Hybrid Real-time Operating System for Resource-constraint Wireless Sensor Nodes

[1,2]Xing Liu, [2*corresponding author] Kun Mean Hou, [2]Christophe de Vaulx, [1]Chengcheng Guo, [2]Hongling Shi, [2]Bin Tian
[1]Laboratory of Internet and Information Technology, Wuhan University, Wuhan, China
Email: netgcc@whu.edu.cn
[2]LIMOS Laboratory UMR 6158 CNRS, Blaise Pascal University, Clermont-Ferrand, France
Email: {liu, kun-mean.hou, devaulx, shi}@isima.fr

***Abstract*—Wireless sensor network (WSN) has been used in widespread domains, and the real-time response is required by many WSN applications. However, due to the memory resources limitation on the sensor nodes, the current WSN OSs such as TinyOS, Contiki, SOS, mantisOS, etc., are not real-time ones. To achieve the objective of designing a real-time OS with low memory resource consumption, a new WSN OS named HEROS is developed and presented in this paper. For HEROS, it adopts a hybrid scheduling strategy. Both the event-driven and multithreading schedulers are implemented in parallel, and these two schedulers can switch to each other when necessary. By this means, HEROS take advantages of both the event-driven system's low memory resource consumption as well as multithreading system's high real-time performance. Besides these, HEROS uncouples the applications from the underlying systems by using the pre-linked mechanism (PLM). With this mechanism, a user-friendly development environment can be provided to the WSN users. Finally, to evaluate the performance of HEROS, it is compared with some other WSN OSs on the iLive platform (8-bit AVR microcontroller). The final experimental and evaluation results prove that HEROS is a memory resources efficient, real-time supported and user-friendly OS, and can be used on most resource-constrained sensor nodes to support the diverse kinds of WSN applications.**

***Index Terms*— operating system, real-time , hybrid, wireless sensor network**

I. Introduction and Backgrouand

Wireless sensor nodes (WSN) have been used in widespread domains ranging from the precise agriculture to the military surveillance [1-4]. And for the software development on sensor nodes, the operating system (OS) is one of the key technologies. This is because an outstanding OS can not only manage the platform resources well, but also provide good services for the WSN applications.

Currently, several challenges exist for the WSN OS development. Firstly, the WSN platform resources are constrained [5]. Most sensor nodes are small size and low price ones with limited memory resources, e.g., the sensor nodes equipped with the AVR ATmega1281 microcontroller has only 8 kilobytes RAM. Therefore, a good WSN OS should have low memory resources consumption. Secondly, the real-time response is required by many WSN applications, such as the engine control process in the industrial system, the heart pacemakers monitoring in the medical system, etc. However, the current popular WSN OSs, such as TinyOS, Contiki, etc., cannot support the real-time reaction well. Thirdly, the WSN application development process is complicated for the users since the underlying WSN hardware and software platforms are diverse [6].

Up to now, several WSN OSs have been developed, such as TinyOS [7], Contiki [8], SOS [9], MantisOS [10], BitCloud OS [11], LIMOS [12], uCOS [13], AVRX [14], etc. However, these OSs cannot address the above challenges well. On one hand, most of these OS [7-9, 11, 12] are the event-driven scheduling OSs. For these OSs, the advantage is the memory resource consumption is low, whereas the real-time performance is poor as the preemption cannot be supported. For the other OSs such as [10, 13, 14], the multithreading scheduling model is used. With the multithreading scheduler, the preemption can be achieved by the thread switch. Consequently, these OSs have the real-time performance better than the event-driven ones. Nevertheless, the memory resource consumption of these OSs is relatively high as each thread needs to have its own run-time stacks. Thus, how to achieve an OS which has good real-time performance as well as consumes less memory resources become essential for the current WSN OSs. On the other hand, the monolithic system architecture is used in many current WSN OSs [10-13]. For these OSs, the applications are not uncoupled from the systems, thus it is difficult for the WSN users to develop the applications as they are required to understand the low-level system details. For [7-9], the applications can be separated from the systems either by the virtual machines (VMs) [7, 15, 16, 17] or by the dynamic linking mechanisms (DLM) [18, 8, 9]. Yet, they are still not sound. Because for [7], the applications should be programmed by using the non-popular byte code instructions directly. And for [8, 9], the memory resource consumption of the DLM is high and the code loading process in DLM is also complicated. Due to these reasons, the design and implementation of a new WSN OS becomes essential.

doi:10.4304/jsw.9.7.1767-1780

In this paper, a hybrid real-time OS HEROS is developed to address the challenges above. The key features of HEROS include the following aspects: Firstly, it adopts a hybrid scheduling mechanism. Both the event-driven and multithreading schedulers are implemented. Event-driven is used to dispatch the non real-time events while the multithreading scheduler is used to dispatch the real-time events. Secondly, it implements a dynamic memory allocator which avoids some drawbacks of the allocator in the current WSN OSs. By means of the dynamic memory allocator as well as the hybrid scheduler, HEROS can react to the time-critical events with low memory resources consumption. Thirdly, the applications in HEROS are uncoupled from the systems by using the pre-linked mechanism (PLM). This mechanism has been proved to be resources efficient. With it, the user application development process can be simplified.

The main structure of this paper is as follows: In section IV, the hybrid HEROS scheduler is presented. In section V, two kinds of HEROS dynamic memory allocators are discussed. In section VI, the implementation of the software timers is introduced. In section VII, the HEROS PLM is designed and implemented. From section VIII to XI, a new OS debug method, the related work, the performance evaluation, the conclusion and ongoing work are presented respectively.

II. Background of WSN OSs

In terms of the scheduling policy, the WSN OSs can be classified into two types: the event-driven OSs and the multithreading OSs. In the past research work, there exist some debates and discussions about these two types of OSs [19-23].

A. Event-driven WSN OSs

SOS and Bitcloud OS are all pure event-driven OSs. In these OSs, a set of event handlers are defined, each handler is related to an event. Once an event is triggered, the related handler will be invoked. Each handler runs to completion with respect to each other. The interruption is enabled during a handler's executing process, but the preemption from one handler to another is not allowed. Since all the handlers are executed one by one, only one stack is needed and be shared by all the handlers. Thus, the memory consumption of event-driven system is low. However, the real-time performance cannot be well supported, e.g., after a time-critical event is triggered, it cannot be executed immediately by preempting the current executing handler, even if the current executing handler is not a time-critical one.

B. Multithreading WSN OSs

The OSs such as MantisOS, uCOS are multithreading ones. In multithreading system, each handler is executed by one thread. All these threads do not execute one by one like event-driven systems do, but run concurrently by the thread switch. Since the thread switch exists, each thread needs to have a private thread stack which will be used for the storing of the thread's run-time context. By means of the thread switch, the real-time performance in multithreading systems can be better than that in the event-driven systems. However, the memory resource consumption is also higher if compared with that in the event-driven one [19]. In Table I, a comparison between the event-driven and the multi-threading OS is shown.

Besides the pure event-driven and multithreading OSs, some current OSs have implemented both the event-driven and multithreading schedulers in the system, such as the TinyOS [7], Contiki [8] and LIMOS [12]. However, the multithreading scheduler in these OSs is implemented as an optional library upon the event-driven scheduler (presented in detail in the related work in the section XI). Therefore, these OSs [7,8,12] are not native hybrid system, but still event-driven system in the native scheduling layer.

TABLE I.
COMPARISON BETWEEN EVENT-DRIVEN OS AND MULTITHREADING OS

Features	Event-driven OS	Multithreading OS
Scheduling Manner	All event handlers execute one by one	All handlers run concurrently
Preemption	Not enable	Supported, thus the overhead of stack switch exists
Real-time response	Cannot be supported well	Can be well supported by thread preemption
Stacks	All handlers share one global stack	Each thread should have its own private stack
Computation Resources	Shared among all handlers in a cooperative way	Divided among all threads by thread switch

III. System Architecture of HEROS

The basic terms in HEROS include the event, event handler and the system process.

Event: Event is a system signal which indicates that the condition to take some system action has been satisfied. An event can be generated by a key pressing, a wireless packet reception, an expired system timer, etc. In HEROS, events are classified into three types in terms of their emergence: the common events, the hard real-time events and the soft real-time ones.

Common events are the ones which don't have a strict requirement to the response time, e.g., in the garden caring applications, when the humidity value of the soil decreases to a given level, an event should be triggered to request the operation of opening the hydro valve to water the flowers. In this case, this event is a common one because the response delay of even tens of seconds to it can still be accepted.

Hard real-time (HRT) events are the imperative ones which should be responded within a strict deadline, if not, a great disaster can be caused. They can be generated in the systems such as the car engine control, the human life medical care, etc. These events are rarely generated and are triggered particularly in some emergent situations.

Soft real-time (SRT) events are the ones that should be reacted immediately, but the response constraint time is not so strict that a short time delay is still allowed, e.g., when a wireless packet is received, an event will be generated to request the operations of processing this

packet as well as sending back an acknowledgement (ACK) packet to the sender. For this event, it can be deferred for 1 or 2 seconds in case that the sender's retransmission operation is still not triggered. And this event is a SRT one.

Event handler: Event handlers are a series of system subroutines or methods. Each event handler is bound to at least one event. Once an event is triggered, the handler corresponding to it will be invoked.

Process: Several event handlers that are logically connected or belong to an identical component can build up together to form a process. A process can be regarded as a wrapper of a set of handlers, it is proposed as it can make the system organization to be clear as well as avoid too many small pieces of handlers defined in the system.

Fig. 1 describes the event data structure in HEROS and also its relationship with the system handlers and process. Inside the event structure, a pointer member named *process* is used to indicate the process which this event will be posted to, and another member *handler_id* is used to point out which event handler in this process will be invoked by this event. Moreover, the pointers *event_data* and *data_size* are used to point out respectively the event data's address and the data's length.

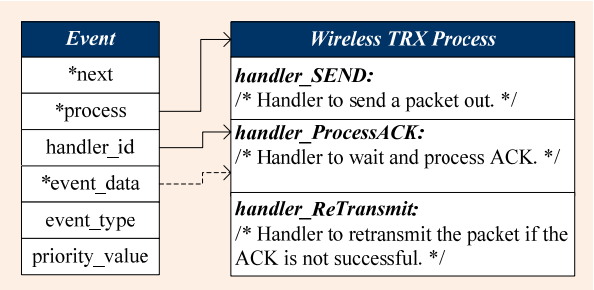


Figure 1. Relationship among event, event handler and process.

IV. HYBRID SCHEDULER IN HEROS

In HEROS, a hybrid scheduling policy is adopted. The event-driven scheduler is used to dispatch the common and SRT events while the multithreading scheduler is used to dispatch the HRT events. At any time, only one kind of scheduler is active. If only the common and SRT events exist in the system, the OS runs in the pure event-driven model, in this case, all the common and SRT event handlers are executed one by one. If one or more HRT events are triggered, the execution of the current common or SRT event handler will be suspended immediately, and then the OS will switch into the multithreading model. In this model, one thread will be created for each HRT event, and the *run* method of each thread is to execute the event handler related to this HRT event. The run-time context of each thread is created dynamically. And after all the HRT handlers run to completion, the related thread contexts will be released, then the OS will switch back to the event-driven model to continue the dispatching of the common and SRT events.

Since HRT events are generated only in some typical cases, HEROS runs in pure event-driven model most of the times and this is the default scheduling model of HEROS. Thus, the memory resource consumption of HEROS is low. However, due to the scheduler switch mechanism, the event-driven scheduler in HEROS can switch to the multithreading one when required. Therefore, the real-time reaction to the time-critical events can also be achieved. And when HEROS runs in the multithreading model, the thread run-time stacks are allocated in a dynamical way. Consequently, HEROS can achieve the objective of being an OS which supports the real-time response on the resource-constraint WSN platforms.

In this section, the design and implementation of the event-driven and multithreading schedulers in HEROS are presented in the part A and B respectively, and the scheduler switch strategy is discussed in part C.

A. Event-driven Scheduler in HEROS

Due to the event-driven scheduler's features, several topics should be considered for the implementation of the event-driven scheduler:

Events buffering mechanism: As the events are dispatched one by one, the event dispatcher may not be able to handle all the events as quickly as they arrive, thus an event buffering system is needed to buffer the upcoming events. Seen in the Fig. 2, after the events are generated, they will be posted into the event queue, and then be extracted and dispatched one by one by the event dispatcher. After an event is dispatched, the related event handler will be invoked.

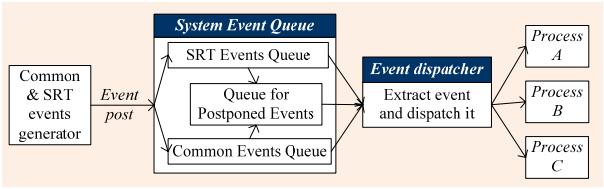


Figure 2. Structure of event-driven scheduler in HEROS.

In HEROS, three kinds of event queues are defined: the common event queue, the SRT event queue and the postponed event queue.

Common event queue is used to buffer the common events. Since the response time of the common events is not so constrained, the simple FIFO (First Input, First Output) algorithm is used to manage this queue.

SRT event queue is used to buffer the SRT events, the SRT events are inserted into this queue in the sequence of their event priorities. The priority value of a SRT event is pre-defined in terms of this event's emergency. The more emergent an event is, the smaller its priority value will be. And in order to prevent a low priority event becomes stale by the keep coming high priority events, a dynamic priority mechanism is adopted for the SRT events. With this mechanism, the priority values of all the SRT events will decrease by a given value every time the system PIT (periodical interruption timer) is fired. By this means, when a SRT event will be extracted is not only dependent on the original priority, but also on the time that it has been pending in the SRT event queue.

For the postponed event queue, it is used to buffer the common or SRT events that cannot be processed for the moment, e.g., an event is dispatched to send a packet out from the wireless media, if the wireless media is detected to be busy, this event can be transferred to the postponed event queue. In Fig. 3, an example is shown about the event initialization and posting process in HEROS.

```
/* When an ACK packet arrives,
   generate an event and post it into event queue. */
/* fill an event. */
event->process = (sys_process_t)wirelessTRXprocess;
event->handler_id = ACK_ARRIVAL_HANDLER;  /* handler to be invoked */
event->event_data = TRX_rcvpkt_buffer;      /* Data buffer related to event. */
event->data_size = 16;        /* payload of ACK packet is 16 bytes */
event->event_type = SRT;    /* this event will be inserted into the SRT event queue. */
event->pri_value = 3000;     /* event priority is assigned as 3000 (3 seconds). */

/* post this event. */
event_post(event);    /* post this event into the event queue. */
```

Figure 3. Event initialization and post process in HEROS.

Events dispatcher: Event dispatcher runs in a loop, it extracts the events one by one from the event queue, and then invokes the corresponding event handler. In case that no events exist in the event queue, the sensor node will fall asleep.

For the extraction sequence, the events in the postponed event queue are precedent over the other two queues. Then it is the SRT event queue. And only when the postponed and SRT event queues are empty, the common event queue will be traversed.

Achievement of OS concurrency: In event-driven OS, the handlers are not preempted. Every event handler should run to completion before the next one can be invoked. Thus, how to schedule all the handlers concurrently becomes a challenge. And to address this challenge, two problems should be solved. One is the execution time of each handler should not be too long, otherwise, the other time-critical events can become stale. The other is that the execution of the event handlers should not be blocked, if not, the system run-time process will be hung up during the blocked time.

In Contiki, these problems are solved by the proto-threads [24]. With the protothreads, a blocking run-time context without the overhead of multithreading stacks can be implemented in the event-driven system, and this is achieved by defining a local continuations (LC) variable in each process. By means of this variable, if a handler needs to yield the control or be blocked, it can save the current executing address into the LC variable. And then, next time this handler is called again, it will not be executed from the beginning, instead, the LC value will be loaded and the execution resumes from the recorded LC address.

For HEROS, the protothread method can also be used to solve the resource sharing and blocking problems. However, due to its special system architecture, the handler phase-split mechanism is more suitable to be used. With this mechanism, if a handler's execution time is too long, this handler will be split into more pieces. Likewise, if a handler will be blocked at a given point, it can also be split into two from the blocking address. By these means, the problems of blocking and long-time handlers can be avoided.

Compared with the Contiki protothread, the advantage of HEROS handler split method is that it doesn't need to define the global LC variables for all the handlers, but it shows the drawback in the structure control, e.g., if the *while* loop is needed to be used between two split handlers, the *goto* statement should be used, and this may increase the handlers' programming complexity.

Event-driven scheduling workflow: To understand how the event-driven scheduler works in HEROS, an example is shown in Fig. 4. In this figure, H_i *(i=1,2,3,…)* represents the system handlers, Es_i *(i=1,2,3,..)* represents the SRT events, Ec_i *(i=1,2,3,..)* represents the common events. After event Es_1 is dispatched, the related event handler H_1 will be invoked. During the execution of H_1, another event Es_6 will be posted, and Es_6 will lead the handler H_4 to be called, etc. Assumed the priority of Es_m is higher than Es_n in case that m is smaller than n. Assumed the equation of *nextEventExtract*(E_1, E_2, E_3, ...) returns the next event to be extracted from the events (E_1, E_2, E_3, ...). And to simplify the explanation process, assumed the priorities of all SRT events are static other than dynamic updating. Then, the event-driven scheduling workflow will be as follows:

Firstly, the dispatcher extracts one event from the event queue. In result, *nextEventExtract*(Es_1, Ec_1, Es_3) which is equal to Es_1 is returned because its priority is higher than Es_3 and Ec_1. After Es_1 is dispatched, H_1 will be invoked which generates a new event Es_6. Later, when H_1 runs to completion, the dispatcher will extract the next event *nextEventExtract*(Es_6, Ec_1, Es_3), and this time Es_3 will be extracted. For the next step, handler H_3 will be invoked, etc. Finally, the handlers' scheduling sequence will be as:

$$\{H_1 \text{-> } H_3 \text{ -> } H_4 \text{ -> } H_2 \text{ -> } H_6 \text{ -> } H_5 \text{ -> } H_7\}. \quad (1)$$

From this workflow, it can be seen that event-driven OS is a cooperative scheduling system in which all the event handlers cooperate with each other to share the computation resources.

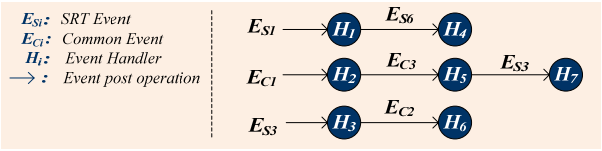


Figure 4. Event-driven scheduling workflow in HEROS.

B. Multithreading Scheduling Policy

For the HRT events, once they are generated, they should be handled as soon as possible. In HEROS, this is achieved by the multithreading scheduler. For the design and implementation of a multithreading system, the topics as follows should be discussed:

Thread control block (TCB) and thread stack: Once a thread is created, a TCB as well as a thread stack will be created. This TCB is used to record some key information about this thread, such as the related HRT event, the thread status, the thread stack pointer as well as the synchronization queue pointer, etc.. And for the thread

stack, it will be used both for the thread execution and the thread switch context saving.

Thread scheduling and synchronization: If the HRT events are generated in the system, the HRT threads will be created correspondingly. All the HRT threads will execute concurrently by the thread switch, and the static-priority RMS (rate-monotonic scheduling) algorithm is used for the thread scheduling. For the thread synchronization in HEROS, the semaphore mechanism is used.

Multithreading scheduling workflow: Different from the event-driven OS in which the handlers are executed one by one, in the multithreading OS, the HRT event handlers run concurrently by the thread switch. Fig. 5 shows an example about the multithreading scheduling workflow in HEROS. Assumed that the executing time of all the HRT handlers is identical, and then the handler execution sequence will be as follows:

$$\{H_1 \text{-> } H_2 \text{ -> } H_3 \text{ -> } H_4 \text{ -> } H_5 \text{ -> } H_6 \text{ -> } H_7\}. \quad (2)$$

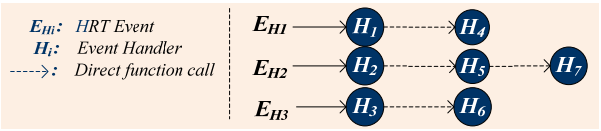


Figure 5. Multithreading scheduling program in HEROS.

C. Scheduler Switch in HEROS

As both the event-driven and multithreading schedulers are implemented in HEROS, how to switch from one scheduler to another is an important topic. In order to make the switch process to be more efficient and easy-to-managed, the event-driven scheduling process in HEROS is also considered as a thread "*main_thread*", and this thread's *run* function is to extract and dispatch the events one by one from the common/SRT/postponed event queues, seen in the Fig. 6.

Therefore, the hybrid scheduling process in HEROS is as follows:

- If there are no HRT events generated in the system, only one thread *main_thread* exists. In this case, the system runs in the pure event-driven model, and this is HEROS's default scheduling model.
- Once the HRT events are generated, the *main_thread* will be suspended. Then, the OS switches to multithreading scheduling model.
- If all the HRT threads run to completion or are inactive, the *main_thread* will be resumed and the common/SRT/postponed events will be dispatched again. In this case, the OS becomes an event-driven system once more.
- When a HRT thread is created, all the required resources will be dynamically allocated. After it runs to completion, these resources will be freed.

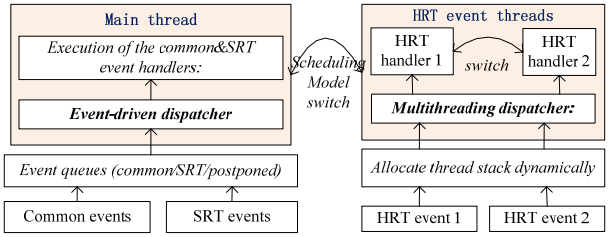


Figure 6. Hybrid scheduler model in HEROS.

V. Memory Allocation

Since the RAM resources on the sensor nodes are precious, the memory allocation becomes a key issue for the WSN OS.

In Contiki, the memory area is statically reserved, but dynamically allocated in block. Seen in the Fig. 7, a set of block areas are pre-reserved for the different kinds of system objects. Each block area is composed of a few blocks. The number of the blocks in each block area is pre-defined, and the block flag mechanisms are used for the block management. The advantage of this allocation method is that it is simple to be implemented. However, since the memory areas are statically pre-reserved, two drawbacks exist. Firstly, the memory resources in one block area cannot be shared with the other areas. Secondly, the number of the blocks in each block area is difficult to be decided, because in different applicable environments the required numbers can be different. And in the Contiki system, in case that the pre-reversed block number is not enough, an error will return. A common method to solve this problem is to pre-reserve each block area as large as it can be required, but the memory resource consumption will increase greatly in this case.

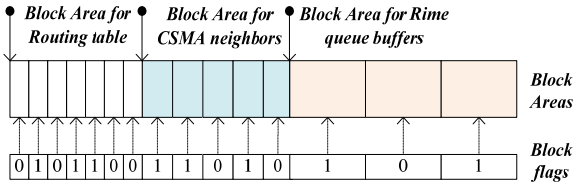


Figure 7. Memory blocks allocation in Contiki.

In SOS, the allocation mechanism is similar with that in Contiki. What is different is that the link queue other than the block flags is used for the allocation management. Compared with the flag mechanism, the queue management is more complicated to be implemented, but the free block can be quickly acquired by deleting directly from the free queue head.

For mantisOS, the allocation is different from Contiki and SOS. It doesn't use the block allocation. Instead, all different kinds of objects are allocated in the same heap. This method shows the advantages that no memory areas are needed to be pre-reserved. However, the memory fragments can appear after a few pairs of allocation and free operations, seen in the Fig. 8. Currently in mantisOS, no mechanism is implemented to solve this memory

fragment problem, and this will decrease the utilization efficiency of the memory resources.

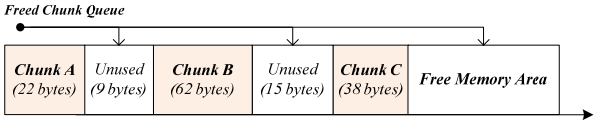


Figure 8. Memory allocation in mantisOS.

From the memory allocation mechanisms in the above OSs, it can be seen that there are mainly two challenges for the allocation of the memory resources in embedded system. The first results from the pre-reservation problem, and the second results from the memory fragments.

In HEROS, to avoid the above drawbacks, a dynamic allocation mechanism which supports fragment collection is implemented.

Seen in the Fig. 9, the memory heap in HEROS is divided into three parts: the object allocation area, the virgin area and the reference area. Every time a new object is required to be allocated, it will firstly be allocated from the freed object queue. If not success, then the allocation will be done from the virgin area. And in order to clean up the memory fragments, the fragment assembling mechanism is used. E.g., in the Fig. 9 (a), there are two freed members in the freed object queue, if a new 60-byte size object is required to be allocated, it cannot be processed successfully as there is no continuous memory space of which the size is as large as 60 bytes. However, the total size of the virgin area plus the two memory fragments is larger than 60 bytes, thus this allocation can be done successfully in case that all these freed memory resources are assembled, seen in Fig. 9 (b). And to assemble these memory resources, the chunks of object B&C need to be shifted. However, after the shifting of these chunks, they can no more be accessed correctly by the others as their addresses have been changed. Therefore, to address this challenge, the indirect memory access mechanism is used in the HEROS memory allocator. With this mechanism, every time a new object is allocated, a corresponding reference pointer will be created and point to it, and the access to the allocated objects will be done indirectly through these reference pointers. By this way, if the allocated objects' addresses changed after the fragments are assembled, the value updating to these reference pointers can keep the access to these objects still be available. Consequently, the memory fragments can be collected in the HEROS memory allocator, and the memory resources can be utilized efficiently in this way.

VI. Software Timer

Software timer is a counter which will request the system to take some actions when the counter value meets a preset condition. Commonly, the software timers are based on the hardware PIT.

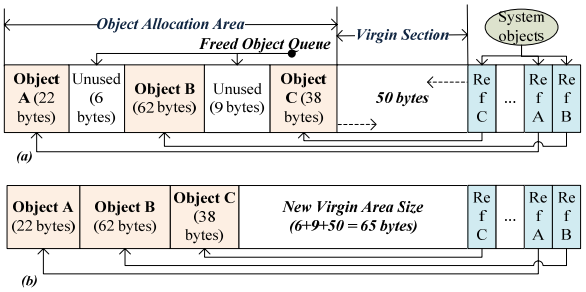


Figure 9. HEROS allocator with fragment assembling support.

In HERSO, the timers can be classified into two types: the one-slot timer (*OsT*) or the periodic timer (*PrT*). For the one-slot timer, it will be deleted once it is expired. For the periodic timer, it will be reset and restarted after its expiration.

And the timers can also be classified into two modes: real-time reaction (*RtR*) mode and asynchronous reaction (*AcR*) mode. For the former mode, once the timer is fired, a related callback function will be called directly. For the later mode, after the timer is fired, an event will be posted into the system event queue, and only when the event is dispatched, the related event handler can be executed.

As for the implementation of the timers, HEROS implements in two ways, one uses the relative counter value (*RCV*) and the other uses the absolute counter value (*ACV*). And a system timer queue needs to be created to link all the started timers. Every time the PIT is interrupted, this timer queue will be checked to see whether some timers that have been fired. If a timer is observed to be fired, the related timer handler will be processed.

VII. User Application Development

As the application program usually requires the OS services to perform the function, an excellent OS should provide a friendly application development environment to the users.

To develop the applications on sensor nodes, programming and reprogramming are two important processes. And one way to simplify these processes is to uncouple the applications from the low-level systems, by this way, the whole software system will be divided into both the user application space and the system space, and two executable images will be generated independently, seen in the Fig. 10 and 11.

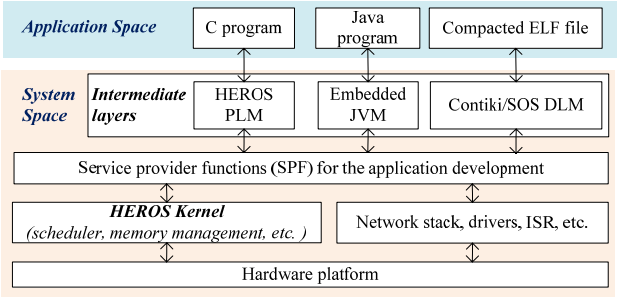


Figure 10. Architectures to uncouple the applications from systems.

After this separation, for the persons who are WSN experts, they can in charge of the system space and pre-burn the system image onto the sensor node. Then, for the WSN users, they only need to focus on the application space without the necessity of considering the lower system details.

Currently, several mechanisms have been developed to uncouple the applications and the systems. One way is to use the virtual machine (VM), and this method has been used in TinyOS. For TinyOS, several VMs have been implemented such as Maté [15], Bombilla [16], SwissQM [17], etc. However, the users are required to program the applications by the VM byte code instructions directly, and only a limited instruction set is provided. To address this challenge, many embedded Java VMs (EJVMs) are developed to be used to the resource-constrained devices, such as FijiVM [25], JamaicaVM [26], Squawk JVM [27], Darjeeling VM [28], simpleRTJ [29], NanoVM [30], etc. With the EJVMs, the users can program the applications by the popular, robust, object-oriented and hardware independent Java language. Yet, these JVMs either have non-trivial memory resource consumption [25-29], or support limited VM features [30]. Moreover, the byte code execution efficiency is low, thus more energy resources will be consumed on the node. Therefore, for the high resource constrained sensor nodes, the EJVMs are not a suitable choice. Besides the EJVM, the dynamic loadable module (DLM) is another way, with this method the WSN application are built into a loadable module and then uploaded to the sensor nodes to be linked dynamically before being executed. This method has been used in the OSs such as Contiki and SOS [18]. In these OSs, the applications are built into an optimized ELF (Executable and Linkable Format) file, and on the sensor devices, the function and variable references inside this file will be resolved and linked in a dynamical way. Compared with the JVM mechanism, the drawback of DLM is that the applications need to be programmed by the C language which is not so robust and user-friendly as Java. Moreover, the DLM module needs to be linked before being executed. However, after the DLM module is resolved and linked, the final executive code is the pure machine code with a high execution efficiency.

In HEROS, in order to provide a simple and efficient method to separate the applications from the systems, the pre-linked mechanism (PLM) is adopted. And there are several reasons for this: *1).* the size of the PLM application image will be smaller as no resolving or interpretation data is needed to be contained inside, and this will improve the energy consumption as well as the code transmission success probability in the application reprogramming process. *2).* most PLM work is done on the PC other than the sensor device, thus the software architecture on the sensor nodes can be simplified. *3).* since the PLM image uses the machine code which can be executed directly without resolving or interpretation, its execution efficiency is high.

However, several problems need to be solved for the PLM method, such as the code flexibility, the parameters passing between the PLM code and the system image, etc.

A. Implementation of HEROS PLM

To implement the HEROS PLM, two kinds of functions need to be defined: the service provider function (SPF) in the system space which provides the system services for the application development, and the service subscribe interface (SSI) in the application space which can be used by the users to access to the system services. For each SSI, a corresponding SPF will be programmed, and the call of the SSI from the application image will cause the related SPF in the system image to be executed.

Since the application image and the system image are built independently (Fig. 11), HEROS PLM should function as a bridge between these two images to make that they can interact with each other well:

Link of the SSI in the application image to the corresponding SPF in the system image: By PLM, the application image is build independently from the system one, yet it needs to access to the system services. Thus, how to pre-link an application SSI to the related system SPF is the first problem to be solved.

A common way to solve this problem is to build the system project firstly and generate a map file in which all the system functions' addresses are listed. Then, the pre-linking of the application SSI functions can be done in terms of this list. The drawback of this method is that the application image is inflexible as any change to the system image can cause the application image to become unavailable.

In order to make the application image be flexible, the application SSIs in HEROS PLM are not linked directly to the system SPFs. Instead, an intermediate function jump table is provided in the system project, with this table, the SSIs are linked indirectly to a given address in this table, and then this table will redirect the function call to the corresponding SPFs. By this means, the change to the system image will not affect the validity of the application image in case that the jump table is put at a fixed address and the item order in it is not changed.

Fig. 11 depicts the application image development process after the HEROS PLM is implemented. With the information acquired from the object and ELF files, the raw application image is modified once more to link all the SSIs to the system function jump tables. All these works are done on the PC other than the WSN devices.

For system image, it is pre-burned on the sensor node. After this, if the node application needs to be changed, only the application image is needed to be updated.

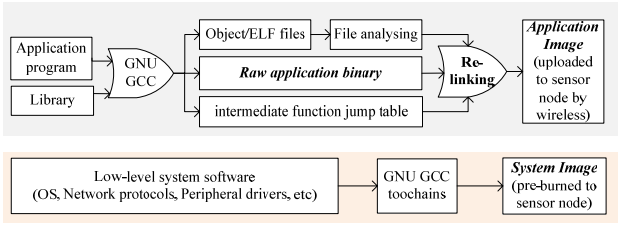


Figure 11. HEROS PLM development process

Parameters' passing between application SSI and system SPF: Besides the pre-linking of an SSI to the

relevant SPF, another problem to be solved is how to pass the function parameter between SSI and SPF.

Assumed HEROS PLM is used on the AVR microcontroller platform. Then, after the application program is built, it can be observed from the AVR assembler code that before the "*CALL*" directive of a SSI is invoked, the function input parameters will be put into the registers gradually from *[R25, R24, ...]*. And in the SPF's assembler code, the function parameters will be extracted one by one from *[R25, R24, ...]* as well. Thus, there is no parameter passing gap between the SSIs and the SPFs, this is because the same compiler is used for both the application and system images, therefore, the same operation rule is followed even if these images are not built in different projects.

For the return parameter from the SPF to the SSI, it is the same case as the input parameters, And the registers *[R25, R24]* is always used to pass the return values.

Callback from the System Image to Application Image: Once the function linking and parameter passing from the SSIs to SPFs are achieved, the application image can access the services in the system image. Yet, the reverse callback operation is not supported.

However, this callback functionality is required in many cases, e.g., the hardware interruption routine service (ISR) is programmed in the lower system image. If an ISR is expected to be programmed by the users in the application image, then there should be a callback operation which redirects the execution from the system ISR to the given application ISR. In HEROS PLM, the callback is achieved by a registration mechanism. If an application function needs to be called back from the system image, it should be registered by a registration interface. After registered, this application function's address will be passed into the system space, and then the call back from the system image to this function can be achieved.

With the mechanisms above, both the application image and the system image can interact well with each other in despite that they are built separately.

B. Performance Evaluation of HEROS PLM

To evaluate the performance of HEROS PLM, it is compared with the Contiki DLM and the simpleRTJ EJVM mechanisms. SimpleRTJ is chosen because it is a clean room implementation JVM for the small embedded devices.

Features and memory resource consumptions: The features and memory resource comparison results are shown in Table II. For simpleRTJ, it provides a good Java development environment. However, the memory resource consumption is high. For Contiki DLM, it consumes more memory resources than HEROS PLM, this is because it uses the dynamic linking mechanism.

Application Image Size: It is significant and economic to reprogram a new WSN application to the sensor node to adjust its functionality. And for the reprogramming process, the application image size is a key factor. This is because it will not only determine the success probability of transmitting this image integrally, but also influence the energy consumption on the sensor nodes [31].

TABLE II.
COMPARISON RESULTS OF FEATURES AND MEMORY CONSUMPTION

Comparison Titles	Mechanisms		
	Contiki DLM	HEROS PLM	EJVM simpleRTJ
Application programming language	C	C	Java
Flexibility of application image	Well	Average	Well
Call from application image to system image	Supported	Supported	Supported
Callback from System to Application	Not support	Supported by registration mechanism	Supported by Java thread interface
Exception handle support	No	No	Yes, more robust
Garbage collection	N/A	N/A	Yes
Required ROM (KB)	5.7	1	18-24
Required RAM (bytes)	18	15	200

For the comparison of different mechanisms, a basic sensor sampling application example is implemented in both C and Java language, with the results shown in the Table III.

TABLE III.
IMAGE SIZE COMPARISON RESULTS OF DIFFERENT MECHANISMS

Mechanism / Application image format	Code size (bytes)
Monolithic system in which applications are not uncouple from systems	114786
Contiki DLM / Compacted ELF	769
EJVM simpleRTJ / Java byte code	2472
HEROS PLM / Pre-linked machine code	182

From the results, it can be seen that HEROS PLM has the minimum size application image, this is because it uses the pre-linked machine code in which no interpreting or references resolving information is needed to be contained.

Application image execution efficiency: For simpleRTJ, the Java byte code is executed, and its execution efficiency is lower than the machine code. For Contiki DLM, after the application module is resolved and linked, the final execution code is the pure machine code. For HEROS PLM, it also uses the machine code, but the access from the application SSIs to the system SPFs is done through the intermediate function jump table, thus the application image execution efficiency is a little lower if compared with the Contiki DLM mechanism, and the execution efficiency proportion value can be modeled as:

$$R_c = R_{HEROS}/R_{Contiki} = (10*C_f+N_i)/N_i = 1+10*C_f/N_i. \quad (3)$$

where C_f mean the numbers of the SSIs in the HEROS PLM applications, *10* is the clock cycles of the jump operation in the function jump table, N_i is the total clock cycles cost for the application execution. In case that N_i is

equal to *900* and the C_f is equal to *8*, then the R_c will be *1.089*.

Conclusion: From the evaluation results, it can be concluded that, the EJVM mechanism can be a good choice for the separation of the applications from the systems if employed on the resource-abundant embedded system. But for the platform of which the resources are high constrained, the HEROS PLM will be a preferable selection.

VIII. OS DEBUG METHOD

As the OS needs to dispatch and execute many system programs concurrently, its run-time process is complicated. Thus, it is essential to implement an effective debug method during the OS development stage.

The traditional debug ways for the embedded system include the usage of *printf*, the debug information sending out from the serial port, the breakpoint setting, etc.. However, these methods cannot work well for the debugging on the resource-constraint WSN devices. For the usage of *printf*, it cannot be used on sensor nodes, because there are commonly no screens on the sensor devices. Moreover, the execution overhead of *printf* is too high for the resource constrained sensor platforms. For the serial port debug method, it is also not ideal, because the sending speed of the debug data from the serial port is much slower than the execution speed of the instruction codes, thus the memory overflow problem will occur. As for the breakpoint setting, it is still not effective, becasue many interruptions and preemptions happened concurrently inside the system, once a breakpoint is set, the execution of the system will halt, so it is difficult to know how the system runs in whole detail. Due to these reasons, it is essential to develop a new OS debug method which will have the following features: *1).* has a low execution overhead, thus will not influence the node's regular execution process. *2).* can process the debug data in a high speed.

To address these challenges, a multi-core WSN system is designed and implemented. In this system, the sensor board is divided into two parts: the working board and the debug board. For the debug board, it is loaded only during the OS development process, and will assist the working board to undertake most of the debug work, such as buffering the raw debug data, analyzing them and displaying them out by string on the PC, etc.. By this means, the debug programs on the working board will become simple.

Seen in the Fig. 12, the working board iLive is connected with the debug board Raspberry Pi through the GPIO (global input and output) ports. Every time the iLive node takes some actions, it can send some raw debug code to the GPIO ports, and then let the left debug work be undertaken by the Raspberry board, e.g., when a new memory object is allocated on the iLive, three bytes debug code "*0xAC, 0x20, 0x1A*" will be sent to the GPIO ports. After Raspberry board receives these codes, it will interpret "*0xAC*" as the operation "*OS_malloc*", "*0x20, 0x1A*" as the new allocated address *0x201A*, and then transfer these debug results to the PC and display them out in the string or graphic format.

The reason of choosing GPIO ports for the board connection is because the operation of sending debug data to the GPIO ports is low in the overhead as well as high in the transmission speed. And since the Raspberry board is equipped with ARM BCM2835 controller of which the processor frequency is 700 MHZ and the SRAM size is 512 M bytes, thus it can be powerful enough to process the raw debug codes quickly, even if these codes are transmitted from the iLive with a high frequency. Consequently, a high debug performance can be achieved even on the severe resource constrained iLive node.

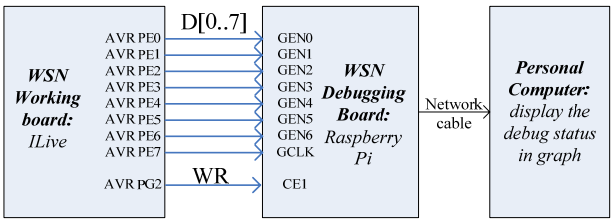


Figure 12. Multi-core WSN system for OS debug support

IX. RELATED WORKS

Scheduling policy and memory allocator are two important topics for an OS. For the memory allocator, the different mechanisms have been discussed in the previous section V, in this section the scheduling policy of different WSN OSs will be focused and presented.

A. TinyOS

TinyOS is an event-driven OS. Like HEROS, it also uses the event queue to buffer and dispatch the system events, but only one kind FIFO event queue is used, thus the intelligence of the system scheduler is not well enough. And in order to ease the application programming complexity, the multithreading scheduling mechanism TOSThread [32] is also implemented in TinyOS. With [32], the TinyOS applications can be programmed simply by the threaded programming model. However, the TOSThread thread scheduler is implemented in the user level with a priority lower than the TinyOS kernel scheduler, thus the event-driven scheduler always runs in precedence as long as the event queue is not empty, and the thread scheduler starts only when the event-driven scheduler becomes idle. Thus, TinyOS is still an event-driven OS in the native scheduling layer, although the thread scheduler is also implemented inside it.

B. SOS

SOS is also a pure event-driven OS, and uses the event queue to dispatch the events. Different from TinyOS, three kinds of priority-based queues are defined in SOS: the high priority queue, the regular system queue and the low priority queue. By means of this priority mechanism, the time critical events in SOS can be processed better than those in TinyOS.

C. BitCloud OS

Bitcloud OS is an OS used to support the full-featured ZigBee PRO stack developed by the Atmel corporation. Like SOS, it is a pure event-driven OS as well, however, it doesn't use the event queue to dispatch the events. Instead, it uses the priority-based event flag mechanism. With this mechanism, every time an event is generated, the flag related to this event will be set. And after an event handler runs to completion, the scheduler will scan all the event flags from the high priority to low priority ones. If an event flag is set, then the given event handler will be called.

D. MantisOS, uCOS and AVRX

Different from the OSs above, MantisOS [10], uCOS [13] and AVRX [14] are the multithreading OSs. All the system handlers are executed by threads, and every thread has its own run-time stack, thus the memory resource consumption of these OSs is higher than that of event-driven OSs.

The main difference between MantisOS and [13, 14] is that MantisOS is implemented dedicatedly to the WSN platforms, thus some mechanisms such as the energy resource reduction are realized inside this OS.

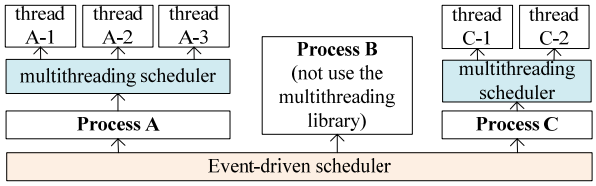
E. Contiki and LIMOS

For Contiki [8] and LIMOS [12], both the event-driven and multithreading scheduling models are implemented, seen in the Fig. 13 (a). Similar to the TOSThread in TinyOS, the multithreading model in [8,12] is implemented as an optional library upon the event-driven model. By means of this scheduling model, the preemption can be achieved among the child threads inside a process. However, the real-time response to the time-critical events can still not be well solved by this scheduling structure, e.g., when the thread *A-1* is in running, if the thread *A-3* needs to be executed quickly, the thread switch from *A-1* to *A-3* can be achieved quickly in real-time. But if process *A* is running and a HRT event is triggered which needs to be processed by the thread *C-2*. In [8, 12], this HRT event cannot be responded immediately as all the processes are scheduled by the event-driven scheduler. By this scheduler, process *A* needs to run to completion before the other processes can be scheduled. Therefore, Contiki and LIMOS are not real hybrid OSs, but actually event-driven OSs in the native layer.

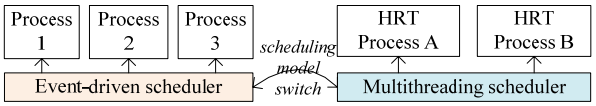
F. HEROS

For HEROS, it is implemented as a native hybrid OS, both the event-driven and multithreading schedulers are implemented. Different from TinyOS, Contiki and LIMOS, in HEROS the two kinds of schedulers are implemented in parallel and can switch to each other efficiently when required, seen in the Fig. 13 (b). With this hybrid scheduler, the hard real-time reaction can be achieved.

As for the differences between HEROS and the other pure multithreading OSs [10, 13, 14], there are mainly two aspects: *1).* In the pure multithreading OS, all the handlers will be processed by threads, thus the number of the threads will be much more than that in HEROS, this will increase the complexity of the system architecture as well as the memory resource consumption, e.g., in mantisOS, in order to manage all these threads well, it needs to use a complicated thread TCB as well as 5 priority-based thread queues for the management of the threads dispatching. *2).* In the current pure multithreading OSs, all the threads are started at the system initialization process, and once a thread is started, the thread stack needs to be allocated. Thus, a huge memory resources need to be consumed. In HEROS, the threads are started only when the HRT events are generated, and they will released after the HRT handler runs to completion. Due to these reasons, HEROS can support hard real-time reaction, but consume less memory resources than the pure multithreading OSs. And these results has been validated in the next section.



(a). Hybrid scheduling structure of Contiki and LIMOS



(b). HEROS scheduling structure

Figure 13. Hybrid scheduling structure in Contiki, LIMOS and HEROS.

X. Experiment and Evaluation Works

In this section, the performance of HEROS is evaluated from the aspects of OS features, memory resource consumption, energy resource consumption, execution efficiency and the portability.

A. Evaluation Platform

Currently, HEROS is implemented on the iLive node. ILive node is equipped with the AVR ATmega1281 microcontroller, seen in the Fig. 14. It has 128 kilobytes of FLASH and 8 kilobytes of RAM. Besides, it is equipped with 11 sensors including 1 temperature sensor, 1 light sensor, 1 air humidity sensor, 3 decagon sensors and 4 watermark sensors. As for the software development environment, the AVR studio integrated development tool [33] is used.

B. OS features

The feature comparison results are shown in Table IV.

Figure 14. ILive sensor node.

TABLE IV.
OS FEATURES COMPARISON RESULTS

WSN OSs	Features of Kernel				
	Scheduling policy	Real-time guarantee	Memory management	Uncouple APP from system	OS simu-lation
TinyOS	Event-driven OS + multithread-ing library	None	Static allocation	Yes, by VMs	TOSS IM [34]
Contiki 2.6	Event-driven OS + multithread-ing library	Only SRT	Dynamic (in fixed size block)	Yes, by DLM	COOJ A [35]
SOS 2.0.1	Event-driven	Only SRT	Dynamic (in fixed size block)	Yes, by DLM	N/A
Mantis OS 1.0	Multithread-ing	SRT supported	Dynamic	None	XMO S [36]
BitClou d	Event-driven	None	Static	None	None
LIMOS	Event-driven OS + multithread-ing library	Only SRT	Dynamic (in fixed size block)	None	N/A
HEROS	Native hybrid	HRT supported	Dynamic	Yes, by PLM	QEM U [37]

Since Contiki and LIMOS are actually native event-driven OSs, only SRT response can be achieved to the timer-critical events. For HEROS, the real time response can be achieved by means of the multithreading thread switch. For mantisOS, the preemption can be supported by the multithreading scheduler, thus it has the real-time performance better than that in the TinyOS, Contiki and SOS. However, it doesn't implement any real-time scheduling algorithm, thus it is still not a real real-time OS, and can support only the soft real-time reaction. For the simplification of the application development process, TinyOS, Contiki, SOS and HEROS all uncouple the applications from the systems by different mechanisms. For MantisOS, it doesn't separate the applications from the systems, but in order to reduce the application reprogramming code size, it uses the diff-patch code to reprogram the new applications, and the reprogramming performance can be greatly improved by this way. For the OS simulation, different methods have been adopted [34-37]. For HEROS, the QEMU [37] is used. [37] is generic and open source. Moreover, it can stimulate the OS and programs that are made on one machine on the other machine, e.g., the AVR platform HEROS code can be stimulated on the X86 PC.

C. Memory Resource Consumption

Memory resource consumptions of different OS components are shown in the Table V.

For the required ROM resources of multithreading scheduler, mantisOS consumes much more than Contiki and HEROS, this is because it is a pure multithreading OS. For it, no matter a handler is time-critical or not, one thread should be created for its execution, thus more threads are required to be created, and this increases the system architecture complexity.

The required RAM resources of different components is also shown in Table V. For the event-driven scheduler, Contiki and HEROS requires *(10+6e+8p)* and *(12+12e)* respectively, where *e* represents the number of system events that have been generated but not yet been processed, *p* presents the number of pre-defined system processes, 8 is the Contiki process structure size, and 6 and 12 are the event structure size. For the memory allocator, the Contiki block allocator requires *(4+8*Nb)*, where *Nb* is the number of the statically allocated block areas, and 8 is the size of the block management structure. For the multithreading scheduler, it needs the RAM resources for the allocation of the thread TCB and the thread stack: (*Ctn, Mtn, Htn)* are the reserved numbers of TCB, *(8, 22, 8)* are the TCB structure size, and (*Csk, Msk, Hsk*) are the stack size for the active threads.

TABLE V.
COMPARISON RESULTS OF MEMORY RESOURCE CONSUMPTION

Titles	***ROM Code** (bytes)*		
	Contiki 2.6	***MantisOS***	***HEROS***
Event-driven scheduler	936	N/A	602
Multithreading	678	3232	988
Memory allocator	298	708	832
timers	740	736	666
Total	2652	4676	3088

Titles	***RAM Data** (bytes)*		
	Contiki 2.6	***MantisOS***	***HEROS***
Event-driven scheduler	10+6*e*+8*p*	N/A	12+12*e*
Multithreading	8+8**Ctn*+*Csk*	40+22**Mtn*+*Msk*	4+8**Htn*+*Hsk*
Memory allocator	4	2	6
timers	10	10	2
Total	32+6*e*+8*p*+8*Nb* 8*Ctn*+*Csk*	52+22*Mtn* +*Msk*	24+*12e*+ *8Htn* +*Hsk*

To evaluate the required RAM data of different OSs, assumed that *e* is equal to 15, *p* is equal to 10, *Nb* is equal to 10. And for the thread numbers *tn*, mantisOS commonly has the most and HEROS has the least, thus it can be assumed that *Ctn* is 5, *Mtn* is 10 and *Htn* is 3. Then, the total RAM consumption of these OSs will be as follows:

Contiki: *(202+(40+Csk))*
MantisOS: *(272+Msk)*
HEROS: *(228+Hsk)* *(4)*

For the thread stack, it is needed for the storing of the thread run-time context, and its size is commonly as high as 100 bytes. Assumed that the thread stack is set to 100 bytes, and the number of the threads that are active in Contik, MantisOS and HEROS are 3, 6 and 2 respectively, then the total RAM consumption of Contiki, mantisOS and HEROS will be:

$(S_{Contiki}, S_{MantisOS}, S_{HEROS}) = (542, 872, 428bytes).$ *(5)*

Since the multithreading in Contiki is used as an optional library, it may not be required to be linked. And in HEROS, the multithreading mechanism will also not be launched if the HRT events are not triggered. Thus, for these cases, the total RAM consumption of Contiki and HEROS will be:

$(S_{Contiki}, S_{MantisOS}, S_{HEROS}) = (202, 872, 228bytes).$ *(6)*

From these results above, it can be concluded that: *1).* The RAM consumption of pure event-driven OS can be much smaller than the multithreading one, seen in equation *(6)*, and this is because the creation of the thread stack is high consumption in the RAM resources. *2).* HEROS consumes less RAM resources than the multithreading mantisOS as well as the hybrid scheduling Contiki, seen in the equation *(5)*, this is because it classifies the system events into three kinds, and only the particular HRT events will be processed by threads, thus the active thread number in HEROS can be less than Contiki and mantisOS.

D. Energy Resource Consumption and Node Lifetime

Energy resources on most sensor nodes are constrained. In order to reduce the energy consumption, the sleeping mechanism is adopted in HEROS. By this mechanism, sensor nodes are configured to sample and transmit the sensor data periodically., Besides, in case that there are no events existed in the HEROS event queue, the sensor nodes will also fall asleep.

With the sleeping mechanism and the energy efficient hardware design, iLive becomes an energy aware sensor node. Currently, iLive node is powered by two 1.5V AA standard batteries, and the energy consumption status is listed in Table VI. It is computed that, if the temperature and light sensors are loaded and the sensor data sampling frequency is set to 3 minutes, the lifetime of iLive can be as long as 826 days. This result is significant, because the sensor nodes are prone to be deployed in some harsh environments where the humans cannot access. Therefore, by means of this energy efficient iLive system as well as the HEROS PLM reprogramming mechanism, the sensor nodes can be avoided to be brought back for the application reprogramming after they have been deployed.

TABLE VI.
ENERGY CONSUMPTION STATUS OF ILIVE SENSOR NODE

Tasks	Working Current	Time cost
Instructions execution only	20.1mA	N/A
Low-power sleep mode	1uA	N/A
Temperature&light sensors sampling	15.6mA	900ms
10 bytes FLASH programming	20.8mA	15ms
Wireless packet transmission	20.7mA	N/A
50 bytes Wireless reception	21.5mA	140ms

E. Performance of basic OS Primitives

The clock cycles of different HEROS primitives are shown in Table VII. In this table, *q* means the counts of searching of the correct position in the freed memory queue, *r* means the counts of searching the correct reference from the chunk reference area, *u* means the number of the references to be updated, *s* means the memory size to be shifted, *f* means the count of searching a free Contiki block flag, *t* means the count of searching the next active thread from the TCBs, and *p* means the count of searching the correct position in the mantisOS priority-based thread queues. For HEROS memory allocator, if the fragment assembling is not processed, the allocation cost is *(87+16q+8r)* cycles. Otherwise, it will be *(136+16q+8r+9u+16s)* cycles.

For the execution cost of the multithreading primitives, Contiki is less than the others as the thread stacks in Contiki are pre-reserved statically. For mantisOS, it costs the more as it is a pure multithreading OS with more threads defined, thus the thread management process is more complicated.

In case that the main frequency of ATmega1281 on the iLive node is set to *16 MHz*, and the value of *(q, r, t, u, s)* is *(3, 5, 2, 4, 80)* respectively, then the execution time of the HEROS primitives can be calculated: For event post, it is *3.63 us*. For common and SRT event extraction, they are *0.63 us* and *1.12 us* respectively. For memory allocation and free operations, they are respectively *10.94 us* and *6.13 us*. If the first-time memory allocation is failed, the fragments will be assembled, then the cost time will be *96.25 us* and this is the worst case of the memory allocation cost. As for thread context creation, it is *13.5 us*. And for the thread context switch, it is *8.38 us*.

Since after a HRT event is generated, the time cost between the generation of this event and the execution of the related event handler is $T_{HRT} = T_{CS} + T_{SC,}$ where T_{CS} is the time cost for creating this thread's run-time stack, and T_{SC} is the time cost for switching the thread context, then the T_{HRT} can be computed and it is *21.88 us*. As the execution time is related to the processor frequency, if the professor frequency increases, all these time value above will decrease correspondingly.

From the results above, it can be concluded that the dynamic memory allocator is not so complicated and inefficient that it can be implemented on the resource-constrained WSN system. Moreover, if used on the AVR ATmega1281 microcontroller, HEROS can support the HRT reaction with the response time limited to *21.88 us*. Thus, for the real-time WSN applications of which the reaction time constraint is not as strict as this, the HEROS system with the iLive node can be competent.

TABLE VII.
PERFORMANCE OF BASIC OS PRIMITIVES

Basic OS Primitives	Execution Cost (clock cycles) [min, max]		
	HEROS	Contiki	MantisOS
Event post	22+12q	28	N/A
Event extraction	[10, 18]	32	
Memory allocation	[87+16q+8r, 136+16q+8r+9u+16s]	28+12f	98+12q
Memory free	62+12q	38+12f	56
Thread context creation	112+(56+16q)	106	96+(98+12q)+16p
Thread context switch	98+18t	116+16t	152+18p

F. Portability

The portability becomes essential for an OS as the hardware platforms of WSN are diverse. To port HEROS to a new hardware platform, all the code that is hardware specific should be adjusted, and this mainly includes two aspects. One is the variable type length, e.g., the integer length on the ATmega1281 microcontroller is 2 bytes while on AT91SAM7S256 it is 4 bytes, thus some data structures should be adjusted. And the other is the assembler codes in the OS components such as the basic hardware initialization, the multithreading scheduler, the software timers, etc. It is computed that to port HEROS from the AVR ATmega1281 platform to the ARM AT91SAM7S256 platform, 92 lines of code needs to be adapted.

XI. CONCLUSION AND PERSPECTIVE

In this paper, a hybrid, resource-aware, real-time and user development friendly OS HEROS is presented. The final performance evaluation and experimental work proves that HEROS can be used on the resource-constrained sensor nodes to support the real-time WSN applications.

For the ongoing work, the following topics will be focused:

Fault-tolerant system: A fault-tolerant system can make the sensor nodes to continue the normal working even if some faults occur on the nodes. To achieve the fault-tolerant system, some measures will be taken from both the hardware and software aspects. For the hardware aspect, the multi-cores WSN nodes are currently developing in our team. With the multi-core platform, the sensor node's fault tolerance can improve by means of the cooperation among the different microcontrollers. For the software aspect, some dependable concepts such as the state machine checking and validation, the run-time monitoring profiler, the roll-back recovery, etc., will be implemented in the next version HEROS.

Distributed system: Due to the development of multi-core hardware platform, HEROS is currently developing toward a distributed OS. This means that HEROS will be capable of splitting a complicated system task into several child tasks and distributing these child tasks to the different microcontrollers. Moreover, it will support the ability of sharing the memory resources on the different microcontrollers.

Application code generation and reprogramming: With the HEROS PLM implemented, the user application project becomes very simple, thus it is feasible to generate the application program automatically by the GUI (Graphic User Interface) toolkit and reprogram the application remotely by the webpage. With the GUI toolkit, the users are not required to be professional in the programming. Instead, they choose some graph modules, connect them logically, configure the module parameters and then generate the application programs. Later, these programs will be transmitted by the web page to the servers, to be built by the HEROS PLM and then updated to the sensor nodes. Currently, such an integrated development environment is under construction by our team with the address: http://edss.isima.fr.

ACKNOWLEDGMENT

This work has been sponsored by the French government research program "Investissements d'avenir" through the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01), by the European Union through the program Regional competitiveness and employment 2007-2013 (ERDF–Auvergne region).

Our thanks also to the China Scholarship Council (No. 2009627016), the China National Natural Science (No. 60903195), and the support from the project "Space satellite research on collaborative ad hoc networks" by the Innovation Fund of Satellite Application Institute in China Aerospace Science and Technology Corporation.

REFERENCES

[1] J Yick, B Mukherjee, D Ghosal. "Wireless sensor network survey", Volume 52, Issue 12, August 2008.

[2] K. Shinghal, A. Noor et al. "Intelligent Humidity Sensor For Wireless Sensor Network Agricultural Application", Journal of Wireless&Mobile Networks, Vol. 3, No. 1, 2011.

[3] A. Flammini, Paolo Ferrari et al. "Wired and wireless sensor networks for industrial applications". In Microelectronics Journal, pp. 1322-1336, 2009.

[4] I. Kirbas et al.. HealthFace: A web-based remote monitoring interface for medical healthcare systems based on a wireless body area sensor network. Journal of Electrical Engineering&Computer Sciences, pp. 629-638, 2012.

[5] W Dargie, C Poellabauer. "Fundamentals of wireless sensor networks: theory and practice", Wiley Series on Wireless Communications and Mobile Computing, 2010.

[6] WSN mini hardware survey. http://www.cse.unsw.edu.au/~sensar/hardware/hardware_survey.html.

[7] J. Hill, R. Szewczyk, et al., "System architecture directions for networked sensors," in ACM SIGOPS Operating Systems Review, vol. 34, pp. 93–104, December 2000.

[8] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki – a lightweight and flexible operating system for tiny networked sensors," in International Conference on Local Computer Networks, pp. 455–462, November 2004.

[9] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in Int'l Conf. MobiSys, pp. 117–124, June 2005.

[10] S. Bhatti, J. Carlson, H. Dai, J. Deng et al.. "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," ACM kluwer Mobile Networks & Applications Journal, August 2005.
[11] Atmel AVR2050: Atmel BitCloud, developer guide. 2012.
[12] Hai-ying Zhou, Feng Wu, Kun-mean Hou. "An Event-driven Multi-threading Real-time Operating System Dedicated to Wireless Sensor Networks", Int'l Conf. on Embedded Software and Systems, Chengdu, China, 2008.
[13] J. Labrosse, MicroC/OS-II: The Real-Time Kernel, 2nd edition, CMP Books, June 2002.
[14] AVRX. Compact Real Time Scheduler, http://avrx.sourceforge.net/.
[15] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In International Conference on ASPLOS, San Jose, CA, USA, Oct. 2002.
[16] Philip Levis. Bombilla: A Tiny Virtual Machine for TinyOS. Version 1.0. September 25, 2002.
[17] Rene Mueller, Gustavo Alonso et al.. "SwissQM: Next Generation Data Processing in Sensor Networks", 3rd Biennial Conference on Innovative Data Systems Research, January, 2007, Asilomar, California, USA.
[18] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for repro-gramming wireless sensor networks," in Proceedings of ACM SenSys, 2006.
[19] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, "A Comprehensive Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems", Journal of Networks, Vol. 3, No. 3, March 2008.
[20] J. Hellerstein et al.. Events and threads. Lecturer Notes, November 2005.
[21] J. K. Ousterhout. "Why Threads Are A Bad Idea (for most purposes)", Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.
[22] R. von Behren Jeremy Condit et al.. Why events are a bad idea (for high-concurrency servers). In 10th Workshop on Hot Topics in Operating Systems (HotOS IX), May 2003.
[23] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, "A performance analysis of TinyOS and MANTIS," Tech. Report. University College Cork, Ireland, November 2006.
[24] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in Proceedings of ACM SenSys, 2006.
[25] PIZLO. F et al.. Towards Java on bare metal with the Fiji VM. In Proc. Int'l Conf. JTRES'09. Madrid, Spain, September 2009.
[26] Fridtjof Siebert. Realtime garbage collection in the JamaicaVM 3.0. In Proc. of Int'l Conf. JTRES, September 2007, pages 277–278.
[27] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: The squawk Java virtual machine. In Proc. Int'l. Conf. VEE, pages 78–88, New York, NY, 2006.
[28] N. Brouwers, K. Langendoen, and P. Corke, B. Darjeeling, a feature-rich VM for the resource poor. in Proc. ACM Sensys, 2009, pp. 169–182.
[29] RTJ Computing Pty. Ltd. simpleRTJ a small footprint Java VM for embedded and consumer devices. Online at http://www.rtjcom.com/.
[30] T. Harbaum, the NanoVM - Java for the AVR, http://www.harbaum.org/ till/nanovm/index.shtml.
[31] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming wireless sensor networks: Challenges and approaches," IEEE Network Magazine, vol. 20, no. 3, pp. 48–55, 2006.
[32] K. Klues, C.-J. M. Liang, J. yeup Paek, et al., "TOSThreads: Safe and Non-Invasive Preemption in TinyOS," in Proceedings of ACM SenSys, 2009.
[33] AVR Studio 6. The Studio to Design All Embedded Systems. http://www.atmel.com/Microsite/atmel_studio6/.
[34] P. Levis, N. Lee, A. Woo, S. Madden, and D. Culler. Tossim: Simulating large wireless sensor networks of tinyos motes. Technical Report UCB/CSD-TBD, U.C. Berkeley Computer Science Division, March 2003.
[35] Fredrik Österlind. "A Sensor Network Simulator for the Contiki OS", SICS Technical Report, 2006.
[36] Hector Abrach, Shah Bhatti et al.. "Mantis - system supports for multimodAl neTworks on in-situ sensors". Conference On Embedded Networked Sensor Systems (SenSys), pp. 336-337, California, USA, 2003.
[37] QEMU, a generic and open source machine emulator and virtualizer. http://wiki.qemu.org/Main_Page.

Xing Liu received his bachelor engineering degree in 2007 from the Electronic Information School in Wuhan University. In 2004, he obtains the National First Classical Scholarship for students (top 0.3% in the National key universities). In 2007, he was recommended to become a graduate student of Wuhan University without the necessity of participating in the National Entrance Examination for MS. Candidates (top 10% of Wuhan University). In 2009, he received the MS. Engineering Degree from the department of Communication and Information System in Wuhan University. From 2009 to now, he was assigned by the Chinese government and China Scholarship Council to study in LIMOS (CNRS UMR 6158, FRANCE) to pursue the PhD degree of University Blaise Pascal. His research interests focus on the development of embedded real-time operating system and embedded Java virtual machine.

Kun-mean Hou was born in Cambodia in 1956. He held a PhD degree in 1984 and a HDR degree in 1996 in Computer Science from the University of Technology of Compiègne (UTC). He worked as associate professor at UTC from 1984 to 1986. In 1986 he joined IN2 as R&D engineer group leader. From 1989 to 1996, he created a research group which investigated parallel architecture dedicated to real-time image processing at laboratory HEUDIASYC UMR CNRS.

In 1997 he joined the college of engineering school 'ISIMA: Institut Supérieur d'Informatique de Modélisation et de leurs Applications' as professor, where he created the SMIR 'Systèmes Multisensoriels Intelligents integrés et Répartis' team of the laboratory LIMOS UMR 6158 CNRS (10 researchers) working on the development of basic hardware and software dedicated to WSN. Different sensor nodes (Bluetooth, WiFi and ZigBee), embedded wireless communication and embedded real-time kernel (SDREAM and LIMOS) are implemented and deployed in different applications such as telemedicine, intelligent transportation system and precision agriculture. He holds 3 EU patents, and he evolved in 3 EU projects and 10 technology transfers. He also evolved in several scientific committees and boards.