http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (22): 6531-6536, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.6531.6536

Design and Implementation of a Task Manager of Sensor Network Operating System

Haicheng Li
Information Technology College, Eastern Liaoning University Dandong, 118003, China

Abstract: Based on the requirements of low-power, high reliability and scalability of wireless sensor network, this article designs and implements a real-time multi-task manager of wireless sensor network. The advantage is that it provides preemptive multi-task scheduling, so sensor nodes could perform cross execution of multiple time-sensitive complex tasks, thus avoiding the situation that any long-time demanding task would block the implementation of some time-sensitive tasks. In addition, this dissertation makes improvement of the bitmap method, which is frequently used in task switching and innovatively presents a positioning switching algorithm to save the storage space of the system, to determine the switching time of system task and to improve the response speed of the system, providing systematic assurance for those real-time tasks. Finally, the system is tested to prove its performance.

Key words: Embedded system, sensor network operating system, real-time multi-task, task manager

INTRODUCTION

Due to the miniaturization of the node design and limited battery capacity, sensor network is greatly limited in the choice of operating system (Pister *et al*., 2003; Polastre, 2003). In addition, the existing embedded operating system does not take consideration on objectives such as small kernel or unrealized low power, so characteristics of wireless sensor network must be taken into consideration (Han *et al*., 2005; Kim, 2003) such as limitation in node computing (Dunkels *et al*., 2004) storage and communication, to study the micro embedded operating system suitable for wireless sensor network. This article designs and implements a wireless sensor network embedded operating system, μOs. This system provides a preemptive multi-task kernel to meet the requirements of wireless sensor network for real-time multi-task to perform reasonable and orderly scheduling on multi tasks in the sensor network.

DESIGN OF THE TASK MANAGER

The task of uOs consists of three parts, as shown in Fig. 1: Task code, task stack and task control block. Among them, the task control block is used to save task properties; task stack is used to save task working environment; task code is the execution part of the task.

The system of μOs supports at most 8 grade priority, which is 0 to 7. The smaller the priority value is, the higher

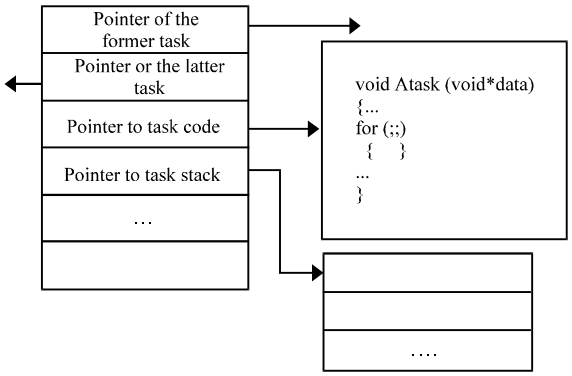

Fig. 1: Task structure of μOs

the priority is. μOs has one preemptive kernel with preemptive scheduling algorithm based on static priority and the preemptive real-time kernel would run the highest priority ready task at any time. First In First Out (FIFO) scheduling policy would be applied on other tasks with the same priority, meaning that other tasks with the same priority would be run in CPU after the completion of proceeding tasks.

REALIZATION OF THE TASK MANAGER

The system uses ATmega128, which is based on 8-bit low-power CMOS microprocessor with AVR RISC architecture, wireless transceiver chip CC2420 and Micaz communication platform of Crossbow company.

Corresponding Author: Haicheng Li, Information Technology College, Eastern Liaoning University Dandong, 18003, China, Tel :0086-15041576069

Task control block: The task control block structure is defined as follows:

```
typedef struct os_tcb {
    OS_STK*OSTCBStkPtr;
    INT8U STCBTpcp;
    struct os_tcb * OSTCBNext;
     struct os_tcb * OSTCBPrev;
    struct os_tcb * OSPrioequNext;
    INT8U OSTaskId;
    OS_EVENT * OSTCBEventPtr;
    INT16U OSTCBDly;
    INT8U OSTCBStat;
    INT8U OSTCBPrio;
    # If OS_TASK_DEL_EN
    BOOLEAN OSTCBDelReq;
    #endif
    } OS_TCB;
```

OSTCBStkPtr would link current tasks. μOs allows each task has its own stack and what is particularly important is that the stack capacity of each task can be set.

OSTCBTpcp records the highest priority limits of the resources that the tasks have. Each bit represents one highest limitation. This record is used to help the system to achieve the highest priority limitation protocol.

OSTCBNext and OSTCBPrev are used for dual link of task control block OS_TCB, the link list is used in clock tick function OSTimeTick() to refresh the task delay variable OSTCBDly of each task. When certain task is delayed or suspended, the task would be linked to the waiting queue of a double linked list and would be deleted when the task is deleted. The double linked list could quickly insert or delete any member to or from the list.

OSPrioequNext would link tasks with the same priority into a linked list and could make tasks in the system with the same priority. It is also one of the basic conditions to realize highest priority limitation protocol.

OSTaskId is the unique identifier for each task. When the system wants to delete one task, firstly the queue with same priority would be found according to the priority of the task and then, the task to be deleted would be found according to the task ID.

OSTCBDly is the variable that would be used when the task need to be delayed for a number of clock ticks, or the task needs to be suspended for some time to wait for the occurrence of certain event and such wait has timeout limit. In this case, this variable keeps the most click ticks that the task allows to wait for certain event. If this variable value is 0, it is indicated that the task does not delay or there is no limit of the waiting time wait for certain event.

OSTCBStat is the task status word. When OSTCBSta equals to 0, the task would enter into ready state. OSTCBStat could be given other values to represent different states.

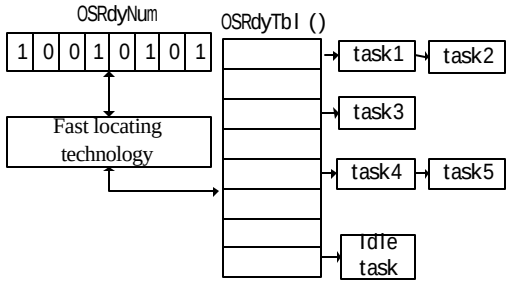

Fig. 2: Task ready list

OSTCBPrio is the task priority. OSTCBPrio value of high-priority task would be low, meaning that, the smaller the value is, the higher the priority of the task is.

OSTCBDelReq is a boolean variable to indicate whether the task needs to be deleted or not.

Realization of task scheduler: The core work of the multi-task operating system is task scheduling. Scheduling means determining which task should be run through an algorithm. The task scheduling idea of uOs is always making make task with highest priority run.

- **Task ready list:** μOs task scheduling is based on task ready as shown in Fig. 2. It can be seen from the task state diagram that, the system always selects a task in ready state to run. In order to clearly know that which task enters into the ready state and in set up a task ready in RAM. The ready state flag of each task is put into the ready list and the list contains one OSRdyNum variable and a pointer array OSRdyTbl() (the size of the array is set by the system in advance)

In the process of creating a task, tasks with different priority are connected to the corresponding OSRdyTbl() element. In this task ready list, the tasks are ordered by their priority level. Each bit of OSRdyNum represents whether there is ready task for certain priority level. If there is ready task in certain priority, the correspondent bit of OSRdyNum would be set to 1, otherwise to 0.

- **Positioning switching scheduling algorithm:**

- Add task queue with priority of prio into the ready queue

  If (OSRdyTbl [prio] = Null) OSRdyNum| = OSBitTbl [prio]
- If a task with a priority of prio is to leave the ready state

  If (OSRdyTbl [prio] == null) OSRdyNum & = ~ OSBitTbl [prio]

OSBitTbl [] is one array defined in uOs to accelerate the computing speed and the value of each element of it is:

OSBitTbl [0] = 00000001B 0 bit of OSRdyNum is set to 1

OSBitTbl [1] = 00000010B 1 bit of OSRdyNum is set to 1

OSBitTbl [2] = 00000100B 2 bit of OSRdyNum is set to 1

OSBitTbl [3] = 00001000B 3 bit of OSRdyNum is set to 1

OSBitTbl [4] = 00010000B 4 bit of OSRdyNum is set to 1

OSBitTbl [5] = 00100000B 5 bit of OSRdyNum is set to 1

OSBitTbl [6] = 01000000B 6 bit of OSRdyNum is set to 1

OSBitTbl [7] = 10000000B 7 bit of OSRdyNum is set to 1

- Obtain the ready task with highest priority from the ready list

Find lowest bit that is set to 1 in OSRdyNum and relates it to the correspondent element of OSRdyTbl [] array

Define ycg (n) (n == 0) * 0+(n == 2) * 1+(n == 4)*2+(n == 8)*3+(N == 16)*4+(n == 32)*5+(n == 64)*6+(n == 128)* 7

n = OSRdyNum and (-OSRdyNum);

This algorithm takes into account of the problem of limited storage capacity of sensor network hardware, so it does not use the traditional bitmap method. Though time complexity of bitmap method is O (1), it wastes a lot of storage space. The time complexity and space complexity of positioning scheduling algorithm is O (1).

Task scheduler: The task scheduler has two most important jobs: one is searching for ready task with the highest priority in ready list and the second is realizing task switching. The task-level scheduling is completed by OSSched().

For the realization of task switching, OSTCBHighRdy must point to the highest priority task control block OS_TCB by assigning the element subscripted with OSPrioHighRdy in OSRdyTbl[] array to OSTCBHighRdy. Then, the statistical counter OSCtxSwCtr would add 1 to keep track of task switching times. As last, the macro would call OS_TASK_SW() to complete the actual task switching. The actual task switching consists of the following two steps: Push the microprocessor register of the suspended task into the stack and then recover the register value of higher priority task from the stack back to the register. In Kernel, from the stack structure of ready task, it always seems that an interrupt just occurred, as all the microprocessor registers are stored in the stack.

μOs task management module provides the following functions:

INT8U OSTaskCreate (void (* task) (void * pd), void * pdata OS_STK the * ptos INT8U prio, INT8U, taskid);

INT8U OSTaskDel (INT8U prio, INT8U, taskid);

INT8U OSTaskSuspend (INT8U prio, INT8U, taskid);

INT8U OSTaskResume (INT8U prio, INT8U, taskid);

Actual realization of the above task management functions is as follows:

- **Task creation function OSTaskCreate ():** Description of OSTaskCreate(): The function should firstly check the validity of priority assigned to the task. The priority of the task must be within the scope that the system specifies. Then, task creation function would call task stack initialization function OSTaskStkInit(), which is responsible for the establishment of the task stack. OSTaskStkInit () function returns the new top of stack (PSP) and stores it in the 0S_TCB of task. After completion of stack establishment, OSTaskCreate() would call the task control block initialization function to obtain and initialize a task control block from a free OS_TCB pool. Once OS_TCB is assigned, the creator of the task would has full possession of it, though the kernel has also created other tasks at this time and these new tasks can not carry out any operation toward OS_TCB. If the task control block initialization function fails to return the initialization of task control block, OSTaskCreate() would give up the establishment of the task. If OSTaskCreate() function is called during the execution of a task, then task scheduling function would be called to determine whether the newly created task has higher priority than that of the original one. If the new task has higher priority, the kernel will conduct task switching from the old task to the new one. If the task is created before the start of multi-task scheduling, task scheduling function would not work
- **Task deletion function OSTaskDel ():** The specific approach is to delete the deleted task control block from the corresponding task queue and return it to the empty task control block linked list and then check the ready list to determine whether to set the ready status of the task to 0. Sometimes, the task would occupy some resources. At this time, if other tasks delete this task, then the resources of the

deleted task would be lost when the resource is not released. Therefore, when deleting a task that occupying some resource, the task presenting a deletion request is only responsible for deleting task request and the deletion job would be performed by the deleted task itself. So, the deleted task would determine when to delete itself and release the occupied resource before deleting itself

- **Suspending function OSTaskSuspen():** Specific operation of OSTaskSuspend() function is: First, the function should make sure that the user's application is not suspended as idle task and then confirm the user-specified priority is valid. The task to be suspended should be searched according to task ID. Then, OSTaskSuspend () would verify whether the user suspends the task that calls this function by specifying OS_PRIO_SELF. Users can also specify the priority and the task ID to suspend the task that calls this function. The task scheduler would be called in both cases. If the user does not suspend the task calling this function, there is no need that OSTaskSuspend() run the task scheduler as the task being suspended has lower priority. Then, the function would verify whether the suspended task exists. If this task exists, it will be removed from the ready list. The task to be suspended may not be in the ready list, because it is likely waiting for event occurrence or expiration of delay. In such case, the corresponding bit in OSRdyNum of the suspended task has been cleared (i.e., zero). It is faster to clear the bit again than verifying its clear status in advance, so the bit would not be verified to clear directly. OSTaskSuspend() could set OS_STAT_ SUSPEND flag in OS_TCB of the task to indicate that the task is being suspended. Finally, OSTaskSuspend() could call the task scheduler only that the suspended task calls this function itself
- **Task resume function OSTaskResume ():** In uKernal, the suspended task could only resume through calling OSTaskResume(). As OSTaskSuspend can not suspend idle task, it is necessary to confirm the user's application is not restoring the idle task. The tasks to be restored must exist, because users need to operate its task control block OS_TCB and the task must be suspended. OSTaskResume() cancels suspending through clearing the bit of OS_STAT_SUSPEND in the OSTCBStat domain. To put tasks in ready state, OS_TCBDly must be 0, it is because that in OSTCBStat there is no sign indicating that the task is waiting for the delay expiration. Only when the above two conditions are satisfied, the task would enter into ready state. Finally, the task scheduler would check whether the priority owned by resumed task is higher than that of the task that calls this function

REALIZATION OF SYSTEM CLOCK

μOs requires the user to provide periodic signal source to implement the time delay and confirm the overtime. The beat rate shall be between 10 to 100 times per second, i.e. 10 to 100 Hz. Higher the clock rate is, heavier the system additional load. The actual frequency of the clock beat depends on the accuracy of user applications. The source of clock beat can be a specialized hardware timer, or it can be from a 50/60 Hz AC power signal.

The start of clock metronome must be after the start of multitasking system, namely in the call OSStart(). In other words, the first thing to do after calling OSStart()is to initialize the timer interrupt. Usually, the common mistake is putting the allowance of clock metronome interrupt after the system initialization function OSInit() and before calling multitasking system start-up function

```
OSStart ().
    void main (void)
    {OSInit ();

    Call OSTaskCreate () to create at least one task;
Allow clock beat (TICKER) to interrupt;
OSStart ();
}
```

The potential danger here is that clock interrupt may happen before μOs starts the first task, at this time μOs is in a state of uncertainty, the user application is likely to collapse. The clock beat service in μOs is realized by calling OSTimeTick() in the interrupt service subroutine. Clock beat interrupt obey all the rules described in the previous section. Clock beat interrupt service subprogram's signal code is as flows.

```
    void OSTickISR (void)
    {
    Save the value of the processor registers;
    Call OSIntEnter ();
    Call OSTimeTick ();
    Call OSIntExit ();
    Restore the value of the processor registers;
    Execute interrupt return instructions;
    }
```

Clock beat function OSTimetick() starts from calling the clock beat outconnection definable function OSTimTickHook(), which can extend the clock beat function OSTimtick(). Calling OSTimTickHook() first is to give the user a chance to do something at the start

of clock beat interrupt service, because some time-demanding work may need to be done by the user. A large amount of work in OSTimeick() is to minus 1 to each user task control block OS_TCB's time delay OSTCBDly (if it is not zero). OSTimeTick() starts from the OSTCBList, along the OS_TCB linked list to the idle task. When the time delay item OSTCBDly of some task's task control block fell to zero, then this task enters the ready state, however, the pending tasks of the pending function OSTaskSuspend() will not enter the ready state. The execution time of OSTimeTick() is proportional to the number of tasks established in the application. OSTimeTick() also accumulated time by calling OSTime(), using an unsigned 32-bit variable. Note that using the interrupt before adding 1 to OSTime, because most of the microprocessor has to use multiple instructions to add 1 + 32 digits.

PERFORMANCE TEST

Test of minimum memory space: Least kernel code space refers to the program space needed by the operating system to complete the most basic function. Least kernel code space is the minimum program space needed to judge whether an operating system is available. As the basic function loaded in the operating system kernel each time is different, the code volume is also different, so the value of the least kernel code is not unique, the least value is a relative thing.

Least kernel code space test scheme is as follows:

- Write a test program min_code.C. Install all the library files needed for the operating system in the test program, including task management, memory management, clock management, semaphore and exclusive semaphore. Test program, after the initialization of the kernel, in addition to start the Idle task, will start one task to complete system function, which can guarantee the kernel size to be the least value
- Compile and link test procedure in AVR STUDIO, generate hexadecimal .hex file, then use JTAG emulator to burn .hex file into Micaz node, finally use JTAG emulator to read the usage of FLASH, calculate the size of the code space, which is the least value of the kernel code space for the current time
- After finishing a test, modify the test program code. Load other system functions into the kernel code. Then repeat steps (2)
- Test repeatedly, record the minimum, maximum and average values. The results 200 test is shown in Table 1

Table 1: Least kernel code space of μOS

Testing times	Minimum value (KB)	Maximum value (KB)	Average value (KB)
50	3.238	3.452	3.316

It can be seen from the results that, in 50 tests, the least memory space of uOs kernel could be 69 bytes and the average value is less than 146 bytes, which is far less than the RAM number on MCU chip in Micaz nodes of wireless sensor nodes (4 KB), so it leaves a larger memory space for application development.

Test of minimum memory space: The test program of minimum memory space is as follows:

- Coding test program min_mem.c. Load all the library files required by the operating system into the test program, including task management, memory management, clock management, semaphore and mutex semaphore. After initialization of the kernel, except for the idle task, the test program only starts one task to perform system function. In this way, it can be ensured that the kernel size is the smallest
- Compile and link test program in AVR STUDIO to generate one hexadecimal .hex file and then use the JTAG emulator to burn the .hex file into the Micaz node and at last make use of the JTAG emulator to read the RAM usage condition and calculate the memory space size of this time and the size of the memory space is the value of the least memory space at this time
- After completing primary test, modify the code of the test program. Load the kernel to perform other system functions and then repeat processes in step (2) to re-test
- Perform multiple tests to record the minimum, maximum and average value

The results 200 test is shown in Table 2.

Test of task switching time: Task switching time is an important indicator to reflect the performance of real-time operating system. In the test, 10 different tasks with priorities from the highest to the lowest level and task switching is performed among these tasks to calculate the time of task switching. As in the μOs kernel, scheduling function _thread_sched () performs jobs of saving current task context, selecting new task and resuming context of new task. Therefore, the execution time of scheduling function _thread_sched( ) is just the task switching time of Kernel kernel.

Table 3 records the minimum, maximum and average value of uOs kernel task switching time in 200 tests.

Table 2: Least memory space of μOs

Test no.	Minimum value (Byte)	Maximum value (Byte)	Average value (Byte)
50	69	189	145.5

Table 3: Task switch time of μOs

Test no.	Minimum value (μs)	Maximum value (μs)	Average value (μs)
50	23.25	42.18	35.65

Task switching time is divided into three parts: time of saving the context of the currently running task; time of selecting the task to run from the task READY queue; the time of restoring the context of the task to be run. In uOs kernel, each time of task switching, the time of saving the current running task context and the time of resuming the context of the task to be run are the same. The time difference of task switching lies in that, the time is different each time to select a task to run from the READY queue.

CONCLUSION

This study designs and implements multi-task management of embedded operating system for wireless sensor network, presents a new positioning scheduling switching algorithm to save the storage space of the system, to determine the specific time of system task switching time and to improve the system response speed; the system makes use of preemptive real-time kernel and a high-priority scheduling algorithm. The performance analysis shows that the task switching time of this system is close to other commercial embedded systems.

ACKNOWLEDGMENTS

This study is supported by the Science and Technology Foundation of the Education Department of Liaoning Province (No .2008212, No. L2010141). the National Natural Science Foundation of Eastern Liaoning University (2009-Z03).

REFERENCES

Dunkels, A., B. Gronvall and T. Voigt, 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, November 16-18, 2004, Tampa, FL., USA., pp: 455-462.

Han, C., R. Kumar, R. Shea, E. Kohler and M. Srivastava, 2005. SOS: A dynamic operating system for sensor networks. Proceedings of the 3rd International Conference on Mobile System, Applications and Services, June 5-7, 2005, Washington, USA., pp: 163-176.

Kim, K.H., 2003. Basic program structures for avoiding priority inversions. Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, May 14-16, 2003, Hakodate, pp: 26-34.

Pister, K., B. Hohlt, J. Jeong, L. Doherty and J.P. Vainio, 2003. Ivy: A sensor network infrastructure for the college of engineering. http://www-bsac.eecs.berkeley.edu/projects/ivy/

Polastre, J.R., 2003. Design and implementation of wireless sensor networks for habitat monitoring. Master's Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.