Design and Implementation of a Flexible and Memory Efficient Operating System for Sensor Nodes

R. C. Hansdah   Deepak Ravi   Sahebrao Sidram Baiger   Amulya Ratna Swain

IISc-CSA-TR-2010-5
http://archive.csa.iisc.ernet.in/TR/2010/5/

Distributed Computing Laboratory
Computer Science and Automation
Indian Institute of Science, India

August 2010

Design and Implementation of a Flexible and Memory Efficient Operating System for Sensor Nodes

R. C. Hansdah Deepak Ravi Sahebrao Sidram Baiger
Amulya Ratna Swain

Abstract

Sensor network nodes exhibit characteristics of both embedded systems and general-purpose systems. A sensor network operating system is a kind of embedded operating system, but unlike a typical embedded operating system, sensor network operating system may not be real time, and is constrained by memory and energy constraints. Most sensor network operating systems are based on event-driven approach. Event-driven approach is efficient in terms of time and space. Also this approach does not require a separate stack for each execution context. But using this model, it is difficult to implement long running tasks, like cryptographic operations. A thread based computation requires a separate stack for each execution context, and is less efficient in terms of time and space. In this paper, we propose a thread based execution model that uses only a fixed number of stacks. In this execution model, the number of stacks at each priority level are fixed. It minimizes the stack requirement for multi-threading environment and at the same time provides ease of programming. We give an implementation of this model in Contiki OS by separating thread implementation from protothread implementation completely. We have tested our OS by implementing a clock synchronization protocol using it.

1 Introduction

Wireless Sensor Networks(WSNs) typically contain large (hundreds or thousands) numbers of tiny sensor nodes, also called motes. These sensor nodes can communicate, compute, sense, and store data. But they have many constraints like program memory, processing power, battery power, available bandwidth. The processing power and program memory is restricted to minimize power consumption as the motes run on battery power. Therefore, it is important to design operating system for sensor nodes that use only fixed size memory rather than memory allocated at run time, i.e., dynamic memory such as stack for each thread. Typical applications of sensor networks include battlefield surveillance, forest fire detection, home automation, traffic monitoring, object tracking etc. When the sensor network detects an event, it propagates the information to the base station for further action. Once deployed, the sensor network is expected to work correctly for many years without any user intervention. The reminder of this report in organized as follows. In the following, we give a brief overview of existing work and motivation for our work. Section 2 describes proposed model of OS and its implementation. Section 3 gives details of various thread APIs and synchronization APIs that we have implemented. Experimentation results are discussed in section 4. Section 5 ends the paper with a concluding remark.

1.1 Available Operating Systems

SOS[12] is an example of a kernel-based operating system. SOS kernel implements message passing, memory management, timers, module loading and unloading and other services. The system calls are specific to each of the services. To avoid dependency of a module on kernel implementation, these system calls are implemented using *jump tables*. *Jump tables* provide one level of indirection and avoid module recompilation whenever kernel changes.

Mantis Operating System(MOS)[4] kernel provides services like thread management, communication device management and input-output device management. The I/O device management service provides a UNIX like uniform interface to the underlying hardware.

Resilient Extensible and Threaded Operating System(RETOS)[6] implements its kernel using layered architecture, viz. static kernel and dynamic kernel. The static kernel consists of mainly thread management and module management. Dynamic kernel implements the common libraries.

TinyOS(TOS)[13] follows kernel-less architecture. A TinyOS application consists of a set of components which interact with other components through commands and signaling of events. The core part of the operating system is also provided as components.

The t-kernel[11] is an example of virtual machine based operating system that is identical to the underlying hardware. The t-kernel provides additional functionality like software based memory protection, virtual memory etc. using load time code modification. The detail survey of existing sensor network operating systems is given in [16].

1.2 Execution Models

There are various execution models for both embedded and general purpose operating system, but they are not directly applicable to sensor network operating system. In this section, we examine common execution models in sensor network operating systems.

Event based : In event based approach, events trigger the execution of event handlers. The events are usually generated by an interrupt caused by timer, sensor devices etc. The event handlers are usually short to avoid long period between generation and processing of the event. To support longer computation, the tasks are usually scheduled by an event handler to be executed either by directly modifying the scheduler data structure, or by setting a bit in event handler which is polled after the execution of an event handler.

Event driven programming model is usually more efficient in terms of time and space. Also event driven model does not require a separate stack for each execution context. In contrast, it is difficult to implement long running tasks using event driven programming model. Event driven model exposes the control flow information.

TinyOS[14] follows event based approach. This allows to have only one stack and single execution context. But it delivers interrupt as asynchronous events which will preempt the running task. TinyOS uses a single queue with length 7 and a two level scheduling. Events have high priority and can preempt tasks that have low priority, but the reverse is not true. Tasks always run to completion with respect to other tasks, and do not preempt each other. The lowest level events are directly triggered by hardware interrupts. Typically when an interrupt triggers an event, that event is processed by a event handler, which will post tasks to the task queue. Events are not supposed to use lot of time, but the task are used to do computation. So longer computation is supported by using tasks by queuing it in a single FIFO scheduler. Though this avoids context switches and need for separate stacks, it restricts TinyOS to run only one task at a time.

Thread based : In thread based approach, each thread can be preempted at any time without polling the scheduler, though it is usually done using a fixed or variable interval timer. Use of variable timer is preferred compared to fixed interval timer, as it reduces energy consumption and decreases the latency. Though this approach makes programming computation oriented tasks easier, it introduces the complexity of context switches and need for separate stacks. To support sharing of data across threads, operating system should also provide one or more synchronization strategies like message based, lock-based synchronization etc.

MOS[4] is based on thread based approach, and is currently implemented using a fixed interval timer. MOS uses lock-based synchronization approach and provides mutexes, read-write locks and semaphores as synchronizing primitives.

Fiber based : In fiber-based approach, each fiber has to explicitly yield to another fiber and thus uses co-operative/non-preemptive scheduling. This approach is usually implemented along with event driven approach or thread based approach. This approach avoids the need for timers for context-switch. Fibers are usually implemented using co-routines. This approach makes it difficult to implement computation oriented tasks like cryptographic operations, as the task has to explicitly yield the CPU to another fiber. Since fiber uses co-operative scheduling, sharing of data across fibers does not require any explicit synchronization strategy.

TinyOS fibers provide threading support in terms of fibers with blocking I/O calls.

TinyThread : TinyThreads is a multi-threading library for TinyOS with co-operative scheduling. A separate stack is kept for each thread. TinyThread library provides an option of providing preemptive scheduling.

Hybrid : Event and thread based approach can also be supported simultaneously in a same system. Contiki supports preemptive scheduling as a library. Allocating a separate stack for each thread may not satisfy the memory constraints. Fiber-based approach usually require a single shared stack but does not support preemptive scheduling. A hybrid approach uses less stack space but supports preemptive scheduling.

Y-Thread : Y-threads[15] are similar to fibers but supports preemptive scheduling. It uses single shared stack for non blocking operations but uses multiple separate stack for blocking operations. Y-threads thus consume lesser amount of stack space compared to pure preemptive scheduling.

1.3 Shared Stack vs Multi-Stack

In a single shared stack implementation of kernel stack, data in a stack is lost during a context switch. The function to poll the scheduler is usually modeled as a function that never returns. RETOS uses single shared stack for kernel.

In a multi-stack implementation, data in a stack is retained over context switch. The function to poll the scheduler is usually modeled as a blocking function. Multi-stack implementation consumes more memory compared to single shared stack implementation.

Table 1 shows various sensor node operating systems and their execution model. The other sensor node operating systems included in this table are EYES[8], CORMOS[18], Nano-QPlus[2], and KOS[5].

Single Stack / Event based	Multi-Stack/ Thread based	Hybrid
TinyOS	MantisOS	Contiki
SOS	Nano-QPlus	kOS
EYES	-	-
CORMOS	-	-

Table 1. Execution model and OSs

1.4 Scheduling Model

TinyOS uses single FIFO model with fixed length queue and the tasks in TinyOS are statically scheduled. In MOS, a stack is allocated for each thread. A program in MOS can have a fixed number of threads that are statically or dynamically allocated. SOS uses priority queues for task scheduling. Contiki uses a two level scheduling hierarchy, in which all event scheduling is done at a single level. It does not disable interrupts to support real-time operations.

1.5 Ease of Application Development

A good operating system should provide an easy programming interface for the application developers. It is the same for the sensor network operating systems too.

TinyOS uses event driven model, and requires application developers to explicitly design their application using a state machine which can be difficult for some developers.

MOS abstracts the devices and provides a Unix like interface which makes it easy for application developers to write generic code. MOS also provides threads with blocking calls for easy programming.

Contiki and SOS also use event driven approach, but they provide protothread based model for ease of programming.

Most of the sensor operating systems are programmed in C. There exist few other programming languages for sensor operating systems like nesC, galsC, SQTL, etc. In TinyOS, both the operating system and the applications are written in nesC whose learning curve progress rather slowly for a normal developer.

1.6 Contiki-2.4 and Protothread

Contiki-2.4 : Contiki[9] is an open source sensor node operating system having event driven kernel. It provides dynamic loading and unloading of programs and services. Even though Contiki's kernel is event-based, preemptive multi-threading can be provided at the application layer on a per-process basis. In Contiki operating system, processes are usually implemented using protothreads to provide thread like programming style on top of the event-driven kernel.

Protothread : Protothreads[10] are stack-less functions usually implemented in C programming language. All protothreads use the same kernel stack. Activation records are not preserved across blocking wait. Protothreads are implemented using C language macro preprocessor.

1.7 Related Work and Motivation

Most sensor network operating systems like TinyOS, Contiki, SOS etc. are based on the event driven approach. Event driven approach is efficient in terms of time and space. Also this approach does not require a separate stack for each execution context. But using this model, it is difficult to implement long running tasks, like cryptographic operations.

In TinyOS, commands and events do very little work, and need to post a task to do long computations. The task runs to completion, and is not preempted by other tasks. Tasks are scheduled using FIFO scheduling policy. Since a task cannot be preempted by other tasks, at any point in time, there can be only one task running. Thus, computation intensive tasks like cryptographic operations will force other critical tasks to wait and such a situation is not desirable. This may force the application developers to split the long computation task into many small tasks, which is extremely difficult.

Mantis OS tries to solve this issue by designing the kernel based on a thread-based approach. A thread-based approach requires a separate stack for each execution context and is less efficient in terms of time and space. This approach increases memory consumption and energy consumption due to requirement of separate stack space for each thread, and context switch overhead. It may not be possible to support this overhead and such a situation is not desirable in many sensor applications.

TinyThread try to provide ease of programming in TinyOS by implementing a multi-threading library for TinyOS with co-operative scheduling and optional preemptive scheduling. A separate stack is needed for each thread, thus increasing the memory consumption of the sensor applications. Depending upon the availability and requirement of resources, the comparison between event driven and multithreaded sensor network operating system is given in [7].

Because of security problems in sensor networks, there is a requirement of performing long running computations like cryptographic operations. But at the same time, it is required that only a limited memory is used. It is also needed to ensure that the users should not worry about managing memory facilitating ease of programming. Lastly, energy consumption also needs to be minimized, and reliability of sensor nodes also has to be improved.

Multithreading can be implemented on Contiki reusing protothreads. But protothreads have several limitations. The implementation of protothreads using the C switch statements imposes a restriction on programs. Using protothreads, programs cannot utilize switch statements together

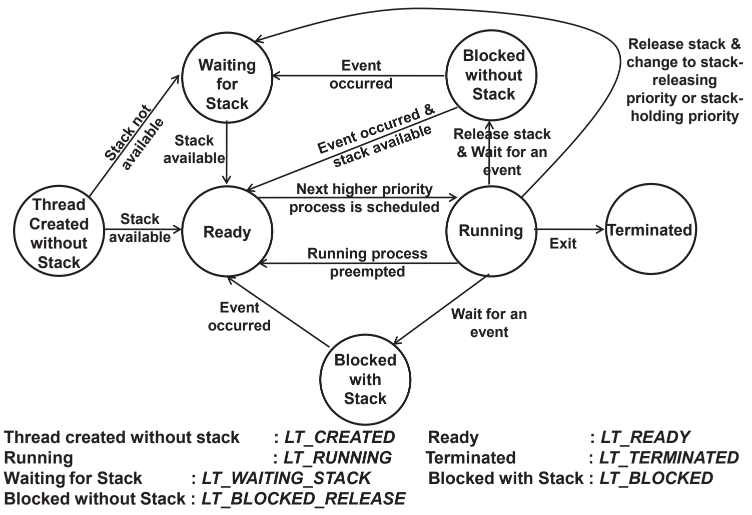
Waiting for Stack
Event occurred
Blocked without Stack
Release stack & change to stack-releasing priority or stack-holding priority
Stack not available
Stack available
Event occurred & stack available
Release stack & Wait for an event
Thread Created without Stack
Stack available
Ready
Next higher priority process is scheduled
Running
Exit
Terminated
Running process preempted
Event occurred
Wait for an event
Blocked with Stack
Thread created without stack : LT_CREATED
Ready : LT_READY
Running : LT_RUNNING
Terminated : LT_TERMINATED
Waiting for Stack : LT_WAITING_STACK
Blocked with Stack : LT_BLOCKED
Blocked without Stack : LT_BLOCKED_RELEASE

Figure 1. State transition diagram of the proposed OS

with protothreads. The Contiki provides multi-threading library to support preemption, but it does not assign priority to threads. It is necessary to have priority based preemption to provide real-time effect for some applications and also use a separate stack for each thread.

2 Proposed Model of OS and Its Implementation

Due to the presence of long running tasks, an OS should allow preemption. We also need a priority based scheduling to provide real time effect for applications. Yet, an OS should have very small footprint, leaving space for applications. We have chosen Contiki as our development platform, as it provides many features as mentioned in section 1.6. Keeping basic event driven model of Contiki kernel as it is, we embedded our thread based model for applications. In our model, the number of priority levels (MAX_LEVELS) (i.e., priority queues) are fixed . We have fixed the number of stacks for each priority level ($STACKS\_PER\_LEVEL$), and from this pool of stacks, a stack is allocated to a thread before the thread gets scheduled. Therefore, we will be having fixed number of threads in LT_READY state at any point of time.

Total stack space required by the system is bounded by treating it as a limited resource. Stack must be allocated to a thread before moving it to the LT_READY state. The stack will be released either explicitly by the thread and blocking for an event or by the system when the thread has exited. It is required that the threads in the same priority level either always hold the stack or always release the stack, while waiting for an event. If a priority level requires the threads to hold the stack while waiting for an event, then the priority is said to be a stack holding priority; otherwise, the priority is said to be a stack releasing priority.

When a thread is created ($LT\_CREATED$ *state*), the stack is not allocated to it. It will be moved from $LT\_CREATED$ state to LT_READY state only when stack is allocated for this thread; otherwise it will be moved to $LT\_WAITING\_STACK$ state as shown in Figure 1. While running, the thread can either choose to release the stack and wait for an event ($LT\_BLOCKED\_RELEASE$ state), or to hold the stack and wait for an event($LT\_BLOCKED$ state). If the thread is in $LT\_BLOCKED\_RELEASE$ state, and the event occurs, the thread is moved from $LT\_BLOCKED\_RELEASE$ state to $LT\_WAITING\_STACK$ state or to LT_READY state depending upon the availability of the stack. There is also an another possibility, when a thread wants to change its priority either to stack holding priority or to stack releasing priority, it directly moves from $LT\_RUNNING$ state to $LT\_WAITING\_STACK$ state. Also, when a thread is allocated to a stack, the location of stack may be different from that of the previous stack that was allocated to it earlier as we can have multiple stacks per priority level.

A typical application of release stack and wait for an event, can be a thread for encrypting the message. Once the encryption of the message is done, it can choose to release the stack, and wait for the next message to be encrypted. The advantage of having the option of releasing stack and wait is that we do not need to waste resources on the overhead of creating context again and again for periodic tasks.

Due to causal relationship between type of applications and priority, priority can either be assigned by the application programmer or by the Operating System.

In our implementation, we have completely separated protothreads and our thread implementation. But still protothreads can be used as higher priority threads using stack space of the event driven kernel stack

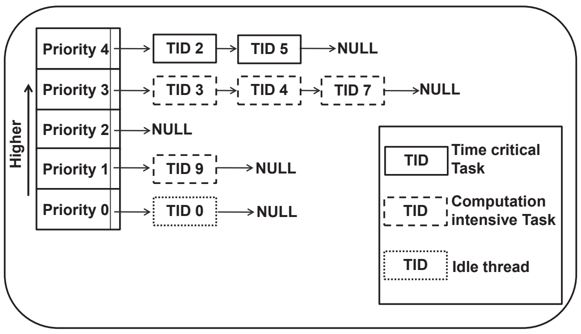

Figure 2. Snapshot of threads active in system for one stack per level

2.1 Scheduling Policy

Our proposed model uses the following scheduling policy.

- Fixed priority preemptive scheduling policy is used (thread can change priority).
- FIFO is used for threads having same priority.
- If a thread with stack releasing priority moves to a stack holding priority, it does not hold any resources.
- If a thread with stack holding priority moves to a stack releasing priority, it first releases the stack, but can continue to hold other resources.

The implication of the above scheduling policy is the following.

- A low priority thread is scheduled only if no higher priority thread is present in ready state.

- The currently running thread is preempted if another thread with priority strictly greater than the currently executing thread is moved to ready state.
- Once a thread is preempted, it is scheduled first before a new thread of same priority is scheduled.

Number of Priority Level as shown in Figure 2 are fixed but can be changed at compile-time. Priority Levels cannot be added or removed at run-time to reduce overhead of cleaning priority queues and managing priority conflict. The priority of a thread can be changed, but the thread has to release its stack, if it is holding one. Using FIFO scheduling policy among the threads with same priority, the number of preemption needed will be much lesser compared to that needed by a round robin scheduling policy. As the number of context switches decrease, the energy consumption will also decrease. Using this scheduling policy we can take maximum advantage of dynamic voltage scaling than using round robin.

There is no possibility of deadlock due to contention for stack in the above scheduling scheme. This is because a thread in stack holding priority level does not hold a resource while waiting for a stack. Therefore, no other thread will wait for it, while it waits for stack. A thread in stack releasing priority level can hold resources while waiting for stack, but it is always assured of getting a stack as other threads in the same priority level with ready or running state will continue to be executed until they release the stack.

$GetThreadToSchedule()$ returns the thread id of the thread to be scheduled. Thread 0 is the default thread for system, and it monitors the system. If there are no threads to be scheduled, thread 0 switches to power saving mode if supported by the microprocessor.

2.2 Stack Allocation

Fixed Number of stacks are associated for each priority level. Default value for $STACKS\_PER\_LEVEL$ is 1. It can be changed to have multiple stacks per level. The default size for a stack is (LT_STACK_SIZE) 256 bytes. Thus, total stack space for multi-threading environment is constant and calculated by $\{MAX\_LEVELS\}*\{LT\_STACK\_SIZE\}*\{STACKS\_PER\_LEVEL\}$. In case of stack holding priority level, the threads which are holding the stack can only be present either in the ready state, running state, or blocked state with holding stack. In case of stack releasing priority level, the threads which are holding the stack can only present either in the ready state or in the running state. So a stack can be allocated to a thread depending upon the total number of stacks assigned for that level and the number threads belongs to that level currently present either in the ready state, running state, or blocked state with holding stack. Multiple stacks per level can be utilized efficiently with fixed size stack.

2.3 Data Structures

Two main data structures are Level and TCB (Thread Control Block).

Level (*Level_t*) contains a pointer *List* to keep track of threads in that level and a *thread_stack* array containing stack(s).

typedef struct **Level** {

```
  lpid_t *List;
  unsigned int NumThreads;
  STACK thread_stack[STACKS_PER_LEVEL];
}Level_t;
```

TCB (Thread Control Block) contains information about thread id, priority, state, space for saving context of a thread, and a pointer to next TCB in the same priority level. The *tmp_SP* parameter is used for saving some extra context when a thread moves from running sate to blocked state after releasing the stack. TCB's *expire_time* (in milliseconds) element can be used to have real time effect for tasks. If the thread performing a task exceeds *expire_time*, then that thread is terminated.

```
typedef struct lt_process{
  lpid_t pid; // pid for thread
  unsigned int pri; // priority of thread;
  lt_thread_state_t state;
  unsigned int PC,SP;
  char tmp_SP[4];
  lpid_t next;
  int Stack_id;
  time_t expire_time;
}tcb_t;
```

2.4 Hardware Platform

To write an optimized and energy efficient software, one needs to understand how the underlying hardware works. We have studied the architecture of TelosB mote and MSP430 microcontroller. The brief description of TelosB and MSP430 are as follows.

TelosB : We have used Crossbow TelosB [3] mote for testing our implementation. It provides a platform for research and development work for wireless sensor network. It includes various peripherals like MSP430 microcontroller, 1MB external flash for data logging etc. It has sensor suite including integrated light, temperature and humidity sensor.

MSP430 : TI's MSP430 [1] has a 16-bit RISC CPU, its peripherals and flexible clock system are combined by using a Von-Neumann common memory address bus (MAB) and memory data bus (MDB).The MSP430 CPU has 16 16-bit registers,27 single-cycle instructions and seven addressing modes. All the registers can be used with all instructions. This CPU has 16-bit data and address buses, which minimizes power consuming fetches to memory. Fast vectored-interrupt structure used in CPU reduces the need for wasteful CPU software flag polling. Intelligent hardware peripheral features allow tasks to be completed more efficiently and independent of the CPU. MSP430 supports direct memory-to-memory transfers without intermediate register holding.

The addressable memory space is 64 KB with future expansion planned as shown in Figure 3. The starting address of Flash/ROM depends on the size of the Flash/ROM present and that

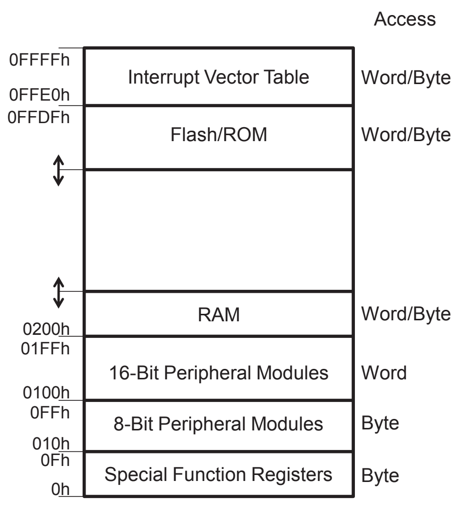

Figure 3. Address Space of MSP430

varies from device to device. The end address for Flash/ROM is 0FFFFh. Flash can be used for both code and data. The interrupt vector table is mapped into the upper 16 words of Flash/ROM. RAM starts at 0200h. The end address of RAM depends on the amount of RAM present and varies from device to device. RAM can be used for both code and data.

MSP430 has six operating modes, each with different power requirements. Three of these modes are important for battery-powered applications.
Active mode : CPU and other device functions run all the time.
Low power mode 3 (LPM3) : LPM3 is the most important mode for battery-powered applications. The CPU is disabled, but enabled peripherals stay active. The basic timer provides a precise time base. When enabled, interrupts restore the CPU, switch on MCLK, and start normal operation.
Low power mode 4 (LPM4) : LPM4 is used if the absolute lowest supply current is necessary or if no timing is needed or desired (no change of the RAM content is allowed). This is normally the case for storage preceding or following the calibration process. This mode is also called off mode.

This ultra low power CPU has up to 10 kB of RAM and up to 120 kB of programming flash.

3 Thread, Timer, and Semaphore APIs

3.1 Thread APIs

1. int **thread_create** (unsigned int *pri*, void (**start_routine*)(void), void **arg*) :

   thread_create is used to create a new thread with priority *pri* and state *LT_CREATED*. If $pri = -1$, then priority is chosen by OS. The thread is created by executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to *lt_exit*().

   If successful, the *thread_create*() function returns unique thread id *tid* of newly created thread. Otherwise, an error number is returned to indicate the error.

2. int **thread_setpri** (lpid_t *tid*, int *pri*) :
   thread_setpri is used to change priority of thread having thread id *tid*. If a thread is holding a stack then the stack is released and priority is changed.

   If successful, *thread_setpri* returns 0, otherwise, an error number is returned to indicate the error.

3. **thread_suspend_HS**() :
   thread_suspend_HS macro is used to suspend a thread by which the thread moves from running state(*LT_RUNNING*) to block state(*LT_BLOCKED*) with holding the stack.

4. **thread_suspend_RS**() :
   thread_suspend_RS macro is used to suspend a thread by which the thread moves from running state(*LT_RUNNING*) to block state(*LT_BLOCKED_RELEASE*) with releasing the stack.

5. lpid_t **thread_resume** (lpid_t *tid*) :
   thread_resume is used to resume the thread having thread id *tid* for execution by changing thread's state to *LT_READY*, if stack is available, otherwise to *LT_WAITING_STACK*.

   If successful, *thread_resume* returns *tid* , otherwise it returns -1.

6. lpid_t **getpid**(void) :
   getpid function returns the thread ID of the calling thread.

7. void **lt_exit**() :
   lt_exit terminates the currently running thread and frees all resources.

8. void **PrintLevel**() :
   PrintLevel prints status of all threads present in all levels.
   Output format of *PrintLevel* is *Level (number)* : *tid*(*lt_status*). This function is typically used for application's debugging purpose.

3.2 Timer APIs

1. int **tm_set**(struct tm **t*, clock_time_t *intrv*, void (**tm_routine*)(void), void **arg*, int *pri*, unsigned char *cpuoff*) :
   tm_set is used to create a new timer *t* which will fire after *intrv* time duration and create a new thread with priority *pri* which will execute the *tm_routine* with argument *arg*. If the *cpuoff* parameter is set to 1 then the CPU will off for *intrv* time duration, otherwise, CPU will remain in the current state.

   If successful, the *tm_set*() function returns 0, otherwise, an error number is returned to indicate the error.

2. void **tm_wait**(clock_time_t *intrv*):
   tm_wait is used to wait a thread for *intrv* duration.

3. int **tm_stop**(struct tm *t):
   tm_stop is used to stop the timer t before the expiration of its time duration.

   If successful, the *tm_stop*() function returns 0, otherwise, an error number is returned to indicate the error.

3.3 Semaphore APIs

1. semid_t **sem_create** (key_t *key* , int *count*, semflag $flag$) :
   sem_create creates a semaphore with value *count*, and returns semid. *key* is used for identifying the semaphore.

   $flag$ parameter indicates how semaphore needs to be created. It could be one of the following types:
   CREAT: Creates a new semaphore if the key does not exist.
   EXCL: If the key exists, it will cause the function to fail.

2. void **sem_post** (semid_t *id*) :
   sem_post unlocks the specified semaphore and calls the scheduler.

3. void **sem_wait** (semid_t *id*) :
   sem_wait waits for a semaphore. If the semaphore is unavailable, puts the current thread in waiting state and calls scheduler.

3.4 Memory Layout and Code Size

Memory Layout : Figure 4 shows the memory layout of Contiki operating system with our multi-threading interface. We implemented our model as part of Contiki kernel.

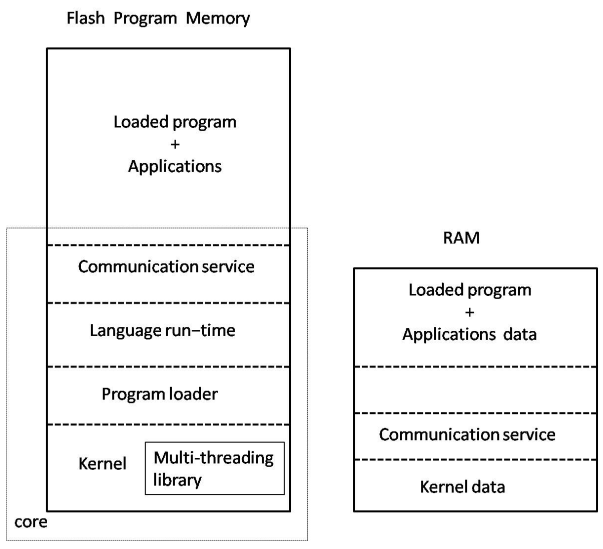

Figure 4. Memory layout

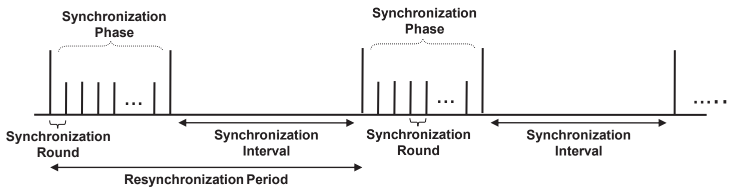

Figure 5. EFCS protocol schedule

Code Size : We tried to keep our code size as small as possible. Total lines of code is about 1.5K. including synchronization primitives. The proposed multi-threading model is compiled for Texas Instruments MSP430.

4 Implementation Results and Observations

The implementation of our model is ported to TelosB [3] mote. To test the implementation of our proposed OS, we have implemented a time synchronization protocol, call EFCS, as given in [17]. In the EFCS protocol schedule, there is a sequence of synchronization phase separated by synchronization intervals, and each synchronization phase consists of a fixed number of synchronization rounds as given in Figure 5. The abstract view of our implementation is as given below.

```
void sync_round(){
  cur_sync_round++;
  if(ur_sync_round < TOTAL_SYN_ROUND)
    tm_set(&myet_round, SYN_ROUND_DURATION, sync_round, NULL, 3, 0);
  else
    tm_set(&myet_interval, SYN_INTERVAL_DURATION, sync_interval, NULL, 4, 0);
}
void sync_interval(){
  cur_sync_round=0;
  tm_set(&myet_round, CLOCK_SECOND/2, sync_round, NULL, 3, 0);
}
void efcs_schedule(){
  sync_interval();
}
thread_create(2,efcs_schedule,NULL);
```

Thread $efcs\_schedul$ is scheduled by $thread0$. At the beginning, the thread $efcs\_schedule$ calls the *sync_interval* routine. In this *sync_interval*, the *tm_set* function creates a timer which will be fired after (CLOCK_SECOND/2) duration of time, and creates another thread which will execute the *sync_round* routine with priority 3. Here, this *sync_round* routine acts as a synchronization round as shown in Figure 5, and this routine will be repeated a fixed number of times depending upon the number of synchronization rounds present in a synchronization phase. For each synchronization round, the *sync_round* routine sets a timer which will create a new thread after SYN_ROUND_DURATION duration that calls the same *sync_round* routine. At the end of the synchronization phase, the *sync_round* routine creates another timer which will fired after SYN_INTERVAL_DURATION duration and creates a new thread which will call the *sync_interval* routine again. The same thing will be repeated in subsequent synchronization phases. We have made this experiment with 3 TelosB sensor nodes. The experimental result shows that the maximum error after each synchronization phase is 2 ticks.

Table 2 shows comparison between various sensor node operating system with our model. The features Low Power Mode, Priority-based Scheduling and Real-Time Guarantees are newly added (marked with ★) to Contiki. The Memory Management is platform dependent feature(marked with ⊛).

	TinyOS	SOS	Mantis	Our model
Low Power Mode	✓	✓	×	✓★
Dynamic Reprogramming	✓	✓	×	✓
Priority-based Scheduling	×	×	✓	✓★
Real-Time Guarantees	×	×	✓	✓★
Memory Management	×	×	×	✓⊛

Table 2. Comparison of sensor node operating systems

5 Conclusions

Our proposed model of OS utilizes stack space better than the protothread model used in Contiki. Using our model, we are able to implement multi-threading environment with limited stack size. It has a lesser requirement of RAM memory than Mantis OS. We also conclude that, the presented model has more generalized thread APIs (Like Unix/Linux) and synchronization APIs, which facilitates the ease of programming. With our model many applications can be ported on to Contiki OS with minimal changes.

Acknowledgment

This work is partially supported by the Defence Research and Development Organization (DRDO), Ministry of Defence, Government of India, under a research grant on wireless sensor network (DRDO 571, IISc).

References

[1] Msp430 data sheet. *http://pdf1.alldatasheet.com/datasheet-pdf/view/27250/TI/MSP430.html.*

[2] Nano-qplus os. *http://www.etri.re.kr.*

[3] Telosb data sheet. *http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.*

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[5] M. Britton, V. Shum, L. Sacks, and H. Haddadi. A biologically-inspired approach to designing wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 256 – 266, 31 2005.

[6] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157, New York, NY, USA, 2007. ACM.

[7] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan. An experimental comparison of event driven and multi-threaded sensor node operating systems. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pages 267–271, March 2007.

[8] S. Dulman, , S. Dulman, and P. Havinga. Operating system fundamentals for the eyes distributed sensor network. In *Proceedings of Progress 2002*, Utrecht, the Netherlands, October 2002.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov. 2004.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[11] L. Gu and J. A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14, New York, NY, USA, 2006. ACM Press.

[12] C. C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[13] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation os for embedded sensor networks. Technical report, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[14] P. A. Levis. Tinyos: An open operating system for wireless sensor networks (invited seminar). *Mobile Data Management, IEEE International Conference on*, 0:63, 2006.

[15] C. Nitta, R. Pandey, and Y. Ramin. Y-threads: Supporting concurrency in wireless sensor networks. In *Distributed Computing in Sensor Systems*, volume 4026 of *Lecture Notes in Computer Science*, pages 169–184. Springer Berlin / Heidelberg, 2006.

[16] A. M. V. Reddy, A. P. Kumar, D. Janakiram, and G. A. Kumar. Wireless sensor network operating systems: a survey. *International Journal of Sensor Networks*, 5(4):236–255(20), Aug 2009.

[17] A. Swain and R. Hansdah. An energy efficient and fault-tolerant clock synchronization protocol for wireless sensor networks. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 177 –186, Jan 2010.

[18] J. Yannakopoulos and A. Bilas. Cormos: a communication-oriented runtime system for sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 342 – 353, 31 2005.