Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms

Yu-Ting Chen
Department of Computer Science
National Tsing Hua University
wtg.design@gmail.com

Pai H. Chou
Center for Embedded Computer Systems
University of California, Irvine
phchou@uci.edu

Abstract

In this thesis, we propose Enix, a lightweight dynamic operating system for tightly constrained platform for wireless sensor networks (WSN). Enix provides cooperative threading model, which is applicable to event-based WSN applications with little run-time overhead. Virtual memory is also supported with the assistance of the compiler, so that the sensor platforms can execute code larger than the physical code memory they have. To enable firmware update for deployed sensor nodes, remote reprogramming ability is available in Enix. The commonly used library and the main logical structure are separated; each sensor device has a copy of the dynamic loading library in the Micro-SD card, and therefore only the main function and user-defined subroutines should be updated through RF. A lightweight, efficient file system named EcoFS is also included in Enix. The code size and data size of Enix with full-function including EcoFS are 8 KB and 512 bytes, respectively, making Enix the smallest one compared to other WSN OSs.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, LaTeX, text tagging

1 Introduction

The right runtime support for a resource-constrained embedded systems platform can make a great difference in the amount of effort required to develop applications for wireless sensor networks (WSN).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1.1 Motivation

Fig. 1 shows three scenarios that WSN researchers often encounter. The first issue is the size and cost problem. Current WSN platforms, for example, Mica2 [22] and MicaZ [2], have a development board with two AA batteries that may be "small" when compared to a computer but are actually, big, heavy, and expensive to wear and embed for many real-life wireless sensing applications. Wearable sensing system, including both human and animal worn ones, impose possibly the most stringent constraints on the size and weight of the entire system, as shown in Fig. 1(a).

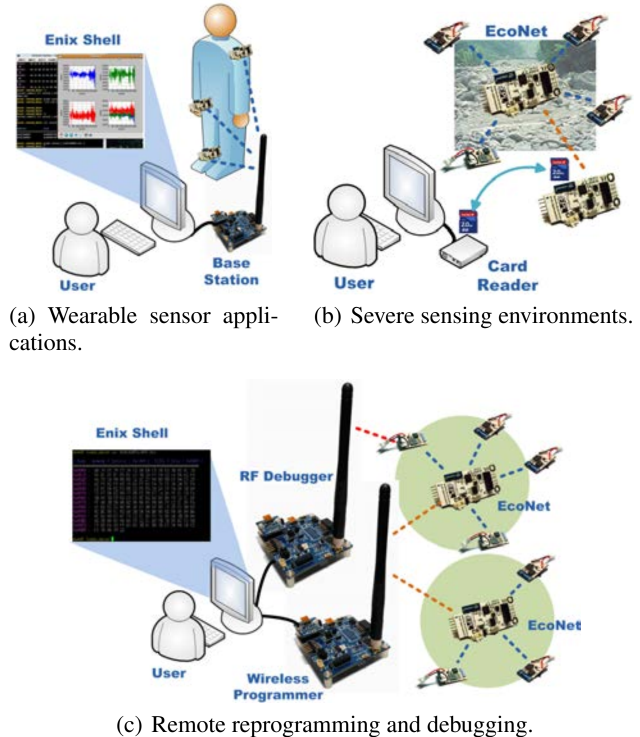
To enable better fitness to wearable applications, some ultra-compact wireless sensor platforms have been developed [33, 37]. The Eco node, which uses an 8051/8052-compatible MCU core, is only 1cm^3 in volume and weighs under 2 grams, making it most competitive in terms of cost and lightweight metrics. However, it is also highly constrained in terms of the amount of memory and computing ability, which pose new challenges on the operating system design.

Fig. 1(b) depicts another real-world WSN application. Most of the outdoor sensing environments have rugged topography and scarce resources. For example, in the Amazon forest and deep ocean, the sparce deployment, long physical distance, and harsh environmental condition make it impossible or impractical to transmit wirelessly even across multihop nodes to a base station with continuous power supply. Instead, the sensor nodes collect data from these severe sensing environments by storing sensed data in nonvolatile memory for later retrieval. Besides, some applications require sensing data at a high sampling rate during specified intervals of time. The common sense-and-transmit pattern used in WSN may not work due to the high sampling rate requirement. Therefore a sensed, logged and transmit (SLT) pattern has been shown useful [12] to solve the problem. According to these RF-unavailable environments and SLT purposes, a reliable and efficient storage system is required. Some general-purpose file systems such as FAT, EXT, JFFS and so on have the simplified version that claimed appropriate for embedded systems. However, the limitations of resources in lightweight sensing systems are far more than the embedded systems mentioned above. Furthermore, the majority of lightweight sensor devices are MMU-less, and thus the lack of hardware for supporting memory management is an issue. To make use of the second storage medium such as

flash memory and EEPROM, software virtual memory can be achieved by using a compiler-assisted scheme, thereby overcoming inadequate built-in code memory of MCUs.

To provide effective management of tight constraints on hardware resources and to enable runtime support of wireless sensor nodes, a lightweight system kernel that is designed specifically for WSN has become increasingly important. Most of people think that an operating system may cause additional overhead and power consumption especially for resource-constrained sensing systems. However, an operating system may actually improve efficiency by supporting power management, memory management, task management, and multi-threading.

Fig. 1(c) shows typical capabilities of runtime operating systems. In addition to the basic scheduling for event-driven or multi-threading models, several other types of runtime support are also becoming important for wireless sening applications. For example, virtual memory enables the MCU to execute larger, more sophisticated programs than its physical memory alone allows. Another important type of support is remote programming, as it may be nearly impossible to update firmware of sensor nodes through wired interfaces after they have been deployed in large numbers. For this reason, a loader that supports dynamic reprogramming becomes an important issue when designing an operating system for WSNs. Users expect that firmware update can be done using a host PC shell environment. Code image will be sent through RF to remote sensor nodes and loaded by a dynamic program loader at runtime. Another issue that arises from remote reprogramming is the size of the code image. In order to reduce the power consumption, sensor nodes should reduce the RF transmission as more as possible; moreover, they should also avoid idle-listening, since that consumes twice the power as the highest RF transmission. A good run-time system may also provide an efficient mechanism to reduce the image size in remote reprogramming. Some common schemes such as patch generation and compression can achieve the objective, but these also increase the runtime overhead and consume considerable runtime memory. As a result, a better solution to the problem must be presented.



(a) Wearable sensor applications.

(b) Severe sensing environments.

(c) Remote reprogramming and debugging.

Figure 1. Enix reference applications.

1.2 Problem Statement

Our goal is to implement a new operating system for wireless sensor platforms. We assume the wireless sensor device is ultra-compact with a small quantity of GPIOs and inadequate code and data memory. It has sensors to collect data and the ability to exchange data through RF; common interfaces such as UART, I2C, and SPI are also available. The external non-volatile memory such as serial flash and Micro-SD card can be connected to the sensor device through SPI. In the following subsections, we list the requirements of a WSN OS design.

1.2.1 Lightweight and Portability

The operating system is supposed to be appropriate for the more resource-constrained version of the current wireless sensor platforms with different MCU ISAs. Low memory and power consumption must be achieved by utilizing only limited resources in order to increasing the life time of wireless sensor devices A portable interface and the reduction of assembly code implementation enable the operating system to be ported to other MCU ISAs.

1.2.2 Appropriate Programming Model

An appropriate programming model not onlyi facilitates software development but also promotes good programming practices. Event-driven programming model is widely used in WSN but has several drawbacks; for example, tedious, unstructured, and repetitive [4, 8, 20, 36]. Consequently, an easy-to-use threading structure that is applicable to event-based WSN applications with little runtime overhead is desired. Context switching and the scheduling policy are the two main sources of run-time overhead of multi-threaded programming; the efficiency of the scheduler must be improved by fast algorithms instead of linear searching for the next running thread; the overhead of context switch can also be reduced by simply storing the critical registers into stack and avoiding using the slow external memory.

1.2.3 Virtual Memory

Virtual memory can overcome the shortage of memory resources problem that a resource-constrained wireless sensor device usually faces. It can be achieved via the assistance of compiler without hardware MMU support; each function is compiled into a code segment that is then stored on the Micro-SD card at a specific virtual address; the segment is loaded on demand while the user program calls this virtual address at run-time. Memory compaction and garbage collection must be implemented to solve the external fragmentation problem and to recycle the unused memory for future allocation. Users should write program without any concerning about what the run-time demand segmentation module

does, and no additional tags and no restriction on function prototypes would be applied to user programs.

1.2.4 Remote reprogramming

To enable run-time firmware updates for deployed wireless sensor devices, wireless reprogramming is required in WSN OS design. Dynamic loading must be achieved to enable partial update instead of the whole image update in order to reduce the time and energy cost while doing remote reprogramming. Position-independent code (PIC) is a suitable approach to achieving dynamic loading; the compiler can assist the generation of PIC on host PC, thus avoiding the overhead of relocating addresses at run-time.

1.2.5 File System

A lightweight file system supported by the operating system is necessary to assist the access to data storage. The design of file system is targeting for WSN applications instead of general purpose file system. Wear leveling and erase-before-write are handled by the MCU in Micro-SD card in order to reduce code size. The file system should be used to store binary code data, preferences of sensor devices and network data such as routing table and the sensed data. The API provided to users must be specific designed and optimized according to the different features such as access pattern of each type. The file system must be configurable, and therefore only the necessary storage types are chose in order to conserve code memory.

1.3 Objectives

Enix provides a lightweight and dynamic operating system with a specialized file system EcoFS for tightly constrained wireless sensor platforms. The most important task for our work is to manage the limited resources of wireless sensor platforms and to provide a well designed programming model for WSN application developers. Enix provides cooperative threading model which enables multi-threaded programming for users with minimized overhead of context switch and reduced code size via a novel way modified from *setjmp/longjmp* system library. A replaceable scheduler is also provided therefore different schedule policies may be applied to WSN applications with different requirements.

The next goal of Enix is the compiler-assisted run-time supports. Run-time functionalities such as virtual memory, remote reprogramming and memory allocation are useful but challenging to implement due to the lack of resources. We take the approch of compiler-assist, which shifts the complexity to the resource-unconstrained part such as the host PC. Virtual memory can be achieve without hardware support via code insertion technique at compile stage of development process. Dynamic loading is very helpful for run-time reprogramming; common run-time relocation approach increases the overhead of sensor nodes; PIC approach can be used to achieve run-time programming with minimized overhead. Accordingly, with the help of host PC, the run-time overhead of wireless sensor nodes is reduced.

In the case of EcoFS, the goals are highly efficient file operation and low power consumption. The separation of four access types of EcoFS increases the access speed. By use of Micro-SD card as secondary storage results in additional energy consumption caused by the operating power required by the Micro-SD card. The design of EcoFS minimizes the active time of the Micro-SD card by I/O scheduling and by limiting the usage of the Micro-SD card via API provided by EcoFS; therefore, the overall power consumption is minimized.

Configurability and small memory consumption are also achieved. Each components in Enix is configurable, and therefore only the required components are configured and compiled before programming the sensor devices. For memory consumption, the code size and data size are both minimized. The code size of our full function implementation of Enix is 8 KB, and the data size is about 512 bytes. By excluding the unused components of Enix, the code and data memory consumption can be further reduced.

The rest of this thesis is organized as follows. Chapter 2 discusses related works on WSN OS design. Chapter 3 presents an overview of our lightweight WSN OS Enix and its design concepts. Chapter 4 describes the implementations of run-time components in Enix including the scheduler, compiler assistant virtual memory, runtime reprogramming, and runtime loading. Chapter 5 describes the file system EcoFS and the storage medium chosen by EcoFS. We evaluate Enix and EcoFS and present the results in Chapter 6. Finally, Chapter 7 conclues this thesis with a summary of contributions with Enix and discusses directions for future research.

2 Related Work

Wireless sensor networks are composed of sensor nodes and base stations. A sensor node has the following characteristics : small physical size, low cost, small code and data memory, limited computing capability, and limited battery power. Together, these factors limit the performance and complexity of WSN applications. For example, the ultra-compact Eco [33] wireless sensor platform has only 4 KB of instruction memory and 256 bytes of data memory. Some tiny real-time operating systems (RTOS) are able to run on such a resource-constrained platform, such as μC/OS-II [25] and FreeRTOS [1], both of which support preemptive multi-threading with round-robin or priority based scheduler. These RTOSs are indeed lightweight and well designed, but they are not suitable for wireless sensor networks because of the lack of functionality such as runtime code update, power management, and resource control capabilities. In response, researchers propose some WSN operating systems that provide support that are specific to WSN applications. Table 1 shows a comparison between existing operating systems aimed at WSNs.

2.1 Programming Model

TinyOS [27] is a widely used runtime system for wireless sensor systems. It uses a special language called nesC [18] to describe the software components that form a sensor system with event-driven semantics. The application code and the runtime library are then compiled into one monolithic executable. Several event-driven runtime-support systems have been developed for wireless sensor networks and applications with similar characteristics, including TinyOS [27], SOS [21], and Contiki [14]. The processes of these operating systems are implemented as event handlers that run

to completion without preemption. Therefore, event handlers in event-driven models may share the same stack to reserve insufficient memory space. Event-driven programming is based on cooperative multitasking, which may be good for tiny, single-processor embedded devices, but users have to perform stack management manually, and as a result the code can become difficult to read and maintain. Some operating systems for WSN provide preemptive multi-threading [8–10, 14, 16, 19, 34], so that achieving real-time ability and preventing the critical events from starvation caused by another event with heavy workload. Multi-threaded programming model is easy to learn comparing with event-driven model and the code is more readable and maintainable. In severe memory and power constrained environments, however, a multi-threaded model has several disadvantages. For example, it occupies a large part of the memory resources, spends more CPU time and consumes more battery power derived from context switch overhead.

Dunkels [15] proposes Protothreads to allow users to write event-driven programs in a threading style, with a memory overhead of only two bytes per Protothread. The concept of Protothreads is almost the same as C co-routines [23, 35] in that they implement "return and continue" by use of C-switch expansion. Protothreads has limitations in that auto variables or local states cannot be stored across a blocking wait, and the C-switch implementation may lead to a table lookup and jump overhead at runtime, and they also increase the code size. The runtime complexity is proportional to the number of yield points in a protothread. Our Enix OS emphasizes cooperative threading, which guarantees that no thread will have to yield control unexpectedly [20], and Enix threads work similarly to co-routines but they are implemented in a mix of C and assembly for smaller code size and better execution efficiency. Swapping between threads in Enix is a real context-switch operation, not just a C-switch. It provides automatic stack management and incurs little runtime overhead compared to preemptive multi-threading. In addition to cooperative threading, Enix also supports lightweight preemptive multi-threading for real-time scheduling. For the power consumed by context switches, MANTIS OS [8] shows that multi-threading and energy efficiency need not be mutually exclusive when an effective sleeping mechanism is used to reduce context-switching overhead.

2.2 Runtime OS support for WSN

In real-world wireless sensor networks, the deployed sensor nodes must have the abilities to manage the tasks and resources at run-time. Run-time reconfiguration and reprogramming also become important issue in WSN OS design. TinyOS, as mentioned before, produces a single image that the kernel and applications are statically linked, thus, updating code image means whole system image replacement. In order to provide efficient runtime remote reprogramming for TinyOS, Maté [26], a virtual machine for TinyOS has been proposed. Using Maté and other virtual machines for WSN [7, 24, 30], code can be distributed and configured at run time. The drawbacks when running virtual machine on sensor nodes include the requirement to learn another language and the runtime overhead of the virtual machine interpreter. The interpretation overhead may result in more energy consumption and might decrease the life time of sensor nodes. SOS [21] is another event-driven OS but consists of dynamically loaded modules and a common kernel. The modules are position independent code (PIC) binaries that implement specific tasks or functions. This modulized design is quite flexible, but the interaction between modules may incur high runtime overhead, and the module must be implemented in fixed format, which increases the entire code size of SOS. To support more than one MCU, Contiki [14], RETOS [10] and LiteOS [9] provide dynamic loading by runtime relocation, rather than relying on position independence. The application binary to be combined with relocation information must be relocated or dynamic linked [13] at run time before loading to program memory, that is, a considerable data buffer is required in order to relocate in space. Our Enix OS support dynamic loading using kernel-supported PIC, which is easy to port to other platforms. The dynamic library is pre-linked to the kernel with minor modification. Hence, our runtime overhead and additional buffer are reduced compared to the runtime relocation approach.

2.3 Virtual Memory Management

To fully utilize the memory of a tightly constrained wireless sensor platform, some researchers propose software virtual memory in MMU-less embedded systems. It can be achieved by code modification by either using a compiler or constructing an additional converter. SNACK-pop [32] provides a framework for compiler-assisted demand code paging with static call graph analysis and optimization. The input Executable and Linking Format (ELF) [3] file is analyzed and translated into an executable image such that every call/return is modified to call the page manager. Choudhuri and Givargis [11] indicates that data segment has a greater need to be paged than code segment, and therefore they propose a data paging scheme with adjustable page size based on an application-level virtual memory library and a virtual memory-aware assembler. t-kernel [19] is the first WSN OS that provides virtual memory for both code and data segments with additional memory protection ability. In addition to software virtual memory approach, MEMMU [6] proposes a new software-based on-line memory expansion technique [5] that requires no secondary storage. Therefore, it increases the performance and minimizes the power consumption comparing to the above approach. However, MEMMU introduces about 4 KB of code size overhead and requires at least 512 bytes of data memory that is not applicable to sensor platforms with tightly constrained code and data memory [33]. Besides dynamic loading, Enix also supplies software segmented virtual memory by code modification, and uses a Micro Secure Digital (SD) card as secondary storage due to the convenience of installing virtual code segments through the host PC. Enix does not provide virtual memory of data segment because of the high runtime overhead, but it does support a data memory allocation scheme to fully utilize the slight data memory. Further, Enix consumes only 886 bytes of code and 256 bytes data memory.

2.4 File Systems for WSN

Storage is an essential factor of data-centric sensor network applications. A well-designed file system may help conserving power and provide convenient non-volatile data management of wireless sensor nodes. In Table 2, a comparison of WSN file systems is presented. LiteFS is a subsystem of LiteOS [9] that provides a hierarchical, Unix-like file system that supports both directories and files. This kind of general-purpose file system along with FAT, Ext2 and other common file systems are not really suitable for WSNs due to the relatively large code size and a great deal of data memory space used to store the hierarchical data structures. Another issue is that the general file format becomes insufficient to store WSNs data due to the absence of fast query support. There are some file systems or databases customized design for WSNs [12, 17, 28, 29, 31, 38]. ELF [12] uses NOR flash to implement a log structured file system. It expects to encounter three major sources of data: sensor data, configuration data, and binary program images, all of which have different access patterns. However, it is not applicable to storing a time series of sensor data and maintaining an index of them to support queries, MicroHash [38], TinyDB [28] and FlashDB [31] use lightweight index structures or databases that work on wireless sensor nodes. Due to supporting high-performance indexing and searching capabilities, in-memory data structures overhead is inevitable in these systems. Capsule [29] covers the abilities mentioned above. It provides the abstraction of typed storage objects to applications including streams, indexes, stacks, and queues. The composition of different objects may satisfy varies requirements of WSNs. However, the high capacity parallel NAND flash used by Capsule makes use of many general-purpose input/output (GPIO) ports, and therefore it does not work on small-sized microcontroller units (MCU) with few available I/O ports. Although the flash abstraction layer in Capsule supports multiple flash devices, it still focus on flash devices without random access capability. Enix includes EcoFS, a lightweight file system using the Micro-SD card as its storage medium, which has high capacity as NAND flash and requires only four I/O lines for the Serial Peripheral Interface (SPI) compatible protocol. Moreover, due to the replaceable feature of SD card, we also builds an EcoFS Shell on the host PC to quickly access an EcoFS-formatted Micro-SD card. EcoFS provides four storage types inclusive of code data, preferences, sensed data and network information with different access patterns in order to improve the performance and decrease the code and data size. It consumes the smallest data memory among all existing file systems designed for WSNs.

3 Enix Overview

This chapter provides an overview of Enix. We first describe the system architecture and the functionality of each component. Next, we present the concepts of Enix design.

3.1 System Architecture

Figure 2 shows the block diagram of Enix, which currently runs on a wireless sensor platform called EcoSpire. The Enix operating system contains four major components. The first component is the *runtime kernel* of Enix that man-

Table 1. Comparison between WSNs operating systems.

OS	Enix	TinyOS	SOS	Contiki	MANTIS OS	t-kernel
Platform	Nordic nRF24LE1	ATmega128L	ATmega128L	ATmega128L & MSP430	ATmega128L	ATmega128L
Programming Model	Thread	Event	Event	Event & Thread	Thread	Thread
Real-time Support	△ [1]	△		○	○	○
Dual Mode Operation	○		○	○	○	○
Remote Update	○	△	○	○	○	
Dynamic Loading	○ [2]	△	○ [2]	○ [3]		
Protection		△				○
Virtual Memory	○ [4]	△				○ [5]
File System	○	△				
Network Abstraction	○	△				
Code Size (Bytes) [6]	8,138	21,132	20,464	3,874 [8]	14,000	28,864
Data Size (Bytes) [7]	512	597	1536	512	512	512

[1] △ means optional components.
[2] Achieved using PIC.
[3] Achieved using runtime relocation.
[4] Support code virtual memory only.
[5] Support both code and data paging.
[6] The code size including the basic kernel and the ○ components, excluded △s.
[7] List the smallest data memory required to startup OS, at least one thread in multi-threaded model.
[8] The code size without *TinyOS* which is the base of Contiki.

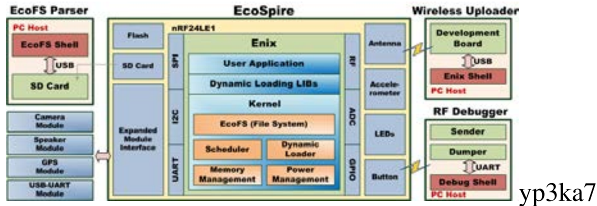
Table 2. Comparison between WSNs file systems.

File System	EcoFS	FatFS	LiteFS	ELF	C
Platform	Nordic nRF24LE1	Nordic nRF24LE1	ATmega128L	ATmega128L	ATm
OS	Enix	No	LiteOS	TinyOS	T
Storage Medium	SD/MMC Card	SD/MMC Card	EEPROM and Flash	NOR Flash	NOI
Abstraction	File System	File System	Unix-like File System	File System	C
Usage Model	File Storage & Network Info & Virtual Memory	General Purpose File Storage	General Purpose File Storage	WSN Specific File Storage	Data an
Line Counts [1]	447	1,757	1,755	3,577	
Data size (Bytes)	36	552	104	512	

[1] The lines of source code is measured by C and C++ Code Counter (CCCC) tool.

ages hardware resources and supports run-time reconfiguration. This component provides memory and power management capabilities for the wireless sensor node to utilize the limited memory and energy resource. With a *cooperative thread scheduler*, developers can write multi-threaded programs to overcome the insufficient single-threaded program. A *wireless code image update manager* and a *dynamic loader* are also included to enable runtime reprogramming, so that the deployed sensor nodes can be easily updated through RF.

Although the file system is one of the modules in the runtime kernel, it provides more support for WSN applications. Consequently, we separate the file system as an individual component of Enix. EcoFS, the file system of Enix, is a configurable and lightweight storage system using a Micro-SD card or an MMC card as the storage medium. EcoFS is divided into the following parts according to the different usage patterns: code data, preferences, network data and sensed data. The code data block stores the binary code segments that can be loaded efficiently by the runtime loader into the code memory on demand. The preferences are the key-and-value pairs applicable to store the node's status and settings. The requirement of preferences is fast searching with modify enabled. Because of the limited data memory and the wish of providing a simple network abstraction, the routing table may be maintained in EcoFS. The network data such as routing tables or the roles of the adjacent nodes must be enumerable and may be changeable if dynamic network topology is applied. Last but by no means the least,



yp3ka7

Figure 2. The block diagram of Enix.

the sensed data block is used to collect the valuable sensing data in harsh environments. This append-only storage type must be quick and efficient as a result of the requirements of high-sampling-rate WSN applications.

The third component of Enix is the dynamic loading library ELIB. ELIB is a special library that is preprocessed by host PC tools. Most of the commonly used library functions are collected and transformed into *segments* which compose ELIB. Each segment is a position-independent function binary code with a unique virtual address, that is, an address where the segment is located in the secondary storage. Due to the constrained memory resource of compact wireless sensor devices and the high energy consumption of the RF transceiver, making use of ELIB enables software virtual memory and reduces the transmission size of runtime reprogramming.

The last part of Enix is the host PC's *utility tools*, including a wireless reprogramming and debugging shell, a file system parser shell, complier and linker. By use of the host PC assistant, the complicated works can be done on the host PC, thus reducing the run-time overhead of the tightly constrained devices. For example, rather than processing delayed linking at run-time as ELF does, the ELIB building tool constructs a position-independent library at link-time. Therefore, the run-time loading burden on the MCU of these compact sensor nodes is reduced significantly, and no additional code is required. Another example is the file system shell, called the *EcoFS shell*, which provides an interactive environment for users to manipulate specially formatted data of EcoFS simply and conveniently. This assistance saves the user from wasting time on troublesome tasks such as reading, modifying and writing secondary storage devices directly by firmware executing on the sensor node using either Serial Serial Peripheral Interface (SPI) or Inter-Integrated Circuit (I^2C) drivers.

3.2 Design Concepts

The primary objectives of Enix is to design a lightweight and dynamic operating system. In this section we will explain the design concepts and novelty of Enix.

3.2.1 Cooperative Threads

To get eliminate the tedious, unstructured, and repetitive event-driven programming shortcoming, multi-threaded programming has been chosen by recent WSN OSs because of the more maintainable source code and modest learning curve [4, 8, 20, 36]. A multi-threaded program requires scheduling support so that the critical threads may be assigned sufficient CPU time to ensure their proper execution. Although multi-threading is often thought of as a technique for improving performance, the runtime overhead of context switching cannot be ignored especially for tiny embedded sensor devices. In fact, most of WSN applications with a multi-threading model do not require the ability to preempt other threads. Moreover, they regard these preemptions as bugs in the program. Cooperative threading is another class of multi-threading environments that emphasize the expression of structure using threads and guarantees that no threads will will be preempted unexpectedly [20]. The programmer designates well-defined points where a thread yields control voluntarily, so that the special work of the thread is guaranteed to be done before other threads get to execute. As a result, this model provides an easy-to-use threading structure that is applicable to event-based WSN applications with little runtime overhead. Enix chooses the cooperative threading model instead of event-driven or preemptive multi-threaded programming model.

Enix also builds a scheduler for the cooperative threads to select the next running thread among all runnable threads. The algorithm to choose the next running thread is not hardwired; it is replaceable with a variety of policies. Enix currently supports priority-based and round-robin scheduler that best suits different WSN applications. Thread control functions such as semaphore, suspend, resume, and sleep are also included in Enix in order to enable synchronization between threads. Instead of using a C-style switch statement to implement cooperative threads as protothreads does, the implementation idea of Enix cooperative threads comes from the setjmp/longjmp system library for enabling non-local jumps. The *setjmp* and *longjmp* system functions are the common method to jump between subroutines. *setjmp* function stores the program counter and stack pointer into a jump buffer; while *longjmp* is called in another subroutine, the data in the jump buffer will be restored thus return to the *setjmp* point. While traditionally each setjmp/longjmp is a pair and the jump direction is fixed from longjmp to setjmp, the cooperative thread in Enix allows jumping from any specific points to another. This can be treated as multi-level setjmp and longjmp. Although some machine dependent assembly code is used, the performance is promoted significantly.

3.2.2 Dual-Mode Operation

Instead of executing a monolithic binary image that is composed of the OS kernel and the user program, ENIX supports dual-mode operation, where the OS kernel and user program are separated. The OS kernel consists of critical modules and hardware drivers and provides a kernel API for user programs, namely the system calls, such as cooperative threads control, virtual memory manager, firmware update module, file system, and RF transceiver drivers; where the user program does the WSN sensing and transmission tasks via cooperative programming model. This dual-mode operation helps implementing run-time support that is highly useful to wireless sensor devices. To reduce the overhead from runtime system call when user program invokes kernel functions, a bridge library is applied to relocate the calls to real addresses at link time. Using this compiler and linker assistance technique, the separated OS kernel and user applications are integrated.

3.2.3 ELIB Design Concepts

The design concepts of ELIB is the separation of user programs and libraries. We discovered that a typical WSN application is composed of a logical structure of the main function and several library function calls. A simple sense-and-transmit application is shown in Figure 3. The program occupies about 1.4 KB of the code memory. However, the main function consumes only 224 bytes compared to the called library functions, which occupy a much larger part of the application's code image. Besides, it is worth mention that these library functions are well designed and modified rarely. The purpose of ELIB is to utilize the characteristics mentioned above. It is the collections of commonly used library functions with special code modification mechanism in order to construct position-independent segments. This mechanism replaces all position-dependent instructions to the kernel functions with fixed address in order to constructs run-time position independent code. By installing these preprocessed segments on the Micro-SD card with unique virtual addresses individually, software virtual memory can be achieved at runtime. Moreover, this design may reduce the transmission size and energy consumption when doing wireless firmware update. Only the main logical structures, that is, the user-defined functions are demanded to be transmitted to remote sensor nodes with ELIB installed.

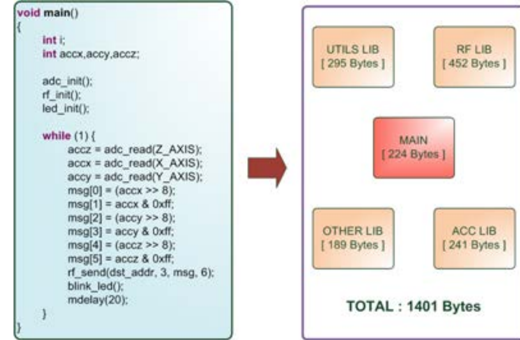


Figure 3. A typical WSN application and the code size of each component.

3.2.4 EcoFS Design Concepts

The Enix operating system has a built-in file system called EcoFS customized for WSN that can support most of the requirements for a tiny wireless embedded sensor system. EcoFS is the first WSN file system that uses the Micro-SD card as its storage medium. The tiny physical size ($1.1cm \times 1.5cm$), large capacity (above 2GB) and simple access interface (4-wire SPI) are the reasons for choosing the Mico-SD cards. Although the MCU in the Micro-SD card consumes additional energy, our experimental results have shown that it saves power to retrieve and store data in an SD card compared with reliable transmission using RF in Chapter 6. Unlike general-purpose file systems that consume a considerable amount of memory for data structures, EcoFS is separated to four storage types by the usage pattern of WSN. With the specific design of each type, EcoFS can modulate the trade-off between in-memory buffer size and improve the performance when designing a file system. The storage types of EcoFS are *code data*, *preferences*, *sensed data*, and *network data*. In order to reduce the in-memory data structures, each data item in EcoFS is either fixed length or wrapped by special tags just like regular TCP/IP packet format and has a special length field to indicate the size of the item. Therefore, parsing process can be done without consuming large data memory by using load-partial-then-parse scheme. A super block is built in the header block of the Micro-SD card for quickly enumerating or searching. For example, a bitmap for network data and the address list for code data enumerate the directly connected node IDs and the addresses of exist code segments, respectively. For the preference type that requires fast search, the super block preserves a hash table for constant-time preferences locating.

Due to the replaceable characteristic of the Micro-SD card, we design an interactive shell EcoFS Shell on host PC OS to provide an easy and simple way to operate EcoFS formatted data instead of handling everything on sensor nodes. EcoFS Shell enables full functionality for handling the EcoFS formatted data including basic read and write operations and advanced sensed data analysis. By putting the full and complex operations on the host side and implementing the demanded one on the sensor node, the consumption of code and data memory for running EcoFS is reduced.

4 Runtime Components in Enix

This chapter describes the key runtime components in Enix: the cooperative threads with the run-time scheduler, compiler-assisted virtual memory, and dynamic loading.

4.1 Cooperative Threads and Scheduler

The *cooperative threading* model has the characteristic that a context switch occurs only when current running thread calls yield or sleep functions; thus, the overhead of context switching and stack usage are lower than preemptive multi-threading and is more appropriate for tightly constrained wireless sensor platforms. In this section, we describe a novel way to implement cooperative threads with light context-switching overhead and that consumes little code and data memory. In addition, two popular scheduling policies, namely priority-based and round-robin, are also presented to provide the adaptive abilities of Enix for supporting different WSN applications.

4.1.1 Multi-points Setjmp/Longjmp

The cooperative threading model, also known as as coroutines, is based on the idea by Donald Knuth. Coroutines allow multiple entry points in subroutines for suspending and resuming execution at certain locations. The purpose of cooperative threads in Enix is to achieve coroutine-like behavior for resource constrained wireless sensor systems, through an efficient, low-cost implementation of coroutines.

The *setjmp* and *longjmp* are common system functions for jumping between subroutines. To achieve the inter-subroutines jump, the *setjmp* function stores the program counter and stack pointer into a jump buffer. When *longjmp* function is called from another subroutine, the data in the jump buffer is restored, and then the program returns to the previous *setjmp* point. It is worth mentioning that the used registers are pushed on the stack before the *setjmp* function is called, and therefore the local variables can also be re-

stored after *setjmp*. However, this approach does not support jumping forward or backward between multiple functions as required by coroutines. First, it provides a single-direction jump only from the *longjmp* point to *setjmp* point according the jump buffer; second, stack overwriting happens while the previous *setjmp* point calls the cascaded function that may modify the stack data of later *setjmp* points.

To achieve cooperative threading, the ideas of *setjmp* and *longjmp* are taken; each never-returned subroutine represents a cooperative thread, which has its own stack and context buffer for storing critical data. The cooperative thread may yield at any specific point to invoke the scheduler and resume another cooperative thread; each yielding call automatically pushes the used local variables on the stack and records the program counter and stack pointer in the context buffer; the resume process simply restores the saved data from the context buffer and the stack. As long as the context-switching point is determinable, only the necessary data would be stored on the stack. Therefore, the per-thread stack does not require large capacity and is adaptive according to the number of threads. The maximum number of cooperative threads is seven in the current version of Enix. This approach to cooperative threads allows subroutines to suspend and resume execution at specific locations without being concerned with stack and thread states. Moreover, the functions such as *semaphores*, *yield*, *sleep*, *suspend* and *resume* are also provided to enable thread control abilities.

Figure 4 shows a sample Enix user application that applies the cooperative threading model. The purpose of this application is to sense the accelerations of three axes, namely X-axis, Y-axis and Z-axis, and then wirelessly transmit the sensed data to a remote sensor node with ID 1234. There are three cooperative threads in this program: the thread THREAD0 first initializes the hardware modules and global variables and then blinks the LED periodically; the thread THREAD1 senses the data while the sensor is ready and the previous sensed data is already transmitted; the last thread THREAD2 transmits the sensed data to the remote sensor node and then clears the global flag to allow the next sensing task in THREAD1. The main function adds the threads to Enix kernel and then calls *enix_kernel_start* function to invoke the scheduler.

4.1.2 Priority Based and Round-Robin Scheduler

As mentioned before, the sample code in Figure 4 finally calls *enix_kernel_start* function at the end of the main function to invoke the scheduler, and the function *enix_kernel_start* never returns to main(). The scheduler decides the next running thread from the list of runnable threads and processes context switching.

Enix provides two scheduling policies: priority-based scheduler and round-robin scheduler to determine the next running thread by priority or registration sequence, respectively. The scheduler policy in Enix is replaceable to provide flexibility for development of WSN applications. The list of runnable threads is represented with a bitmap, which enables the next runnable thread to be found efficiently The thread with priority n is runnable only if the nth bit in the bitmap is set; thus by checking the bitmap, the next running thread can be found. For the priority-based scheduler, the first set

```
#include <elib/epl_utils.h>
#include <elib/epl_uart.h>
#include <elib/epl_acc.h>
#include <kernel/enix_kernel.h>

extern xdata char* malloc_ptr_1;
extern unsigned char gb;

//thread 0 control LED and init everything
ENIX_THREAD(thread0) {
    EA = RF = 1;  //init RF
    //init 3 axis
    epl_acc_init(ACC_8G_SCALE,
                 ACC_DATA_RATE_100HZ);
    //malloc 3 bytes
    malloc_ptr_1 = (xdata signed char *)
                 eco_kernel_mem_req(3);
    gb = 0;  //init flag
    while(1) {
        LED0 ^= 1;
        LED1 ^= 1;
        enix_kernel_thread_sleep(10);
    }
    ENIX_THREAD_END();
}

//thread 1 sensed data from 3 axis
ENIX_THREAD(thread1) {
    while(1) {
        if(gb || !epl_acc_data_is_ready())
            break;

        malloc_ptr_1[0] = epl_acc_read_X();
        malloc_ptr_1[1] = epl_acc_read_Y();
        malloc_ptr_1[2] = epl_acc_read_Z();
        gb = 1;  //set flag
        enix_kernel_thread_sleep(1);
    }
    ENIX_THREAD_END();
}
```

```
//thread 3 transmit sensed data
ENIX_THREAD(thread3) {
    pdata unsigned char *packet;
    packet = enix_kernel_get_tx_buf();
    enix_kernel_rf_start_tx(1234);

    while(1){
        if(gb){
            packet[0] = malloc_ptr_1[0];
            packet[1] = malloc_ptr_1[1];
            packet[2] = malloc_ptr_1[2];
            enix_kernel_rf_send();
            gb = 0;  //clear flag
        }
        enix_kernel_thread_sleep(1);
    }
    ENIX_THREAD_END();
}

int main() {
    LED0 = LED1 = OFF;

    //add thread
    enix_kernel_add_thread(0, ENIX_DEFAULT_INIT,
        thread0, LOW_POWER_ON);
    enix_kernel_add_thread(1, ENIX_DEFAULT_INIT,
        thread1, LOW_POWER_OFF);
    enix_kernel_add_thread(3, ENIX_DEFAULT_INIT,
        thread3, LOW_POWER_OFF);

    //run kernel, never return
    enix_kernel_set_timer_period(0);
    enix_kernel_start();

    return 0;
}
```

bit in the bitmap indicates the next running thread. Most of the powerful architectures such as ARM support a simple instruction to find the first set bit in a 32-bit word. The limited wireless sensor devices with the lightweight MCU like 8051, 8052, AVR, MSP430 and so on do not support such powerful instructions; so, a table-lookup implementation to find the first set bit in a bitmap is applied. Algorithm 1 shows the pseudo code for finding the highest-priority runnable thread by table lookup; *nextPrioTbl* is a byte array with 16 elements, each of which indicates the number of the first set bit according the index of the array; *rdylst* is a bitmap list. Each bit represents a cooperative thread with a different priority; by looking up the number of the first set bit, the next running thread can be found as shown in the algorithm. To implement a round-robin scheduler, we propose another novel, efficient table-lookup algorithm, whose pseudo code is shown in Algorithm 2. By rotating the *rdylist* to the right for *currentPrio*+1 bits and looking up the table, which is the same as Algorithm 1, the next running thread can be easily found. These two algorithms consume additional 16 bytes of code memory for the immutable table but reduce the total code size and improve the performance significantly, as will be shown in Chapter 6.

Algorithm 1 Fast algorithm to get the next running thread (Priority-Based).

```
if nextPrioTbl[ rdylst & 0xFF ] ≠ 4 then
    return nextPrioTbl[ rdylst & 0xFF ]
else
    return 4 + nextPrioTbl[ rdylst >> 4 ]
end if
```

Algorithm 2 Fast algorithm to get next running thread (Round-Robin).

```
if rdylst = 0 then
    return 7
end if
t ← rdylst RR (currentPrio + 1)
r ← bitmap_lookup( t )
x ← r + currentPrio + 1
if x > 7 then
    return x - 8
else
    return x
end if
```

4.2 Compiler-Assisted Virtual Memory

Virtual memory is widely used in operating systems to support larger a memory space than provided by the physical memory. In this section, we describe the implementation details of software virtual memory in Enix.

4.2.1 Demand Segmentation

Virtual memory is achieved in Enix via demand segmentation without any hardware support. The code memory of a wireless sensor device is divided into swappable and non-swappable areas. The Enix kernel and the user-defined logical structures such as the main() function are non-swappable. The swappable area utilizes the the rest of code memory managed by the virtual memory manager of Enix. In order to reduce the runtime overhead of Enix, a library called ELIB is proposed. ELIB consists of binary code segments that are preprocessed on the host PC and is pre-installed on the Micro-SD card, the secondary storage medium natively supported in Enix. Each segment in ELIB represents a function binary code that is runtime position-independent; a unique virtual address is assigned for each code segment to indicate the segment location in the Micro-SD card. Calls to ELIB functions from the user program will be translated at compile time into calls to a special run-time loader routine in Enix kernel using a technique called *source code refinement*, to be described in the next section. Therefore, the demanded segments would be loaded into code memory and executed at run-time. The current memory allocation scheme in Enix is first-fit.

Figure 5 shows the procedure to build ELIB. It takes three passes to construct ELIB. In the first pass, the common functions are collected into a file named *ELIB.LIB*, which is passed to the library parser in order to get the binary code size of each function; and then a *virtual-address allocator* is called to allocate a unique virtual address to each function. The second pass is *code modification*: a library function may call another library function that does not exist in code memory, and therefore such code must be modified. The purpose of code modification is to translate ELIB functions to run-time position-independent code. When the code modification is done, the ELIB is compiled and linked to create the file named *ELIB.HEX*, the hex image that consists of the HEX representation of ELIB functions. The final pass splits *ELIB.HEX* into separated HEX files, each of which is called a *code segment*, that is, the HEX representation of a function. Next, the EcoFS install program is called to install every code segment onto the Micro-SD card according to their virtual addresses. The procedure above can be done automatically by a shell script; after this procedure, the Micro-SD card is available for loading and executing.

4.2.2 Memory Compaction and Garbage Collection

All virtual-memory systems have the common problem of *fragmentation*. *External fragmentation* occurs in demand segmentation system when the free memory blocks are not consecutive. The specialized dynamic loading library ELIB of Enix is runtime position-independent, therefore, fragmentation can be solved by memory compaction. There is a garbage collector routine that executes periodically to observe the memory usage and collects the memory if necessary; it will release the least recently used (LRU) memory and do the compaction of the frequently used segments.

Another problem arises when the garbage collector reclaims those segments that will be executed after the return of current running segment. This is called the *cascaded call* problem: it occurs when the code memory is out of use, and the caller is garbage-collected while the callee is executing, such that the callee returns to a caller that has been swapped out, thereby causing the system to crash. There are some solutions to fix the cascaded call problem. For example, additional checking code can be inserted before callee returns, and thus the absent caller would be reloaded back before it

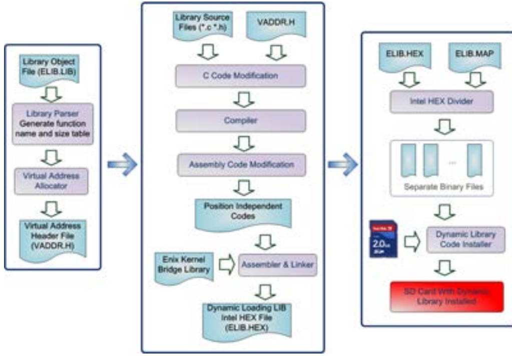


Figure 5. Procedure of compiler assistant dynamic loading library.

is returned to. Enix solves this problem by restricting the garbage collector to only non-swappable code.

4.3 Dynamic Loading and Run-time Reprogramming

Run-time reprogramming and dynamic loading are important issues in WSN OS design. The deployed sensor nodes may provide remote programming ability for the purpose of bug fixing and firmware updating. Enix supports run-time reprogramming and dynamic loading; the user program can be updated through RF. To achieve dynamic loading, Enix uses position-independent code; the binary program can be moved anywhere without any relocation. To update the user program, the source code refinement technique is applied. We describe these topics in the following subsections.

4.3.1 Run-time Position-Independent Code

To achieve runtime loading, Enix uses PIC approach. Traditional PIC is machine dependent, and thus not every architecture supports PIC. We propose a novel way to generate PIC code without hardware support; with the assistance of run-time kernel via code modification, the code becomes position-independent at run-time. This modification is also applied to the function segments in ELIB as mentioned before. Figure 6 shows the code modification details in Enix. Three types of code are position dependent in general Enix user program. First, the kernel calls in the user program: due to the feature of separated kernel and user programs, the kernel calls do not have to be modified; the linker redirects the kernel calls in user program to the right addresses via a linker script as shown in Figure 7(b). The second position-dependent type is the library calls; we redirect this kind of absolute calls to the special kernel function via code modification; the kernel function finds the address of target function at run-time and then jumps there. For the target that does not exist, the loader is invoked to do the same work as mentioned in the virtual memory section. The last type is the local absolute jump. In most cases, the local jumps are relative jumps except for the jump from the begin of a large logical structure to the end. We modify the long jump instruction to a relative jump routine by calculating the relative size from source to the target at compile time, get the current program counter at run-time, and then sum the program counter and the relative size to get the target address at run-time. In addition, Figure 6 also shows two key functions in the Enix kernel to support run-time position-independent code. The function $enix_getpc$ gets the program counter at run-time and stores it to *DPH1* and *DPL1* registers since the *lcall* instruction will push the return address onto the stack, we can read the program counter from the stack and save it into the the registers mentioned above. $enix_fake$ is another kernel function that checks the existence of the target function and redirects the *lcall* instruction to the target function address at runtime. If the target function does not exist in the code memory, then it will be loaded from the Micro-SD card according to the virtual address passed into $enix_fake$ function.

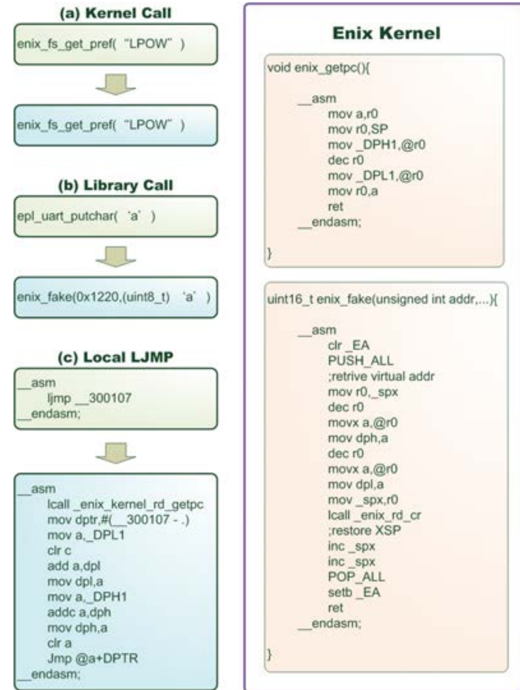
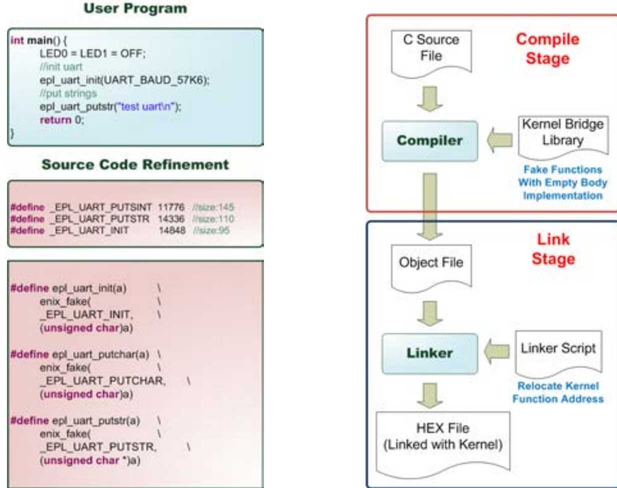


Figure 6. Run-time support position independent code.

4.3.2 Source Code Refinement

In order to hide the details of the run-time loader in user programs development flow and to reduce the amount of code modifications, the source code refinement technique is applied. Figure 7(a) shows a simple Enix user application sending a string of data through UART. For the include files of Enix user program, the function definitions are removed, and C macros are applied as shown in the figure 7(a) By use of C macros and the *vararg* of C language, each library function call in the user program is redirected to a special function $enix_fake$. This *vararg* function allows various numbers and types of parameters, and therefore every function prototype can work with this refinement with additional type casting. Further, the source code refinement technique can be used to achieve system configuration. For example, a general library function *SendPacket* can be called by the user program to send a byte buffer through different interfaces, such as RF, UART, I2C and SPI. It's very easy to provide a configuration interface for users to choose the transmission in-



(a) Source code refinement. (b) Bridge library.

Figure 7. Bridge library and Enix user programs.

terface of *SendPacket* library function by use of source code refinement technique. Hence, different hardware modules of wireless sensor devices can be easily configured.

5 EcoFS : The File System

An lightweight and efficient storage system is essential to ultra-compact sensor nodes that must log all sensed data for later analysis or asynchronous transmission across multi-hop sensor nodes. Nonvolatile storage is also important for recording node states and time stamps of events, especially since power depletion may occur unpredictably on a deployed sensor node. Although the developer can write drivers to directly control the nonvolatile storage, doing so is tedious, error prone, and unstructured. A more structured approach is to build a simple file system abstraction on top of the raw storage device. In this Chapter we describe the file system called EcoFS as a component in Enix. First, we discuss the storage medium for the file system, and then we describe the implementation details.

5.1 Storage Medium in WSN

In recent years, flash memory has been widely used in embedded systems and handheld devices. The characteristics of flash memory include non-volatility, small size, low cost, low power consumption, and shock resistance. In fact, flash memories are equipped in popular wireless sensor platforms such as Mica2, MicaZ and Telos. It can also be added as an expansion module on other wireless sensor platforms. For example, our experiment platform EcoSpire can have an external Micro-SD card module connected via SPI.

Table 3 summarizes the features of different flash memories frequently used in embedded devices. There are two main categories of flash memory: NAND type and NOR type, which are classified by the type of the circuit that holds a single bit. The internal structure of NOR flash is less dense than NAND because NOR uses more logic per bit, and therefore the capacity is limited. For the power consumption, NAND flash consumes less power than NOR flash; SD card consumes more power than raw flash components because

Table 3. Comparison between flash memories.

Flash Memory	SD/MMC	Serial Flash	NAND Flash	NOR Flash
Memory Cell	NAND	NOR	NAND	NOR
Density	High	Low	High	Low
Capacity	High	Low	High	Low
Power Consumption	High	High	Low	High
Intrinsic	page-oriented	page-oriented	page-oriented	word-oriented
Interface	SPI	SPI	Parallel	Parallel
Switchable	○	×	×	×
Cost	2GB 6.5 USD	512KB 1.5 USD	2GB 3.5 USD	128MB 12.5 USD

a controller inside handles wear leveling, auto-erasing, and error correcting codes (ECC) recovering. The word-oriented NOR flash memories are used mainly for code memory, because it provides fast random access ability like static RAM except for erasing, that is, it receives an address on address bus and outputs a data word on data bus. In contract, the page-oriented flash memory operates a page of data by commands and responses, thus resulting slower access time comparing with NOR flash. NOR and NAND flash use parallel interface that requires many GPIOs which is impossible for tiny embedded device with less than five GPIOs. Serial flash is a type of NOR flash memory with serial interface therefore it is page-oriented It is commonly used in tiny embedded system due to the reduced GPIO characteristic of MCU. The drawbacks of serial flash include small capacity which is usually less than 512KB and the non-removable property. SD/MMC card is another flash memory that can be controlled through serial interface. Although Micro-SD card consumes more energy than other flash memories, the simple interface, small physical size, low cost and high capacity characteristics make it suitable for wireless embedded devices. The in-card controller provides NAND flash memory management thus reducing the complexity and memory consumption of wireless sensor devices. Furthermore, the removable characteristic enables the portable SD card to be removed from deployed sensor nodes for later analysis instead of transmitting the logged data back through UART, USB, or RF. According to the advantages mentioned above, SD/MMC card is chosen by EcoFS as storage medium.

5.2 Implementation Details of EcoFS

The implementation of EcoFS consists of two main parts: node side and host side. The part on the host side is also called EcoFS Shell. Due to the resource limitations of tiny wireless sensor devices, the node-side implementation focuses on how to efficiently access EcoFS data with relatively small data memory consumption compared to other file systems for WSNs. On the other hand, the host-side implementation provides complete functionalities of EcoFS, including list, read, write, modify, and binary to HEX translation. There is no severe restriction on the host PC, and therefore the EcoFS Shell focuses on the convenient file handling and user friendly interface.

Figure 8 shows the block diagrams of EcoFS for both the node side and host side. For the wireless sensor node implementation shown in Figure 8(a), an SD/MMC driver is built according to the Secure Digital Card specification that uses SPI protocol to access memory card for basic I/O operations. An EcoFS library is provided in order to recognize specialized data types of EcoFS namely code data, prefer-

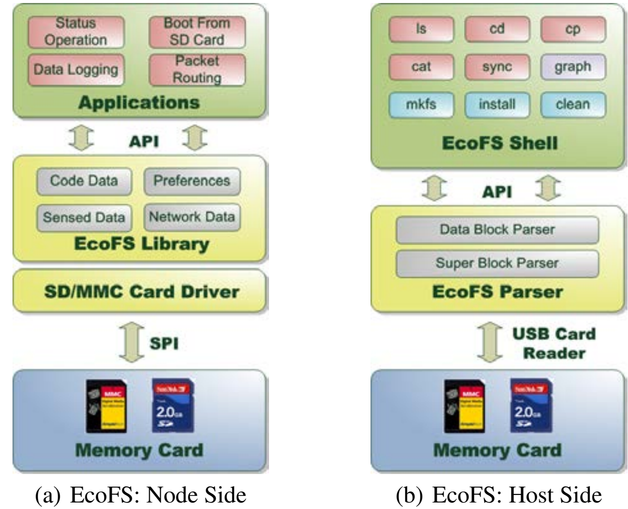
ences, sensed data, and network data. The EcoFS Library is configurable; only the demanded ones are configured and installed on the wireless sensor nodes. By invoking API functions provided by EcoFS Library, the WSN applications can access the EcoFS-formatted SD/MMC card easily as long as the specific library is installed on sensor nodes. Some reference applications commonly used in WSN such as logging sensed data, booting from SD card, periodically refreshing node status, and accessing routing table in multi-hop wireless networks, can all be achieved easily by using EcoFS. Unlike the SPI protocol that used in sensor devices, with the assistance of host PC OS such as Windows, Linux and Mac OS, a USB card reader can be used to access memory card as shown in Figure 8(b). In order to extract EcoFS-formatted data, an EcoFS Parser is required. The raw data read from the card are processed by the parser that converts them into the appropriate file format to be accessed by users. The top layer of EcoFS host-side implementation is EcoFS Shell. The interactive shell environment provides general Unix-like commands such as `ls`, `cd`, `cp`, and `cat` so that users can list the files for each type and see the content of files. There are also special commands, for example, `mkfs` to format an SD card, `clean` to eliminate all files, and `install` to modify and add files to EcoFS. Moreover, the command `graph` can be used to analyze sensed data and statistical chart.

To demonstrate the convenient shell environment of EcoFS host side implementation, Figure 9 shows the snapshots of EcoFS Shell. The users can easily manipulate the EcoFS files by plugging the Micro-SD card to a card reader via the assistance of EcoFS Shell program on the host PC. The collected data by the sensors can be analyzed easily as shown in Figure 9(a). In addition, all the EcoFS-formatted data can be accessed with ease through EcoFS Shell. Figure 9(b) shows the HEX representation of a binary code segment in a Micro-SD card, where the shell enables users to check and debug without any extra effort.

As mentioned in Chapter 3, each data item in EcoFS is either fixed length or wrapped by special tags as used in a regular TCP/IP packet format and has a special length field to indicate the size of the item. Therefore, the parsing process can be done without consuming large data memory by using load-partial-then-parse scheme. In the following subsections, we describe the detailed format of EcoFS data types.

5.2.1 *Code Data*

The purpose of code data is to provide virtual memory ability for Enix. By using the EcoFS Shell, the dynamic loading library ELIB can be pre-installed on the Micro-SD card for later loading. Each code segment has a unique virtual address, which is the location of code segment on the Micro-SD card. The segment format starts from a special BEG(0xAE) tag followed by a two-byte field indicating the binary segment size; after the size field, the binary data with a pre-defined size are presented. When a node wants to retrieve the segment by its virtual address from the Micro-SD card, it first checks for the BEG tag; then it reads two bytes to get the data size and allocate suitable code memory space; finally, the code segment is loaded from the Micro-SD card into the code memory. The current design of the code data block does not allow modification of the code segment by the sensor node itself; a later version of EcoFS will enable binary segments replacement at run-time.



(a) EcoFS: Node Side (b) EcoFS: Host Side

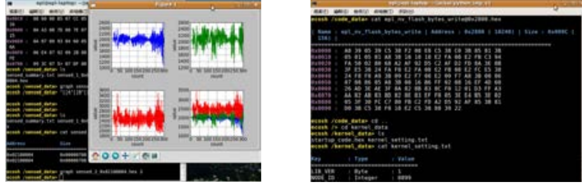
Figure 8. The block diagrams of EcoFS.

5.2.2 *Preferences*

In order to provide a fast query data structure of EcoFS, preference data type is added. The item format of preference is 22 bytes data including 1 byte BEG(0xEA) tag, 1 byte TYPE tag that indicates the type of the value field such as character, integer and string; following are the 10-byte key string and the 10-byte value with a specific type shown before. To prevent high data memory consumption and support modification of data, each preference is distributed to a 512-bytes SD card sector. The characteristic of preference is to support fast searching and additional modifying ability. As a result, a hashing scheme is applied. When the specific value of a key is required, first the key is passed to a simple hash function to generates an integer; second, the integer is added to an offset to get the sector number, the location of preference item; finally, the value is retrieved. For the keys with same has value, we allocate five slots to store the collision preferences; the next linked preference is located just adjacent to the current located sector. There is also a super block data of preference type, which is the bitmap used to check whether the target hash value exist or not. Thus the string-comparison time is reduced for the inexistent preference keys.

5.2.3 *Sensed Data*

The sensed data is appended only when changing the “already sensed data” is unnecessary and the modification of flash memory results in great overhead. Some of the WSN sensing tasks collect sensed data when specific events happened, such as the earthquake and tornado. For this reason, the begin and end tags are added to enclose sensed data so that the sensed data are separated. There is also a timestamps field in sensed data format, so that the later analysis can calculate the event-trigger time using the data retrieved from this field. To fill the timestamps field, either Enix system timer or user application granted time can be used. Besides,



(a) Off-line analysis of sensed data.

(b) The HEX representation of binary code segments in EcoFS.

Figure 9. EcoFS host PC shell environment.

if the Micro-SD card is full, then the decision to overwrite or stop depends on the preference setting.

5.2.4 *Network Data*

EcoFS network data are used to store network related information such as packet routing information, neighbor sensor nodes' states, and the WSN topology. The fields can be customized by user applications. Each network data item is represented by fixed 32-byte data, including 2 bytes of network ID and 30 bytes of a user-defined structure such as the type, state, and remaining power of sensor node, depending on the requirements of the WSN applications. A bitmap is maintained in the super block for the purpose of quickly enumerating existing items in the EcoFS network data area. The current version of EcoFS supports at most 65536 IDs, which consumes 8192 bytes of a bitmap. Even though a sensor node reads 8092 bytes of result in about 12*ms* overhead, this mechanism consumes only 32 bytes of data memory, which is more beneficial especially for constrained wireless sensor platforms. For example, we provide a practical WSN application called "receive and forward" which receives a RF packet, then enumerates all the neighbor nodes' ID and forwards the packet in sequence.

6 Evaluation and Results

This chapter shows the efficiency of the Enix lightweight dynamic operating system. We first discuss our experimental setups, including the hardware platforms and software tools. Second, we present the evaluations of Enix. We compare the context switch overhead of different multi-threaded scheduler implementations as evidence that cooperative thread of Enix has the least overhead. In addition, the code and data memory consumptions of different schedulers are also compared. The power consumption of RF and SD card are also compared to show that it is appropriate to store commonly used code segments on the SD card. Finally, we compare the updated application binary image size with and without Enix during remote reprogramming. The image sizes of the applications processed by VCDIFF tool are also compared accordingly to show that Enix reduces the runtime reprogramming size significantly.

6.1 Experimental Setup

This section describes the hardware platform and software tools that are required for the Enix environment. First, the description of hardware platform called EcoSpire and related modules such as current sensor and battery fuel gauge are presented. Next, the software tools including the Eclipse IDE, RF dumper and USB programmer are also described.

6.1.1 *Sensor Platform*

We use EcoSpire, currently a functional prototyping board for the ultra-compact Eco node as our experimental platform. As shown in Figure 10, EcoSpire is a compact wireless sensor platform including a MCU, RF transceiver, chip antenna, acceleration sensor, power subsystem, and expansion interfaces. A Micro-SD slot and a 32-Mbit on-board flash are also included to provide simple data storage. The physical dimensions of EcoSpire are $23 \times 50 \times 8\text{mm}^3$.

EcoSpire uses the Nordic nRF24LE1 MCU. It is a system-on-chip (SOC) with an 8052-compatible MCU and the nRF24L01+ transceiver. The MCU has a 16 KB on-chip flash as program memory, 1 KB on-chip RAM, 256 bytes of "external" (to the 8052 core) on-chip SRAM, 12-bit ADC, SPI, I^2C, and UART. The nRF24L01+ RF uses the 2.4GHz ISM band and is compatible with the "Low Energy" subset of Bluetooth 3.0 at 2 Mbps on one of 125 (overlapping) frequency channels of 2 MHz bandwidth each. It implements hardware-supported acknowledgement and retransmission and contains six data buffers to receive packets destined for up to six different IDs.

EcoSpire uses the LIS331DL accelerometer with digital output over SPI. It measures acceleration in $\pm$8g range and has low power consumption of less than 1 mW. Other sensors with digital or analog output can also be added via the expansion module interface.

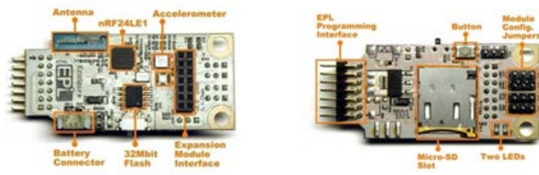
Figure 11 shows the block diagram of EcoSpire. It is worth mentioning the convenience of the hardware expansion module interface of EcoSpire. Extended functions can be added by plugging a module board to the expansion module interface. Several modules have been developed to work with EcoSpire such as pushbuttons, microphone, speaker with amplifier, and LCD. Moreover, more complex modules such as GPS, gyroscope, and camera have also been prototyped.

In order to measure the energy consumption of Enix, the power measurement modules are developed as shown in Figure 13. The current sensor module is used to measure the instantaneous current of another EcoSpire in execution. By setting a suitable sampling rate of current sensor, the energy consumption of EcoSpire can be calculated using Equation 1. In this equation, a total of $(N+1)$ currents are sampled. First, we sum up the product of current C_t and voltage V; and then we divided it by $(N+1)$ thus producing the average power; the product of average power and total time T results in the total energy consumed by EcoSpire in time T. Another power measurement module is the battery fuel gauge, which allows EcoSpire to measure the power and battery capacity itself at run-time.

$$\frac{\sum_{t=0}^{N} C_t \times V}{N+1} \times T \tag{1}$$

6.1.2 *Software Tools*

The software for EcoSpire includes the IDE and graphical user interface tools (GUI) on the host computer, system



(a) Front view. (b) Bottom view.

Figure 10. EcoSpire hardware.

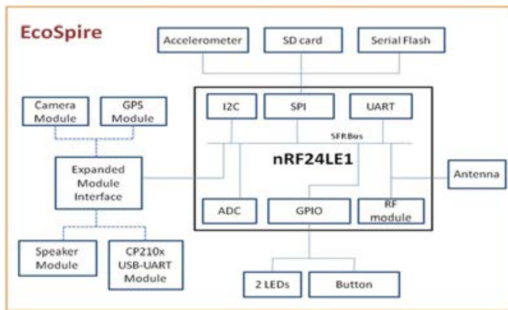


Figure 11. EcoSpire block diagram.

Figure 12. EcoSpire development board.

(a) Current sensor. (b) Battery fuel gauge.

Figure 13. Power measurement modules.

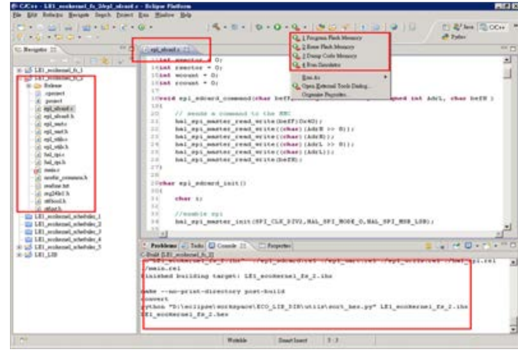


Figure 14. Eclipse IDE.

software on the sensor node and the base station, and utility tools for uploading images and RF debugging.

To build our IDE for EcoSpire, we develop the EcoSpire plug-in for the Eclipse open-source IDE, as shown in Figure 14. It supports source code editing, compiler invoking, and firmware programming. Also, the programming interface is also embedded in Eclipse. By clicking the upload button, the IDE transmits the application HEX image through the USB interface to the development board, and the board starts to program EcoSpire through SPI. The development board is shown in Figure 12.

RF interfaces are difficult to debug due to not only the inherent lossy nature of the wireless medium but also the several levels of abstraction in which errors may occur, including physical, media-access, link, network, etc. Instruments such as spectrum analyzers provide the physical-layer sniffing but are expensive and cannot support higher-level concepts such as header decoding and CRC calculating. To easily debug RF interfaces in development stage, we provide three RF tools: the *RF dumper* and *RF Sender* can emulate the behavior of a sensor node as either sender or receiver; the *RF scanner* scans network addresses in a given range and discovers the available nodes in the network.

6.2 Experimental Results

To evaluate the efficiency of Enix, we develop some test applications to be executed on EcoSpire for comparisons. Some may consider the context-switch of multi-threaded programming models to cause a great deal of overhead and is not appropriate for lightweight sensor platform. We compare the context-switch overhead and resource requirements of different scheduler implementations first to show the efficiency of cooperative threading models. Next, we present the speed and power consumption data of different flash memories to quantitatively show that an SD card makes an appropriate a secondary storage. Finally, we show the reduction of run-time reprogramming size by using Enix comparing with other schemes that also claim to reduce the transmission size of remote reprogramming.

6.2.1 Context Switch Overhead Applied to Different Scheduler Implementations

To evaluate the context-switch overhead of the cooperative threads in Enix, we implement both round-robin (RR) and priority based scheduler with different algorithms and different functionalities that may affect the context-switch

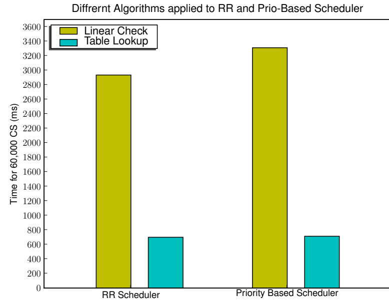


Figure 15. Context switch overhead comparison between different NextRunningThread algorithm.

time. Figure 15 shows the result of using a fast table-lookup algorithm to find the next running thread from the runnable queue. We measure the execution time by running context switch sixty thousand times. It is clear to observe that both RR and priority-based schedulers are improved while applying the fast algorithm. The execution time becomes a quarter of the original linear search implementation. Figure 16 is the comparison of context-switch overhead between different multi-threaded models for both RR and priority-based scheduler. Preemptive multi-threading has the highest context-switch overhead due to the unpredictable preemption time, and therefore all of the registers must be saved and restored during context switch. We implement C-coroutines using C-Switch statements and add the priority-based and RR schedulers to it. C-coroutines and cooperative threads have the feature that context switching occurs only when the running thread calls yield or sleep functions, and thus they have lower context-switch overhead. The implementation of C-coroutines uses C-Switch statements, and this means that every context switch results in several comparisons of variables and an absolute jump. Consequently, a context switch of cooperative threads is simply a replacement of the program counter, stack pointer, and some global variables, and thus it has the lowest overhead. To compare Enix with a real-world OS, μC/OS-II is ported to EcoSpire. Figure 17 compares the context-switch overhead of μC/OS-II and our work. The reason why μC/OS-II has high context-switch overhead is that μC/OS-II uses external memory to store both the per-thread stack and registers, thus producing great overhead from many external memory movements. In addition to the context-switch overhead, the code and data memory sizes are also compared. Figure 18 shows a comparison of the code and data memory between different scheduler implementations. The preemptive one consumes the most memory because of the same reason mentioned before. Other scheduler implementations require about 1KB of code memory, which is frugal compared to other regular RTOSs such as μC/OS-II and FreeRTOS, as shown in Figure 19.

6.2.2 *Efficiency of EcoFS*

This section shows the speed and power consumption of SD cards and other serial flash memories embedded on EcoSpire. The results confirm the reason that the Micro-SD card has been chosen to be the main secondary storage, as

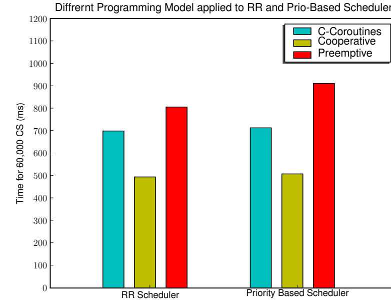


Figure 16. Context switch overhead comparison between different scheduler implementation.

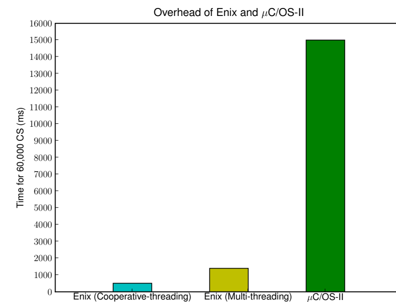


Figure 17. Context switch overhead comparison between Enix and μC/OS-II.

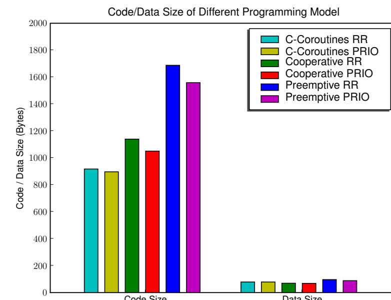


Figure 18. Code and data size comparison between different scheduler implementation.

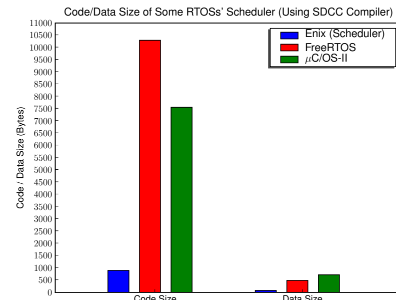


Figure 19. Code and data size comparison between Enix, FreeRTOS and μC/OS-II.

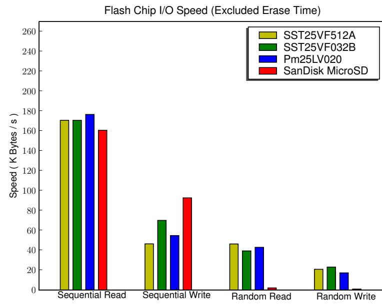


Figure 20. The I/O speed comparison between different flash chips.

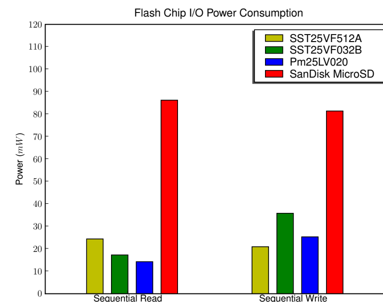
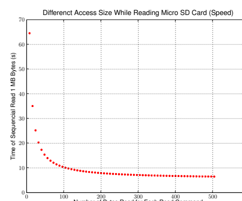


Figure 21. The power consumption comparison between different flash chips.

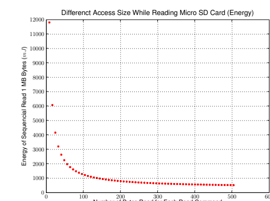
already discussed in Chapter 5.

Figures 20 and 21 compare the speed and power consumption of a Micro-SD card with three other different on-board serial flash memories. These flash memories and the Micro-SD card are connected to EcoSpire through a common SPI bus. The SD card has the fast sequential read and sequential write properties but poor random access speed due to the characteristics of NAND flash memory used by the SD card. Most of the functionalities of EcoFS use sequential reads and sequential writes such as code data and sensed data. For the other preferences types and network data, their access unit is a sector, and therefore the access time is equivalent to sequential access. Thus, the slow random access speed does not affect EcoFS. For the power consumption, the on-card MCU of the SD card causes the highest power consumption among all flash memories. In fact, the data shown in Figure 21 is the active power consumption of the SD card, that is, when the chip-select signal is asserted. When the chip-select signal is deasserted, the power consumption of the SD card is low. Accordingly, the appropriate usage of the SD card may reduce the total energy cost of the sensor nodes. EcoFS is designed according to this requirement: once the SD card is selected by the chip select signal, the I/O operations should be finished as soon as possible.

In addition to the comparison data for speed and power consumption, some other SD card-specific experiments are also presented. Figure 22 shows the time and energy cost of the sensor node performing 1MB of sequential read with different block sizes. Due to the requirement of a start command before each sequential read and sequential write operation, the highest performance and lowest energy cost happen while the maximum block size, 512 bytes is applied. The design of EcoFS tries to use the largest possible block size and reduce the requirement of random-access operation in order to overcome the power and speed bottleneck of the SD card. Owing to the different approaches by SD card manufacturers, we have tried seven 2GB SD cards made by different manufacturers. We measure the speed and power consumption of sequential read and sequential write operations shown in Figure 23. It is very clear to see that the manufacturer SanDisk performs the best in every speed and power conservation. Accordingly, the SanDisk Micro-SD card is chosen for Enix.

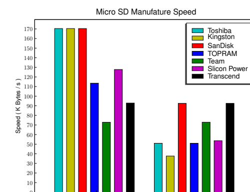


(a) Speed

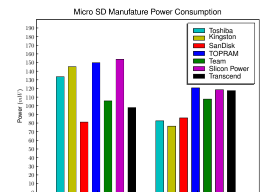


(b) Power Consumption

Figure 22. Read SD card data using different block length.



(a) Speed



(b) Power Consumption

Figure 23. SD card comparison between different manufacture.

6.2.3 *Efficiency of Enix Code Update Scheme*

The key concept of Enix is the separation of user-defined logical structures and commonly used library functions. By storing the dynamic loading library ELIB into the Micro-SD card, only the user-defined logical structures should be remotely programmed through the RF. Hence, the number of RF packets for run-time reprogramming is reduced.

In this section, we develop five WSN applications compiled with and without Enix. The first three applications are general WSN tasks: (1) sense data and then transmit back to base station through RF, (2) sense data and then log the data onto the Micro-SD card, (3) receive the RF packets and then forward to other sensor nodes through RF. The other two applications are EcoNet applications. EcoNet is a simple multi-hop network composed with several EcoSpire sensor nodes; the unique ID and the adjacent nodes are all recorded on the Micro-SD card of each sensor node; every sensor node can invoke the EcoFS API to enumerate its adjacent sensor nodes. The fourth application, EcoNet Transmit, will collect the sensor data with a random number and transmit to the neighboring sensor nodes by enumerating the network data block of EcoFS. The last application EcoNet Receive receives the sensed data from neighboring sensor nodes and checks the duplication of random numbers of sequential RF packets; the valid packets will be forwarded to base station.

Table 4 compares the uploaded image sizes of the above five WSN applications with and without Enix. It is clear that the WSN applications without any OS support have binary image size larger than 6KB on average. Due to the large code size of the RF library, only the second application has a code size of less than 3KB. By comparing the same applications that use Enix as operating system, the size of the program images that required to transmit through RF are reduced significantly. These applications produce 500 bytes of binary image on average except for the fourth application EcoNet Transmit, which generates random numbers without calling any kernel or ELIB functions, and therefore it produces about 1KB program image. Most of the run-time reprogramming schemes use the VCDIFF tool to generate the patch between two binary code images, and the patch will be decompressed by the sensor node. Table 5 shows the results from running VCDIFF for each of the two different WSN applications mentioned above. Although the average binary images size is reduced to 4KB, it is still larger than the applications with Enix.

Figure 24 compares the energy cost by 1MB data transfer with SD card and RF as transmission medium. As the figure shows, the reliable RF transmission and reception consumes the highest energy while doing 1 MB data transfer. Thus it is more power saving to store the reliable binary code on an SD card and then load on demand instead of transmitting through RF. In short, by applying Enix as the operating system, sensor nodes preserve energy and time while becoming capable of efficient remote reprogramming due to the reduced binary image size.

Table 4. Run-time reprogramming code size with/without Enix.

application	Sensing & RF Transmit	Sensing & Log	RF Transmit & RF Receive	EcoNet Transmit	EcoNet Receive
Binary Size without Kernel	4825	2903	7646	8233	7770
Binary Size with Kernel	474	673	514	1027	442

unit : bytes

Table 5. Update code size using VCDIFF delta compression.

Xdelta -9	Sensing & RF Transmit	Sensing & Log	RF Transmit & RF Receive	EcoNet Transmit	Eco Rec
Sensing & RF Transmit	X	3148	2834	2663	28
Sensing & Log	2920	X	1937	1921	19
RF Transmit & RF Receive	4509	4730	X	3168	32
EcoNet Transmit	4927	5201	3760	X	38
EcoNet Receive	4523	4753	3246	3291	X

unit : bytes

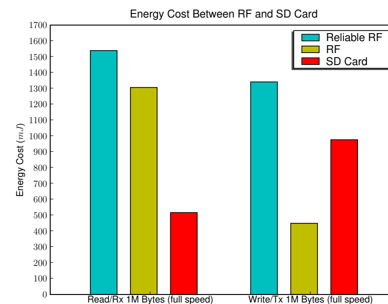


Figure 24. The power consumption comparison between SD card and RF.

7 Conclusions and Future Work

7.1 Conclusions

We propose Enix, a lightweight dynamic operating system for tightly constrained wireless sensor platforms. Enix supports the cooperative threads programming model, dual mode operations, run-time loading, demand segmentation, remote reprogramming, and a lightweight file system EcoFS.

Enix makes five contributions in the WSN OS area. First, the cooperative threads programming model enhances the performance of multi-threaded programming by decreasing the context-switch overhead with multi-points *setjmp/-longjmp* implementation; it is two times faster than the traditional preemptive multi-threaded programming model. This cooperative threading model is easy to learn compared to the event-driven model and, moreover the local states can be saved and restored automatically while the event-driven model and protothreads cannot. Second, Enix provides code virtual memory to overcome the shortage of the on-chip code memory via host-assisted demand segmentation, which most of the WSN OSs do not support. To achieve virtual memory, the ELIB is built on the host PC composed of PIC segments with a unique virtual address for each PIC segment and is loaded to code memory on-demand by the run-time loader of Enix. This PIC approach can reduce the run-time overhead of wireless sensor devices. Third, remote reprogramming is also available in Enix. According to the observation, a WSN application is separated to user-defined logical structures and commonly used library functions. Due to the pre-stored ELIB on the Micro-SD card, the binary image size required to be wireless updated is reduced significantly during the remote reprogramming stage. Fourth, Enix provides a specialized file system called EcoFS that is customized for WSN applications. By using the Micro-SD card as the storage medium, EcoFS is divided into four configurable parts including code data, preferences, network data and sensed data according to the different usage patterns. The efficiency and memory saving features make EcoFS appropriate for tightly constrained wireless sensor platforms. Besides, a shell for the host PC is also provided to control EcoFS-formatted SD cards, including listing, reading, writing, and modification, thus reducing the difficulty and complexity to access EcoFS formatted data. Finally, the code size and data size of Enix with full-function including EcoFS are at most 8KB and 512 bytes, respectively, which are the smallest compared to other WSN OSs. Only ten percent of code is machine-dependent, while the rest is written in C language, and thus it is easy to port to other wireless sensor platforms.

7.2 Future Work

Enix can be extended and improved in several directions, including more complete network management architecture, more efficient dynamic memory allocation mechanisms, and the flexibility to apply EcoFS to different storage media.

7.2.1 Network Architecture

With a well-designed network architecture API, users can easily write a WSN application without the knowledge of the RF chip configuration and the MAC protocol that the network applies. Currently, Enix provides a simple network abstraction via EcoFS network data API; routing tables and network topology can be stored on a Micro-SD card and enumerated if necessary. To enhance the network capability of Enix, a layered architecture of Enix network stack should be implemented. From the bottom up, a MAC protocol provides reliable transmission through RF and must have the ability to identify each sensor node in WSN; a dynamic network layer is used to route RF packet to the target sensor node, and therefore implementations of routing algorithms and routing table maintenance are required; finally, a complete set of network API functions provides full control of the network stack in Enix for WSN developers. In addition to the functionalities mentioned above, the network architecture should also care about the low energy cost and light memory consumption requirement of tightly constrained wireless sensor platforms.

7.2.2 Dynamic Memory Allocator

An efficient dynamic memory allocator is required to utilize the inadequate data memory of tiny sensor devices. Through API functions, user threads can request the memory resource dynamically and release it as long as the memory is no longer used. The simple sequential memory allocation scheme is applied to the current Enix version, but it is not efficient. To support fast memory allocation and memory management, some techniques such as slab allocation or buddy system may be implemented. Both code and data memories required for the memory allocator are the key considerations; using a table-lookup method with bitmaps and bit operations may decrease the code and data memory requirement. Furthermore, the data virtual memory should also be implemented in the later version of Enix in order to overcome the shortage of data memory.

7.2.3 Portable EcoFS

Current EcoFS simply supports SD card as the secondary storage medium. Because of the numerous types of flash memories, SD card is not the only choice for the sensor devices; serial NOR flash and parallel NAND flash are commonly used in wireless sensor platforms such as Mica2 and MicaZ. To achieve portable EcoFS, the implementation can be separated into hardware-dependent module and file system module: the file system module uses the API to achieve the functionalities of EcoFS; the hardware-dependent module should be implemented according to the different flash memories in order to provide the formulated API that the file system module requires. As mentioned before, SD card has an MCU inside that handles the wear-leveling, ECC calculation, and auto erasing while the general raw flash memories do not; hence, these functionalities should be implemented for parallel NAND flash memory. For the serial NOR flash, it is relatively simple: the most important issue is the erase-before-write characteristic, which may consume a large buffer to store the temporary data.

8 References

[1] FreeRTOS. http://www.freertos.org/.

[2] MicaZ. http://www.xbow.com/Products/productdetails.aspx?sid=164.

[3] Portable Formats Specification, Version 1.1.

[4] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 289–302.

[5] BAI, L. S., YANG, L., AND DICK, R. P. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (New York, NY, USA, 2006), ACM, pp. 125–135.

[6] BAI, L. S., YANG, L., AND DICK, R. P. MEMMU: Memory expansion for MMU-less embedded systems. *ACM Trans. Embed. Comput. Syst. 8*, 3 (2009), 1–33.

[7] BARR, R., BICKET, J. C., DANTAS, D. S., DU, B., KIM, T. W. D., ZHOU, B., AND SIRER, E. G. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Oper. Syst. Rev. 36*, 2 (2002), 1–5.

[8] BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10*, 4 (2005), 563–579.

[9] CAO, Q., ABDELZAHER, T., STANKOVIC, J., AND HE, T. The LiteOS operating system: Towards Unix-Like abstractions for wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 233–244.

[10] CHA, H., CHOI, S., JUNG, I., KIM, H., SHIN, H., YOO, J., AND YOON, C. 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* (New York, NY, USA, 2007), ACM, pp. 148–157.

[11] CHOUDHURI, S., AND GIVARGIS, T. Software virtual memory management for MMU-less embedded systems. Tech. rep., Center for Embedded Computer Systems, University of California, Irvine, NOV 2005.

[12] DAI, H., NEUFELD, M., AND HAN, R. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, 2004), ACM, pp. 176–187.

[13] DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM, pp. 15–28.

[14] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 455–462.

[15] DUNKELS, A., SCHMIDT, O., VOIGT, T., AND ALI, M. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM, pp. 29–42.

[16] ESWARAN, A., ROWE, A., AND RAJKUMAR, R. Nano-RK: An energy-aware resource-centric RTOS for sensor networks. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 256–265.

[17] GAY, D. Design of Matchbox, the simple filing system for motes. www.tinyos.net.

[18] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not. 38*, 5 (2003), 1–11.

[19] GU, L., AND STANKOVIC, J. A. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM, pp. 1–14.

[20] GUSTAFSSON, A. Threads without the pain. *Queue 3*, 9 (2005), 34–41.

[21] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services* (New York, NY, USA, 2005), ACM, pp. 163–176.

[22] HILL, J., AND CULLER, D. Mica: a wireless platform for deeply embedded networks. vol. 22, pp. 12–24.

[23] KNUTH, D. *Fundamental Algorithms, Third Edition.* Addison-Wesley, 1997, ch. Section 1.4.2: Coroutines, pp. 193–200.

[24] KOSHY, J., AND PANDEY, R. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems* (New York, NY, USA, 2005), ACM, pp. 243–254.

[25] LABROSSE, J. J. *MicroC/OS-II, The Real-Time Kernel 2ND EDITION.* CMP Books, 2002.

[26] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural*

support for programming languages and operating systems (New York, NY, USA, 2002), ACM, pp. 85–95.

[27] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. TinyOS: An operating system for sensor networks. *Ambient Intelligence* (2005), 115–148.

[28] MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[29] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (New York, NY, USA, 2006), ACM, pp. 195–208.

[30] MÜLLER, R., ALONSO, G., AND KOSSMANN, D. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 145–158.

[31] NATH, S., AND KANSAL, A. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM, pp. 410–419.

[32] PARK, C., LIM, J., KWON, K., LEE, J., AND MIN, S. L. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software* (New York, NY, USA, 2004), ACM, pp. 114–124.

[33] PARK, C., LIU, J., AND CHOU, P. H. Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (Piscataway, NJ, USA, 2005), IEEE Press, p. 54.

[34] PARK, S., KIM, J. W., SHIN, K., AND KIM, D. A nano operating system for wireless sensor networks. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference* (2006), vol. 1, pp. 4 pp.–348.

[35] TATHAM, S. Coroutines in c. http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

[36] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2003), USENIX Association, pp. 4–4.

[37] YAMASHITA, S., SHIMURA, T., AIKI, K., ARA, K., OGATA, Y., SHIMOKAWA, I., TANAKA, T., KURIYAMA, H., SHIMADA, K., AND YANO, K. A 15×15 mm, 1 μA, reliable sensor-net module: enabling application-specific nodes. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks* (New York, NY, USA, 2006), ACM, pp. 383–390.

[38] ZEINALIPOUR-YAZTI, D., LIN, S., KALOGERAKI, V., GUNOPULOS, D., AND NAJJAR, W. A. Microhash: an efficient index structure for fash-based sensor devices. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2005), USENIX Association, pp. 3–3.