPU and begins execution; at time c, Task 4 completes execution and releases CPU resources, allowing Task 3 to resume execution; at time d, Task 3 completes execution and releases CPU resources, allowing Task 1 to resume execution.

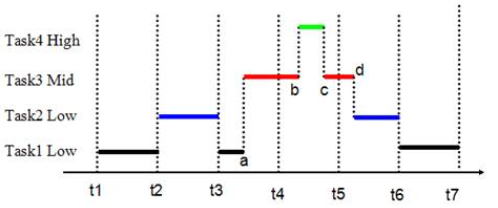


Figure 2 Task switching diagram

**2.3. Task Management Interface**

**2.3.1. Create Task**

OSTaskCreate(TaskEntry, TaskName, TaskStackSize, TaskPara, TaskPri, TaskHandler)

Parameter Description:

TaskEntry: A pointer to the implementation function of the task (effectively just the function name).

TaskName: A descriptive task name. This parameter is not used by the OS and is solely for debugging purposes.

TaskStackSize: The amount of stack space allocated by the kernel. This value specifies the number of bytes (byte) allocated for stack space.

TaskPara: The task function accepts a pointer to void (void\*). TaskPara is the value passed to the task.

TaskPri: Task priority. The priority range can be from the lowest priority 0 to the highest priority (CFG\_MAX\_PRIORITIES – 1).

TaskHandler: Used to pass the task handle. This handle will be referenced in API calls for the created task, such as changing the task priority or deleting the task. If the application does not use this task handle, TaskHandler can be set to NULL.

Return values:

There are two possible return values:

1 indicates that the task was created successfully.

-1 indicates that the OS could not allocate structure data and task stack due to insufficient memory heap space, so the task could not be created.

**2.3.2. Delete a task**

OSTaskExit(TaskHandler)

Parameter description:

TaskHandler is the handle of the task to be deleted (target task).

**2.3.3. Change task priority**

OSTaskPrioSet(TaskHandler, NewPrio)

Parameter description:

TaskHandler is the handle of the task whose priority is being modified (i.e., the target task). A task can modify its own priority by passing a NULL value.

NewPrio specifies the priority to which the target task will be set.

**2.3.4. Blocking a Task**

OSRescheduleTimeout(TimeOut) specifies the number of clock cycles to block.

**2.3.5. Example Code**

79 // Task to be created.

 80 **void** vTaskCode( **void** \* pvParameters )

 81 {

 82     **for**( ;; )

 83     {

 84     // Task code goes here.

 85     }

 86 }

 87 // Function that creates a task.

 88 **void** vOtherFunction( **void** )

 89 {

 90     **static uint8\_t** ucParameterToPass;

 91     xTaskHandle xHandle = NULL;

 92     OSTaskCreate( vTaskCode, "NAME", 1024, &ucParameterToPass, 0, &xHandle );

 93

 94     // Use the handle to raise the priority of the created task.

 95     OSTaskPrioSet(xHandle, TASK\_IDLE\_PRIORITY + 1);

 96

 97     // Use the handle to delete the task.

 98     **if**( xHandle != NULL )

 99     {

100         vTaskDelete( xHandle );

101     }

102 }

**3. Queue Management**

**3.1. Overview**

A message queue is a commonly used method of inter-task communication. It can receive messages from tasks or interrupt service routines and cache them in its own memory space. When the message queue is empty, it can block the reading task. When new messages arrive, the blocked task will be awakened to receive and process the messages. A message queue is an asynchronous communication method.

This OS has only one message queue. If a task needs to receive messages, it must first register the set of messages it wishes to receive. The definition of the message set is as follows:

enum{

        MSG\_MAIN\_CLASS                      = 0x1000,

    MSG\_COMMON\_CLOSE,

    MSG\_EQ,

    MSG\_DEV\_CLASS                         = 0x1100,

    MSG\_USB\_PLUGIN,

    MSG\_USB\_PLUGOUT,

        MSG\_DECODER\_INT\_CLASS     = 0x1200,

    MSG\_DECODER\_XR\_DONE,

    MSG\_USB\_DEVICE\_CLASS        = 0x1300,

        MSG\_USB\_DEVICE\_INTERRUPT\_CB,

    MSG\_BT\_CLASS                            = 0x1400,

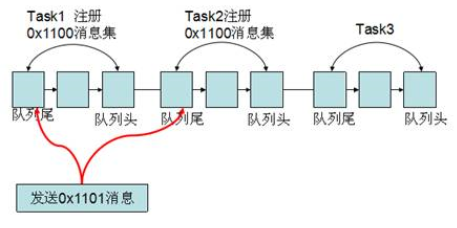
        MSG\_BT\_HF\_INTO\_MODE,

    MSG\_BT\_HF\_OUT\_MODE,

};

The message occupies a total of 16 bits, with the upper 8 bits indicating the message set category. Messages with the same upper 8 bits belong to the same message set. For example, MSG\_COMMON\_CLOSE and MSG\_EQ both belong to the MSG\_MAIN\_CLASS message set. The lower 8 bits indicate the specific meaning of the message.

As shown in the figure below, each Task occupies a portion of the message queue space. Task1 and Task2 have registered the 0x1100 message set. When sending the 0x1101 message in a Task or interrupt function, the message 0x1101 is sent to the queue spaces belonging to Task1 and Task2, respectively.



Translation:

TASK1 注册=Task1 Registration

0x1100消息集=0x1100 Message Set

Task2注册=Task2 Registration

0x1100消息集=0x1100 Message Set

队列尾=Queue Tail

队列头= Queue Head

队列尾=Queue Tail

队列头= Queue Head

队列尾=Queue Tail

队列头= Queue Head

发送0x1101消息=Send 0x1101 Message

Figure 3 Message transmission diagram

**3.2. Message Queue Interface**

**3.2.1. Register Message**

void MsgAddSet(uint32\_t Msg);

Parameter Description:

Msg The message set that the task is to receive messages from.

**3.2.2. Send Message**

int msgq\_msg\_send(int msgcode, void\* buf, int len, unsigned char prio, unsigned char async);

Parameter Description:

Msgcode Message code (the upper 8 bits of the message code represent the message set type)

buf Message entity; can be set to NULL if there is no message entity

len Length of the message entity; can be set to 0 if there is no message entity

async This parameter has no effect

**3.2.3. Receive Message**

int msgq\_msg\_recv(void\* buf, int\* len, int wait);

Parameter description:

buf The starting address of the message body. Can be set to NULL if there is no message body.

len The length of the message body.

wait The time to block the task if there are no messages.

Return value This function returns the message code (the upper 8 bits of the message code represent the type of message set).

**3.2.4. Example Code**

105 **enum**{

106 //// Device plug-and-play messages/////////////////////////////////////////////////////////////

107     MSG\_DEV\_CLASS           = 0x1100,//// The upper 8 bits of the message code represent the message set type

108     MSG\_USB\_PLUGIN,

109     MSG\_USB\_PLUGOUT,

110

111 //// Bluetooth protocol stack messages/////////////////////////////////////////////////////////////

112     MSG\_BT\_CLASS            = 0x1400,

113     MSG\_BT\_HF\_INTO\_MODE,        // Enter hands-free mode

114     MSG\_BT\_HF\_OUT\_MODE,         // Exit hands-free mode

115 }

116

117 **void** RecvTask(**void**)

118 {

119     **unsigned** Msg;

120     MsgAddSet(MSG\_DEV\_CLASS);  // Register the message set to receive

121     MsgAddSet(MSG\_BT\_CLASS);   // Register the message set to receive

122     **while**(1)

123     {  //// Receive messages

124         Msg = (**uint16\_t**)OSQueueMsgRecv(NULL, NULL, 0xFFFFFFFF);

125         APP\_DBG("\n msg=0x%x",Msg);

126     }

127 }

128

129 **void** SendTask(**void**)

130 {

131     **unsigned** i=0,j;

132     i++;

133     j=i%10;

134 //// Send messages

135     OSQueueMsgSend(MSG\_DEV\_CLASS+j, NULL, 0, 1, 0);

136     }

137 }

**4. Semaphores and Mutual Exclusion Lock Management**

**4.1. Overview**

**4.1.1. Semaphores**

Semaphores are used to solve synchronization issues between tasks. Tasks can acquire or release them to achieve synchronization or mutual exclusion. A semaphore acts like a key that locks a critical section, allowing only tasks with the key to access it: a task must obtain the key before it is allowed to enter the critical section; and upon exiting, it passes the key to the next waiting task in the queue, allowing subsequent tasks to enter the critical section sequentially.

If the semaphore value is 5, it indicates that there are 5 semaphore instances available for use. When the number of semaphore instances reaches zero, tasks attempting to acquire the semaphore will be blocked in the semaphore's waiting queue, waiting for an available semaphore.

**4.1.2. Mutexes**

A mutex is a special type of binary semaphore. A mutex has only two states: unlocked or locked (two state values). When a task holds it, the mutex is in the locked state, and the task gains ownership of it. Conversely, when the task releases it, the mutex is unlocked, and the task loses ownership of it. When a task holds the mutex, other tasks cannot unlock or hold it.

**4.2. Function Interface**

**4.2.1. Define Semaphore**

DECLARE\_SEMAPHORE(name, count) ////The initial value of the semaphore is 0 by default

Parameter Description:

name Semaphore handle

count Maximum count value; when count=1, the semaphore can be used as a mutex lock

**4.2.2. Acquiring a Semaphore**

There are two forms of acquiring a semaphore:

OSDOWN(Seam) ////When the semaphore cannot be acquired, the task will be blocked until the semaphore becomes available

\_\_OSDOWN\_TIMEOUT(sema, timeo)

////When the semaphore cannot be acquired, the task will be blocked for timeo

**4.2.3. Release semaphore**

OSUP( Seam)

**4.2.4. Example code**

140 DECLARE\_SEMAPHORE(xMutex,1);

141 **void** TakeSemaTask(**void**)

142 {

143     **while**(1)

144     {

145         OSDOWN(xMutex);// Acquire semaphore

146         // \_\_OSDOWN\_TIMEOUT(&xMutex,0x20);// Acquire semaphore

147     }

148 }

149

150 **void** GiveSemaTask(**void**)

151 {

152     **while**(1)

153     {

154         OSUP(xMutex);// Release semaphore

155     }

156 }

**5. Memory Management**

Whenever a task, queue, or semaphore is created, the kernel needs to perform dynamic memory allocation. The OS provides methods for dynamically allocating and releasing memory.

**5.1. Memory Allocation**

OSMalloc(Size, Opt)

Parameter Description:

Size: The size of the memory to be allocated.

Opt: Can take the following values:

#defineMMM\_ALLOC\_NORMAL(1 << 0) /\* if request memory use as a normal,the OS will allocate it from low memory \*/

#defineMMM\_ALLOC\_FAST(1 << 1) /\* if request memory will be free quickly,the OS do it as MMM\_ALLOC\_NORMAL do \*/

#defineMMM\_ALLOC\_TASK(1 << 2)/\* if request memory will use as long time as task life and won't be free for a while,the OS will allocate the high memory for it \*/

#defineMMM\_ALLOC\_BOOTUP(1 << 3)     /\* if request memory used as bootup memory,it won't be free unless reboot \*/

**5.2. Release memory**

OSFree(Ptr)