Hybrid real-time operating system integrated with middleware for resource-constrained wireless sensor nodes

Xing Liu

N° d'ordre : 2472

EDSPIC: 659

UNIVERSITÉ BLAISE PASCAL-CLERMONT II

ÉCOLE DOCTORALE DE
SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FERRAND

Thèse

Présentée par

Xing LIU

Pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

Spécialité: INFORMATIQUE

Hybrid Real-time Operating System Integrated with Middleware for Resource-constrained Wireless Sensor Nodes

---

Soutenue publiquement le 30 Juin 2014 devant le jury :

Directeur de la thèse :

M. Kun Mean HOU	(University Blaise Pascal, France)

Membres du jury :

Mme. Edwige Pissaloux	Rapporteur
M. Haiying Zhou	Rapporteur
M. Alain Quilliot	Examinateur
M. Jean-Pierre Chanet	Invité
M. Christophe de Vaulx	Invité

Remerciements

I would like to gratefully and sincerely thanks to my supervisor professor Kun Mean Hou for his kindness, trust, help and guidance to me during my PhD studies in the LIMOS laboratory. Our supervisor is an excellent person, he has good inner personal characters and demonstrated many good examples to me in the past years.

I would like to thanks to my supervisor professor Chengcheng Guo in Wuhan University, thank you to him for his always support, caring and help to me in the past years.

I would like to thanks to professor Alain QUILLIOT (director of LIMOS), Dr. Christophe de Vaulx, Dr. Jian-jin Li for their efforts of providing a good living and studying environment to us in the LIMOS.

I would like to express my appreciation to all the members of dissertation jury for their valuable time, their suggestions and the feedback to this thesis.

I am also thankful to Chinese government and China Scholarship Council, thanks for the scholarship that is provided to me during my PhD period.

I also want to express my gratefulness to my colleagues and my friends (Hongling Shi, Jing Wu, Xunxing Diao, Hao Ding, Yibo Chen, Bin Tian, Peng Zhou, Muhammad Yusro, Khalid El Gholami, etc.), for their kindness, caring and help to me these years, and also for their contributions to this dissertation.

The special thanks to Clermont, France, the city where I spent more than four years of my youth life here; the city that is harmony, cultural and beautiful. Many appreciations and pleasant past memories are memorized here.

Finally, the special appreciation to all my family, especially to my kind-hearted, fatherly, diligent and selfless parents. Thank you to you for your unlimited love, for all that is.

Résumé

Avec les avancées récentes en microélectronique, en traitement numérique et en technologie de communication, les noeuds de réseau de capteurs sans fil (noeud RCSF) deviennent de moins en moins encombrants et coûteux. De ce fait la technologie de RCSF est utilisée dans de larges domaines d’application. Comme les noeuds RCSF sont limités en taille et en coût, ils sont en général équipés d’un petit microcontrôleur de faible puissance de calcul et de mémoire etc. De plus ils sont alimentés par une batterie donc son énergie disponible est limitée.

A cause de ces contraintes, la plateforme logicielle d’un RCSF doit consommer peu de mémoire, d’énergie, et doit être efficace en calcul. Toutes ces contraintes rendent les développements de logiciels dédiés au RCSF très compliqués.

Aujourd’hui le développement d’un système d’exploitation dédié à la technologie RCSF est un sujet important. En effet avec un système d’exploitation efficient, les ressources matérielles d’une plateforme RCSF peuvent être utilisées efficacement. De plus, un ensemble de services système disponibles permet de simplifier le développement d’une application. Actuellement beaucoup de travaux de recherche ont été menés pour développer des systèmes d’exploitation pour le RCSF tels que TinyOS, Contiki, SOS, openWSN, mantisOS et simpleRTJ. Cependant plusieurs défis restent à relever dans le domaine de système d’exploitation pour le RCSF. Le premier des défis est le développement d’un système d’exploitation temps réel à faible empreinte mémoire dédié au RCSF. Le second défi est de développer un mécanisme permettant d’utiliser efficacement la mémoire et l’énergie disponible d’un RCSF. De plus, comment fournir un développement d’application pour le RCSF reste une question ouverte.

Dans cette thèse, un nouveau système d’exploitation hybride, temps réel à énergie efficiente et à faible empreinte mémoire nommé MIROS dédié au RCSF a été développé. Dans MIROS, un ordonnanceur hybride a été adopté ; les deux ordonnanceurs évènementiel et multithread ont été implémentés. Avec cet ordonnanceur hybride, le nombre de threads de MIROS peut être diminué d’une façon importante. En conséquence, les avantages d’un système d’exploitation évènementiel qui consomme peu de ressource mémoire et la performance temps réel d’un système d’exploitation multithread ont été obtenues.

De plus, l’allocation dynamique de la mémoire a été aussi réalisée dans MIROS. La technique d’allocation mémoire de MIROS permet l’augmentation de la zone mémoire allouée et le réassemblage des fragments de mémoire. De ce fait, l'allocation de mémoire de MIROS devient plus flexible et la ressource mémoire d'un nœud RCSF peut être utilisée efficacement. Comme l'énergie d'un nœud RCSF est une ressource à forte contrainte, le mécanisme de conservation d'énergie a été implanté dans MIROS. Contrairement aux autres systèmes d'exploitation pour RCSF où la conservation d'énergie a été prise en compte seulement en logiciel, dans MIROS la conservation d'énergie a été prise en compte à la fois en logiciel et en matériel. Enfin, pour fournir un environnement de développement convivial aux utilisateurs, un nouveau intergiciel nommé EMIDE a été développé et intégré dans MIROS. EMIDE permet le découplage d'une application de système. Donc le programme d'application est plus simple et la reprogrammation à distance est plus performante, car seulement les codes de l'application seront reprogrammés. Les évaluations de performance de MIROS montrent que MIROS est un système temps réel à faible empreinte mémoire et efficace pour son exécution. De ce fait, MIROS peut être utilisé dans plusieurs plateformes telles que BTnode, IMote, SenseNode, TelosB et T-Mote Sky. Enfin, MIROS peut être utilisé pour les plateformes RCSF à fortes contraintes de ressources.

Mots Clés— Système d'exploitation temps réel; intergiciel; réseau de capteurs sans fil; ordonnanceur hybride; allocation dynamique de la mémoire; machine virtuelle; énergie efficiente.

Abstract

With the recent advances in microelectronic, computing and communication technologies, wireless sensor network (WSN) nodes have become physically smaller and more inexpensive. As a result, WSN technology has become increasingly popular in widespread application domains. Since WSN nodes are minimized in physical size and cost, they are mostly restricted to platform resources such as processor computation ability, memory resources and energy supply. The constrained platform resources and diverse application requirements make software development on the WSN platform complicated. On the one hand, the software running on the WSN platform should be small in the memory footprint, low in energy consumption and high in execution efficiency. On the other hand, the diverse application development requirements, such as the real-time guarantee and the high reprogramming performance, should be met by the WSN software.

The operating system (OS) technology is significant for the WSN proliferation. An outstanding WSN OS can not only utilize the constrained WSN platform resources efficiently, but also serve the WSN applications soundly. Currently, a set of WSN OSes have been developed, such as the TinyOS, the Contiki, the SOS, the openWSN and the mantisOS. However, many OS development challenges still exist, such as the development of a WSN OS which is high in real-time performance yet low in memory footprint; the improvement of the utilization efficiency to the memory and energy resources on the WSN platforms, and the providing of a user-friendly application development environment to the WSN users.

In this thesis, a new hybrid, real-time, energy-efficient, memory-efficient, fault-tolerant and user-friendly WSN OS MIROS is developed. MIROS uses the hybrid scheduling to combine the advantages of the event-driven system's low memory consumption and the multithreaded system's high real-time performance. By so doing, the real-time scheduling can be achieved on the severely resource-constrained WSN platforms. In addition to the hybrid scheduling, the dynamic memory allocators are also realized in MIROS. Differing from the other dynamic allocation approaches, the memory heap in MIROS can be extended and the memory fragments in the MIROS can be defragmented. As a result, MIROS allocators become flexible and the memory resources can be utilized more efficiently. Besides the above mechanisms, the energy conservation mechanism is also implemented in MIROS. Different from most other WSN OSes in which the energy resource is conserved only from the software aspect, the energy conservation in MIROS is achieved from both the software aspect and the multi-core hardware aspect. With this conservation mechanism, the energy cost reduced significantly, and the lifetime of the WSN nodes prolonged. Furthermore, MIROS implements the new middleware software EMIDE in order to provide a user-friendly application development environment to the WSN users. With EMIDE, the WSN application space can be decoupled from the low-level system space. Consequently, the application programming can be simplified as the users only need to focus on the application space. Moreover, the application reprogramming performance can be improved as only the application image other than the monolithic image needs to be updated during the reprogramming process.

The performance evaluation works to the MIROS prove that MIROS is a real-time OS which has small memory footprint, low energy cost and high execution efficiency. Thus, it is suitable to be used on many WSN platforms including the BTnode, IMote, SenseNode, TelosB, T-Mote Sky, etc. The performance evaluation to EMIDE proves that EMIDE has less memory cost and low energy consumption. Moreover, it supports small-size application code. Therefore, it can be used on the high resource-constrained WSN platforms to provide a user-friendly development environment to the WSN users.

Index Terms— real-time operating system; middleware; wireless sensor network; hybrid scheduling; dynamic allocation; virtual machine; energy conservation

Contents

List of Figures

List of Tables

List of Acronyms

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ACK	Acknowledgment
ADC	Analog-to-Digital Converter
API	Application Programming Interface
CoAP	Constrained Application Protocol
ContikiRPL	Contiki routing protocol for low-power lossy IPv6 networks
DSN	Distributed Sensor Networks
EDF	Earliest Deadline First
EH	Exception Handling
EJVM	Embedded Java Virtual Machine
ELF	Executable and Linkable Format
FCFS	First Come First Served
FIFO	First Input, First Output
GC	Garbage Collection
IPC	Inter-Process Communication
ISR	Interruption Service Routine
IETF	Internet Engineering Task Force
IoT	Internet of Things
JVM	Java Virtual Machine
LR-WPANs	Low-Rate Wireless Personal Area Networks
MANET	mobile ad hoc network
MAC	Media Access Control
OS	Operating System
OTAU	Over-The-Air-Upgrade
RTOS	Real-Time OS
RT	Real-Time
RR	Round-Robin
RMS	Rate-Monotonic Scheduling
RPL	Routing Protocol for Low-power lossy IPv6 networks
SRT	Soft Real-Time
SJF	Shortest Job First

SFL	Segregated Free List
SF	Sequential Fit
TDMA	Time Division Multiple Access
TCB	Thread Control Block
UART	Universal Asynchronous Receiver/Transmitter
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
VANET	Vehicular ad hoc network
VM	Virtual Machine
WSN	Wireless Sensor Network
WPAN	Wireless Personal Area Networks

Chapter 1.
Introduction

Wireless Sensor Network (WSN) consists of spatially distributed autonomous wireless sensor nodes to monitor physical or environmental conditions (temperature, sound, pressure, etc.), and to cooperatively pass their data through the network to a main location (e.g., the sink node). WSN can communicate with the other networks (e.g., the Internet) through the base station (BA) or sink node (Figure 1.1). With the BA or sink, the sensing information can be propagated to the remote servers for storage, analysis, and processing.

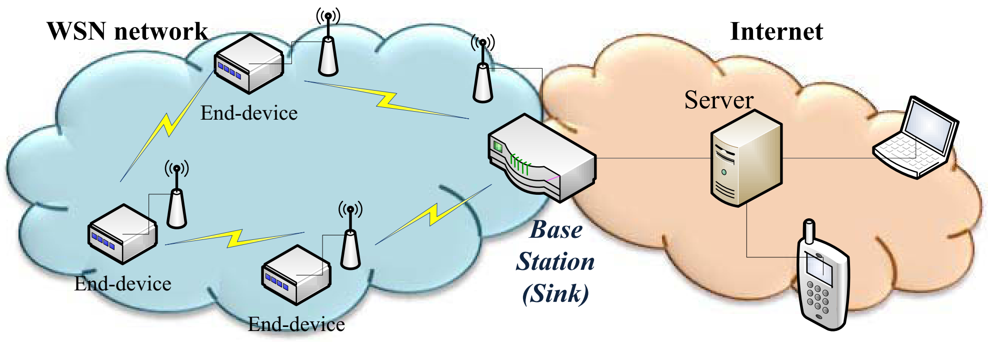


Figure 1.1: Topology structure of wireless sensor network.

Wireless sensor network operating systems (WSN OSes) are a collection of softwares that manage the hardware resources on the WSN nodes and provide common services to the WSN applications. The main tasks of the WSN OSes include the task scheduling, memory management, power management, networking, interaction with the hardware devices, shielding the machine-level details from the applications, etc. Since most WSN nodes are constrained in the memory and energy resources, the WSN OSes need to be low in the memory consumption and efficient in the energy utilization.

Middleware in this thesis refers to an intermediate adaptation layer which lies between the user application space and the low-level system space (Figure 1.2). This layer decouples the application code from the system code, implements a set of system services in the system space and provides an array of programming interfaces to the application space. The common functionalities of the WSN middleware include:

- satisfy the programming requirements of the WSN application, e.g., the support of multitasking programming in the WSN application.
- simplify the application programming process. Middleware should shield the underlying system details from the applications so that the users can program the applications easily without the necessity of considering the low-level platform details.
- improve the application reprogramming performance. Middleware can decouple the application code from the system code. Therefore, only the application image needs to be updated during the reprogramming process, and the application reprogramming performance can be improved.

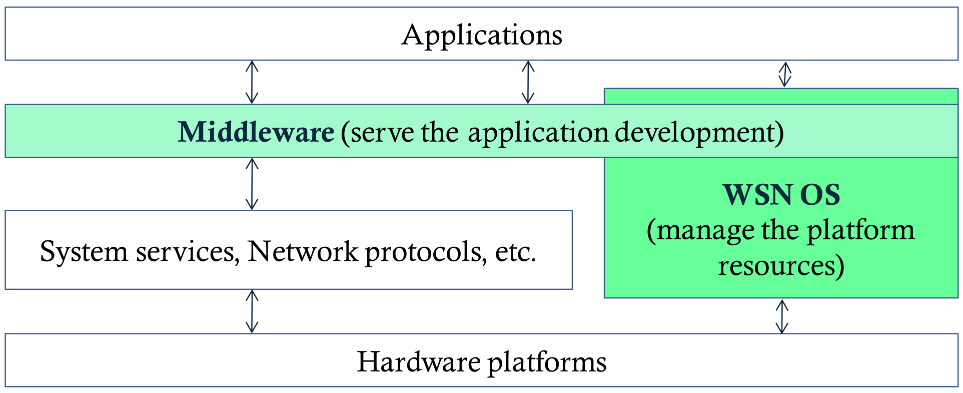

Applications
Middleware (serve the application development)
WSN OS
(manage the platform resources)
System services, Network protocols, etc.
Hardware platforms

Figure 1.2: Software architecture for the OS and middleware.

In this thesis, the research works about the design and implementation of a new WSN OS and a new WSN middleware will be presented.

1.1. Overview to the Wireless Sensor Network (WSN)

1.1.1. WSN Research History

Modern research of WSNs can be traced back to 1980s with the Distributed Sensor Networks (DSN) program at the Defense Advanced Research Projects Agency (DARPA). In 1978, the Technology components of DSN including the sensor technology, the communication and processing techniques, the distributed software and artificial intelligence, are identified in a Distributed Sensor Nets workshop [1]. Although at that time, the researchers on sensor networks had realized in the mind the development potential of DSN, this technology was yet not ready due to the large physic size of sensor nodes. However, the recent advances in the microelectro-mechanical, computing and wireless communication technology have led to a significant transformation in the WSN research, making it optimistic to achieve the early sensor network vision. At around 1998, the new wave of WSN research started. With the development of the small size and low cost nodes, many new sensor network applications have emerged, such as the vehicular sensor network, the body sensor network. Once again, DARPA acts as a WSN research pioneer and organizes a research program called SensIT [2] which focuses on the sensor information technologies, such as dynamic tasking, ad hoc networking, multitasking and reprogramming. At the same time, the IEEE organization defined the IEEE 802.15.4 standard [3] which specifies the physical layer and media access control (MAC) for the low data rate wireless personal area networks. Later, based on IEEE 802.15.4, the ZigBee standard [4] which specifies the high level of network communication protocols is published by the ZigBee Alliance. Currently, the WSN has been viewed as one of the most important technologies in the 21st century [5], and the WSN technologies are being accelerated by many commercial companies, including the Crossbow (xbow.com), the Worldsens (worldsens.citi.insa-lyon.fr), the Dust Networks (dustnetworks.com), the Sensoria (sensoria.com), the Ember Corporation (ember.com) and so on.

1.1.2. Introduction to the WSN

WSN is different from the other wireless networks (ad hoc, mesh, etc.) in that it has some typical features, such as the constrained memory, energy and computation resources; the limited communication bandwidth; and the widespread application domains.

1.1.2.1. WSN Nodes

Wireless sensor network nodes (WSN nodes), also known as motes, are the nodes that are capable of performing some processing, gathering sensory information and communicating with other connected nodes in the WSN network.

WSN nodes are commonly composed of the microcontroller, the wireless radio transceiver, the external memory, the power source and the electronic circuit used to interface with the sensors (Figure 1.3).

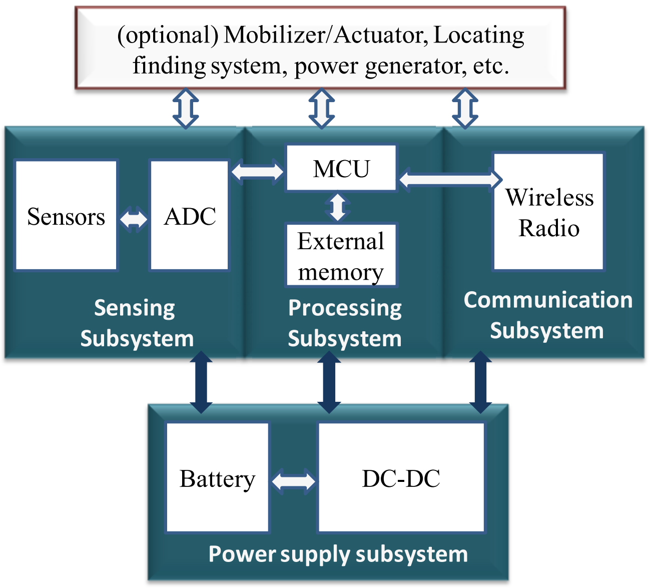


Figure 1.3: Typical architecture of WSN node.

1). Microcontroller: Microcontroller is often equipped on the WSN nodes due to its low cost, ease of programming and low power consumption. The general purpose processors commonly has higher power consumption than microcontrollers, thus they are not suitable for the WSN nodes. Digital Signal Processors are also not often considered for the WSN nodes, this is because the signal processing tasks in WSN is less complicated.

2). Transceiver: Currently, the ISM (industrial, scientific and medical) frequency band, which gives free radio, spectrum allocation and global availability, is used for the wireless access medium in WSN. On most WSN nodes, the functionality of both transmitter and receiver are combined into a single device known as a transceiver. The operational states of the transceiver include transmitting, receiving, idle, and sleep. These operations are generally managed by state-machine. As the power consumption of most transceivers operating in idle mode is almost equal to that in receiving mode [6], it is better to completely shut down the transceiver rather than leave it in the idle mode when there is no transmitting or receiving operations.

3). Power source: Since WSN nodes are often placed in a hard-to-reach location, changing the battery regularly can be costly and inconvenient. Thus, the guarantee of adequate energy to power the WSN system is important. On most WSN nodes, the power is stored either in batteries or capacitors, and it is mostly consumed by the tasks such as sensing, communicating and data processing. More energy is required for data communication than the other processes [104]. Since power resource on the WSN nodes is limited, the power conservation policy is essential to be used on the WSN nodes.

4). Sensors: Sensors are the hardware devices that produce a measurable response to a change in a physical condition like temperature or pressure. The continual analog signal produced by the sensors is digitized by an analog-to-digital converter (ADC) and sent to controllers for further processing. A sensor node should be small in size, consume extremely low energy, operate in high volumetric densities, be autonomous and be adaptive to the environment.

Currently, many kinds of WSN nodes have been developed, such as the BTnode, the Imote, the Mica2, the T-mote and the TesloB. (Figure 1.4). As WSN nodes are expected to be low in the price cost and small in the physic size, they are commonly constrained in the memory resources and computation ability (Table 1.1).

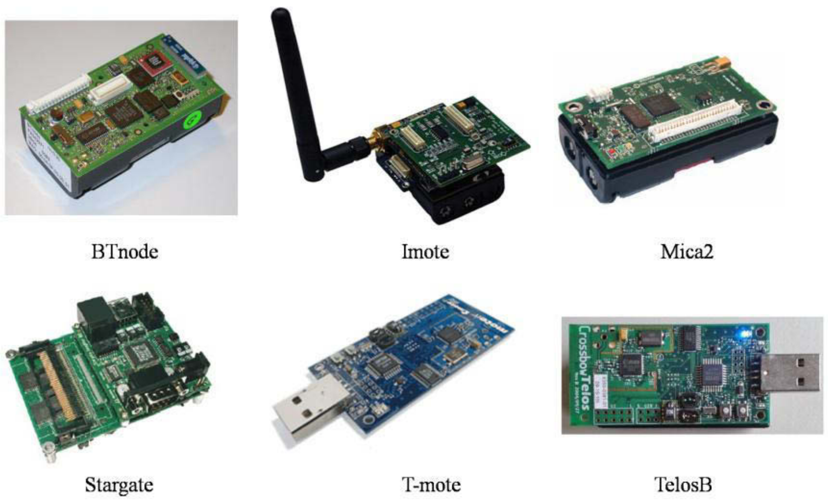


Figure 1.4: Current popular wireless sensor nodes.

1.1.2.2. WSN Network Standards

There are a number of standardization bodies in the field of WSNs. The IEEE focuses on the physical and MAC layers (e.g., the IEEE 802.15.4.), the Internet Engineering Task Force (IETF) works on layers Three and above. Currently, some predominant standards that are commonly used in WSN communications include the ZigBee, IEEE 802.15.4 and 6LoWPAN.

Sensor node	**microcontroller**	**Transceiver**	**Program+Data Memory**	**External Memory**
BTnodes	Atmel ATmega 128L (8 MHz @ 8 MIPS)	Chipcon CC1000 (433-915 MHz) and Bluetooth (2.4 GHz)	64+180K RAM	128 KB FLASH, 4 KB EEPROM
Eyes	MSP430F149	TR1001	60K FLASH + 2K RAM	256 B information memory
EyesIFX v2	MSP430F1611	TDA5250 (868 MHz) FSK	48 KB FLASH + 10 KB RAM	8 Mbit
IMote	ARM core 12 MHz	Bluetooth with the range of 30 m	64 KB SRAM	512 KB flash
IMote2	Marvell PXA271 ARM 11-400 MHz	TI CC2420 802.15.4/ZigBee compliant radio	32 MB SRAM	32 MB flash
Mica	ATmega 103 4 MHz 8-bit CPU	RFM TR1000 radio 50 Kbit/s	128+4 KB RAM	512 KB flash
Mica2	ATMEGA 128L	Chipcon 868/916 MHz	4 KB RAM	128 KB flash
MicaZ	ATMEGA 128	TI CC2420 802.15.4/ZigBee compliant radio	4 KB RAM	128 KB flash
SenseNode	MSP430F1611	Chipcon CC2420	10 KB RAM	48 KB flash
SunSPOT	ARM 920T	802.15.4	512 KB RAM	4MB flash
TelosB	Texas Instruments MSP430	250 Kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10 KB RAM	48 KB flash
T-Mote Sky	Texas Instruments MSP430	250 Kbit/s 2.4 GHz IEEE 802.15.4 Chipcon Wireless Transceiver	10 KB RAM	48 KB flash
XYZ	ML67 series ARM/THUMB microcontroller	CC2420 ZigBee compliant radio from Chipcon	32 KB RAM	256 KB flash

Table 1.1: Characteristics of current popular WSN nodes.

IEEE 802.15.4 is a standard which specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs). It is the basis for the ZigBee, ISA100.11a, WirelessHART, and MiWi specifications. These specifications further extend the IEEE802.15.4 standard by developing the upper network layers. Alternatively, the IEEE802.15.4 can be used with 6LoWPAN and standard Internet protocols together to build a wireless embedded Internet.

ZigBee [4] is an emerging low-cost, low-power, two-way wireless standard. It is based on the existing physical and data-link layers of IEEE 802.15.4 [3] Personal Area Network standard. It is different from the other network standards such as the Bluetooth and the Wi-Fi. In Table 1.2, the differences among the different standards are compared.

6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Networks. Its concepts originated from the idea that "the Internet Protocol could and should be applied even to the smallest devices" [7] and that "low-power devices with limited processing capabilities should be able to participate in the Internet of Things" [8].

Standards	**ZigBee IEEE 802.15.4**	**Bluetooth IEEE 802.15.1**	**Wi-Fi IEEE 802.11a, b, g**	**UWB IEEE 802.15.3 a**
Industry organizations	ZigBee Alliance	Bluetooth SIG	Wi-Fi Alliance	UWB Forum and WiMedia Alliance
Topology	Mesh, star, tree	Star	Star, mesh	Star
RF frequency	868/915 MHz, 2.4 GHz	2.4 GHz	2.4 GHz, 5.8 GHz	3.1 to 10.6 GHz (U.S.A.)
Data rate	250 Kbits/s	723 Kbits/s	11 to 105Mbits/s	110 Mbits/s to 1.6 Gbits/s
Range	10 to 300 m	10 m	10 to 100 m	4 to 20 m
Power	Very low	Low	High	Low
Battery life	Months to years	Rechargeable (days to weeks)	Rechargeable (hours)	Rechargeable (hours to days)
Node numbers	~ 65 000	~ 8	~ 32	128

Table 1.2: Feature comparison among ZigBee, Bluetooth, Wi-Fi and UWB.

1.1.2.3. WSN Applications

Different types of sensors can be equipped on the WSN nodes and a wide variety of ambient conditions can be monitored, such as the temperature, humidity, lightning condition, noise level, object movement and the pressure. By means of these sensors, WSN nodes can be used in widespread application domains, such as the environmental monitor, health care, industry automation, military, smart home and precious agriculture.

Environmental monitoring: The environmental monitoring involves the applications such as temperature sensing, light sensing, air pollution modalities, forest fire detection, volcano detection and river monitoring. Some related works focused on this domain include the GreatDuckIsland project [9], Sonoma Dust project [10], Foxhouse project [11], [12-14] and so on.

Health care: The advances in WSN bring along the new opportunities in the health care system. With the WSN technology, the health status of the patients can be monitored, and the doctors will be alerted in case that some emergent situations occur. Some related works in this application domain include the body sensor networks [15], the CodeBlue project [16], the research works [17-19], etc. In the articles [20-22], the survey to the health care WSN applications are investigated.

Industry automation: WSN nodes can be used in industry automation to monitor or control the industrial equipments and processes. Some related works in this domain include the WiSA [23], the research works [24-28], etc. In the article [29], the research challenges of using the WSN technology in the industrial automation are discussed.

Military: WSN can be used in military for the applications such as the battlefield surveillance, information collecting and the enemy tracking. Some related works in these domains include the research works [30, 31]. A survey for this application domain has been done in the article [32].

Smart home: Currently, the smart home WSN applications include the home monitoring [33-35], home automation [36], event detection [37, 38], home caring [39, 40], etc.

Agriculture: WSN has also played an important role in the agricultural monitoring and control, some related works include the research works [41-46]. In the articles [47, 48], the surveys about the WSN agricultural applications are made.

For more detail survey of WSN applications, it can be found in the articles [49, 50].

1.2. Significance of WSN OS Research

The main challenges of the WSN development derive from the contradictions between the constrained WSN platform resources (constrained memory resources, energy resources and computation ability on the WSN nodes) and the complicated WSN applications. With the recent advances in the microelectronic, computing and communication technologies, the wireless sensor network (WSN) nodes become physically smaller and more inexpensive. Since WSN nodes are minimized in physical size and cost, they are mostly restricted to the platform resources such as the processor computation ability, the memory resources and the energy supply. As a result, WSN nodes have limited ability to execute the complicated tasks. However, the WSN applications become more and more complicated with the development of the WSN technology, e.g., some WSN applications requires multitasking programming and real-time responsiveness guarantee. Therefore, the demand of utilizing the constrained platform resources to execute the complicated WSN applications becomes a challenge for the WSN development.

One way to address the challenges above is to design and implement a new OS dedicated to the WSN nodes. With an outstanding WSN OS, not only the constrained WSN platform resources can be utilized efficiently, but also the WSN applications can be served soundly. Currently, a set of WSN OSes have been developed, such as the TinyOS [51], the Contiki [52], the SOS [53], the openWSN [54] and the mantisOS [55]. However, many OS development challenges still exist. Firstly, most WSN OSes are either event-driven or multithreaded ones. The event-driven OSes cost less in data memory, but rate poorly in real-time performance. The multithreaded OSes are good in real-time performance but high in data memory cost. Since most WSN nodes are constrained in the memory resources and the real-time guarantee is required by many WSN applications (industrial engine control, medical care, contaminant detection, etc.), the development of a WSN OS which is high in real-time performance yet low in memory footprint is a critical challenge. Secondly, the RAM resources on most WSN nodes are precious and need to be utilized efficiently. To utilize the RAM resources efficiently, dynamic allocation needs to be implemented. However, many WSN OSes use the static allocation, such as the TinyOS and the openWSN. With these OSes, the utilization efficiency of the memory resources is low. Some other OSes like the Contiki and mantisOS implement dynamic allocators. However, the allocators are either inflexible or cannot address the memory fragmentation challenge well. Therefore, performance improvement for these allocators is needed. Thirdly, the application programming and reprogramming processes on most current WSN OSes are complicated. The programming is complicated because different application programming patterns are supported by different WSN OSes, thus the users need to understand the diverse low-level system details to begin programming. The reprogramming is complicated because the application part in many WSN OSes is not decoupled from the system part. Thus, the monolithic software image, which can be larger than 100 kilobytes, needs to be updated during the reprogramming process. This process is difficult to complete as the energy resource on the WSN nodes is constrained and the communication bandwidth in the WSN is limited. Therefore, the development of a WSN OS that can on the one hand simplify the application programming complexity, and on the other hand improve the application reprogramming performance becomes essential. Finally, the energy resource on most WSN nodes is constrained. To prolong the WSN nodes' lifetime and avoid the labor work of bringing the deployed nodes back for the power recharging, the energy conservation mechanism needs to be realized. The current sleep/wakeup mechanism used in most OSes for energy conservation is not quite adequate. To reduce significantly energy consumption, a new energy conservation approach becomes essential.

In this article, a new hybrid, real-time, memory-efficient, energy-efficient and user-friendly WSN OS MIROS is designed and implemented. MIROS targets to address some challenges existed in the current WSN OSes and has the following features: achieving the real-time scheduling with low data memory cost; utilizing the memory resources efficiently; managing the energy resources efficiently; simplifying the application programming complexity; improving the application reprogramming performance; improving the node reliability, as well as developing a new OS debugging approach.

1.3. Contributions and Thesis Organization

In this thesis, the works about the design and implementation of a new WSN OS MIROS and a new middleware EMIDE are presented. The organization and contributions of this thesis are as follows:

In Chapter 2, the related works about the WSN OSes are presented. In this chapter, seven different kinds of embedded OSes are surveyed. These OSes are investigated from different OS concerns and their pros and cons are analyzed (Section 2.2). This survey can help the researchers to understand the different features of the current WSN OSes, to select an appropriate OS for a typical WSN application, or to improve the performance of some WSN OSes. At the end of this chapter, a conclusion is made and the prospect of how to design the new OS is proposed (Section 2.3).

In Chapter 3, the design and the implementation of the new WSN OS MIROS is presented. MIROS targets to address some existed OS challenges, such as the achievement of the real-time scheduling with low memory cost; the improvement of the memory and energy resources utilization efficiency; the simplification of the application programming process, as well as the improvement of the application reprogramming performance. To achieve these objectives, the MIROS hybrid scheduling model (Section 3.1), the improved dynamic memory allocators (Section 3.2), the energy conservation mechanism (Section 3.4) and the new WSN middleware (Chapter 4) are designed, implemented and discussed.

In Chapter 4, the research on the middleware EMIDE is discussed. Two kinds of middleware mechanisms are mainly investigated in this chapter: the embedded Java virtual machine (EJVM) and the new mid-layer software EMIDE. In the Section 4.1, the fundamental concepts of the EJVMs are presented with the simpleRTJ used as the example. And then, a survey to the current EJVMs is made. This survey can help the researchers to understand the different EJVM mechanisms, to select a suitable EJVM for a given application context, or even to enhance the functionalities of a given EJVM. Based on this survey, the advantages and the drawbacks of the EJVM mechanism are discussed. And in order to take the EJVM mechanism's advantages yet avoid its drawbacks, a new middleware EMIDE is developed (Section 4.2). EMIDE has a set of functionalities similar to EJVMs, but keeps low memory and energy cost. With this feature, EMIDE can be a substitution of the EJVM when using on the severely resource-constrained WSN platforms. In the Section 4.3, the evaluation works to the EJVM and the EMIDE are done.

In Chapter 5, the application development works are presented. Finally, in Chapter 6 the conclusion and ongoing works are discussed.

Chapter 2.
Related Works of the WSN Operating Systems

In this chapter, a survey to the current WSN OSes (TinyOS, Contiki, SOS, openWSN, mantisOS, uCOS and simpleRTJ JavaOS) is investigated. The concerned OS features in this survey include the OS architecture, the scheduling model, the memory management, the application programming, the application reprogramming, the power management and the debugging approaches. Different approaches applied in these OSes are addressed, and the pros and cons of each approach are also discussed. This survey can help the researchers to understand these current WSN OSes, to select an appropriate OS for a given WSN applications, and to set forth the future research directions of designing a new WSN OS. The structure of this chapter is organized as follows: In Section 2.1, the basic terminologies of the WSN OSes are defined; In Section 2.2, the key features of each WSN OS are investigated. In Section 2.3, the comparison among the different OSes is listed, and the prospect of designing the new WSN OS MIROS is discussed.

2.1. OS Terminologies

In this section, some key OS terminologies are defined. These terminologies will be used for the presentation of the different OSes in this chapter.

Event: Event is the system signal which indicates the condition to take some action has been satisfied. An event can be generated by a key pressing, a packet reception, a timer expiration, etc.

Task: Task is the subroutine or method which will be executed after the corresponding event is triggered. In some OSes, task is synonymous with "process" (e.g., Contiki), in others with "thread" (e.g., mantisOS).

Preemption: Preemption is the act of suspending a task being executing temporarily without requiring its cooperation, and with the intention of resuming the task at a later time. Preemption is commonly achieved by the context switch and carried out by a preemptive scheduler. A preemption example is shown in the Figure 2.1. In this example, a low priority task A is preempted by a high priority task B.

Real-time (RT): Real-time qualifier claims that the correctness of an operation (e.g., the execution of a task) depends not only upon its logical correctness, but also upon the time in which it is performed (often referred to as "deadline"). Missing of the deadline will cause the operation to be useless or lead to disastrous consequence, e.g., a car did not stop in time to avoid the collision.

Soft Real-time (SRT): Soft real-time qualifier claims that an operation is expected to be completed within a time constraint, but the deadline guarantee is not obligated, and the missing of the deadline can be tolerated although it may degrade the system's service quality.

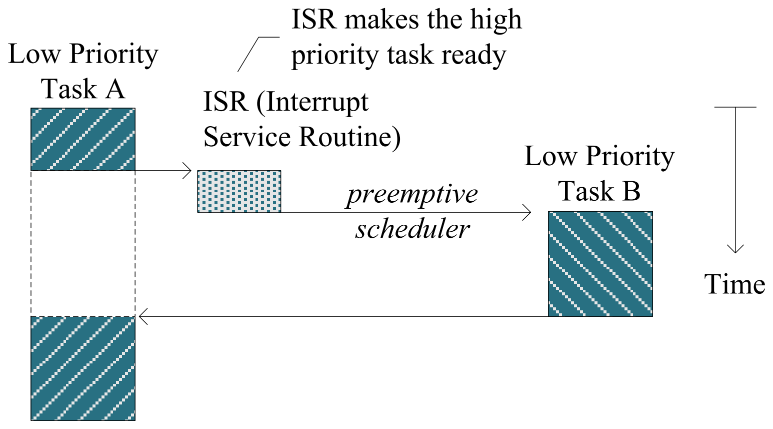

ISR makes the high priority task ready
Low Priority Task A
ISR (Interrupt Service Routine)
Low Priority Task B
preemptive scheduler
Time

Figure 2.1: Preemption and interruption.

2.2. Key Features of Different OSes

In this section, the key features (OS architecture, scheduling model, memory management, application development, network protocols, debugging, etc.) of the current popular WSN OSes are presented. The concerned OSes include the TinyOS [51], Contiki [52], SOS [53], openWSN [54], mantisOS [55], uCOS [57] and simpleRTJ JavaOS [56].

2.2.1. OS Architectures

OS architecture will not only have an effect on the kernel code size, but also on the way how the system services are provided. The current WSN OS architectures can be classified into three kinds: the monolithic architecture, the modular architecture and the virtual/script machine architecture.

2.2.1.1. Monolithic Architecture

With the monolithic architecture, the application component and the underlying system components are built together to form a single software image. This architecture has been used in the OSes like TinyOS and openWSN.

TinyOS [51] adopts the component-oriented programming abstraction which can provide the code modularity and facilitate the component reuse, but comes at the cost of larger learning curve and code complexity. TinyOS is developed by using the nesC language, which is a dialect of the C language optimized for the memory constraint of wireless sensor networks. Each component in TinyOS consists four parts: commands, events, tasks and frame. Components can connect to each other using the interfaces, and they are connected in a layered structure (Figure 2.2). The high-layer component can issue commands to the low-layer component, and the low-layer component can signal events to the high-layer component. Tasks in TinyOS are scheduled by a non-preemptive FIFO scheduler. During the development process, all the TinyOS components are built together with the tiny task scheduler to form a static executable image.

The advantage of the monolithic architecture is that the program code is compact. Moreover, the application can interact with the OS efficiently. However, any change to the monolithic system will cause the whole software system to be rebuilt and reprogrammed. Therefore, monolithic architecture OS has worse performance in the WSN reprogramming.

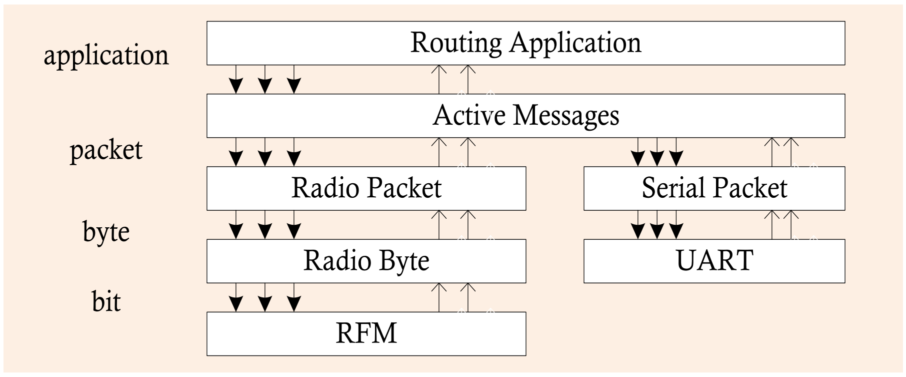


Figure 2.2: TinyOS monolithic architecture example.

2.2.1.2. Modular Architecture

Modular architecture OS is different from monolithic architecture OS in that some software components (the applications, protocols, drivers, etc.) are built independently into interactive modules. These modules can be updated remotely through the wireless, and then be resolved and executed on the WSN nodes. Currently, this architecture has been used in the SOS [53].

The modular architecture of SOS is shown in the Figure 2.3. Except the kernel and hardware-dependent components, all the other components (networking protocols, system services, applications, etc.) are built into modules.

The key advantage of the modular architecture OS is that the modules can be updated efficiently, thus the system fault repairing and application reprogramming can be performed with high performance. However, a module's address may change after it is updated, e.g., if the size of an upgraded module is larger than the original size, this module needs to be stored in a new memory space. Therefore, some mechanisms needs to be implemented in the modular architecture OSes to ensure that an updated module can still be accessed correctly by the others. In SOS, this mechanism is achieved by applying two approaches: 1). The position independent code (PIC) should be used for each SOS module. By so doing, the interactions within a module will not become invalided after this module is updated. Commonly, this PIC attribute can be achieved by enabling the PIC compiling option. However, the AVR "jump" instruction can only jumps up to 4KB, thus the size of the PIC module is limited to be less than 4KB on the AVR microcontroller. 2). Modules need to access each other in an indirect way. In this manner, the interactions among the modules can still keep valid after the modules are updated. In current SOS, two indirect interactive modes are realized: the asynchronous mode and the synchronous mode (Figure 2.4). With the synchronous mode, one module can access the others synchronously by using the function jump table. After a module moves to a new address, the update to the jump table can keep it still be accessible. With the asynchronous mode, one module can access the others asynchronously by posting a message into the message scheduling queue. Once this message is extracted by the scheduler, the other module can be processed. The asynchronous mode is needed in SOS is because SOS is an event-driven OS. In the event-driven OS, the preemption is not enabled, and the synchronous access cannot be allowed in many cases.

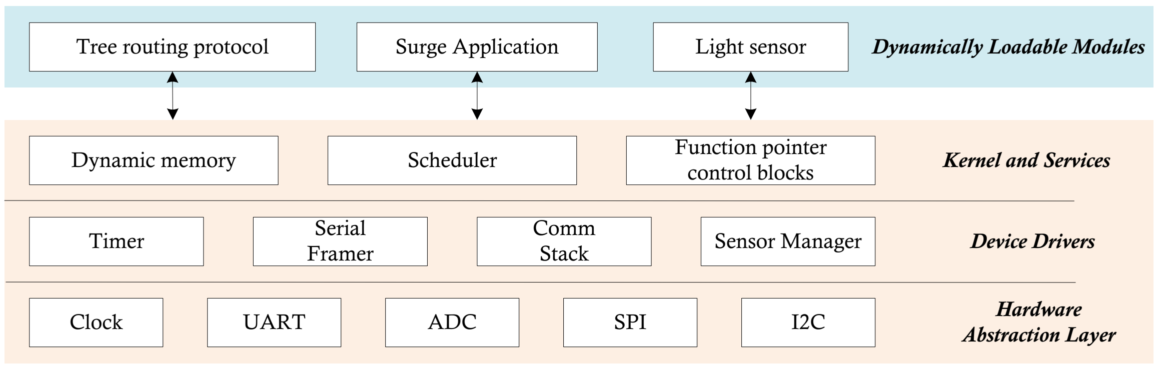


Figure 2.3: Module-based architecture of the SOS.

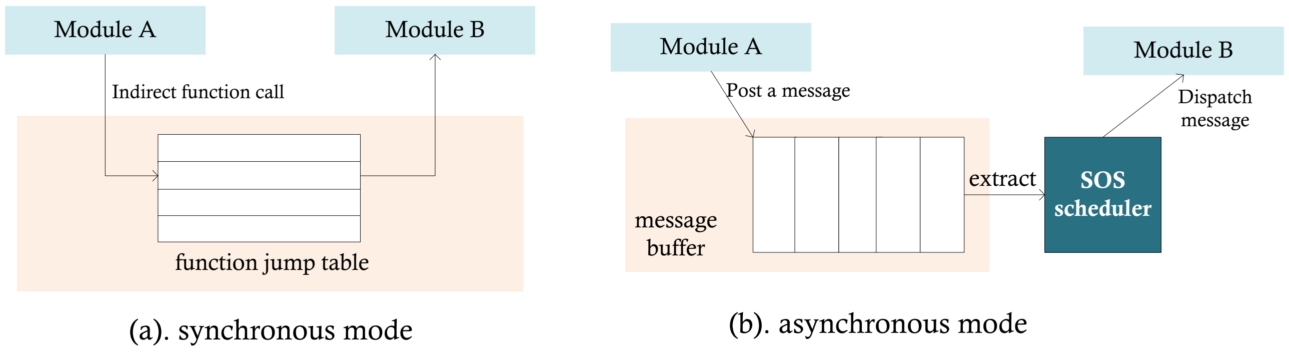


Figure 2.4: Indirect interactions among the SOS modules.

2.2.1.3. Virtual Machine Architecture

In the virtual machine (VM) based OS, the application code is separated from the underlying system code, and built independently into a stand-alone image. This application image will be loaded on the WSN nodes and be interpreted by the VM interpreter. Since the application image is executed in an interpretation way, the VM-based application can be hardware independent. This feature is significant for the users to address the challenges of programming in the heterogeneous WSN contexts. Currently, this architecture has been used in the simpleRTJ [56].

SimpleRTJ is a full-time specific Java VM/OS for the embedded devices. The architecture of simpleRTJ is depicted in the Figure 2.5. SimpleRTJ consists of two key components: the embedded JVM (EJVM) and the lightweight multithreaded JavaOS. The EJVM is in charge of interpreting the Java byte code while the JavaOS is in charge of supporting the multithreaded Java programming.

One advantage of simpleRTJ is that a user-friendly development environment can be provided to the WSN users. With simpleRTJ, the WSN applications can be programmed by the popular and user-familiar Java language. Moreover, the application reprogramming performance is high since only the Java application image is needed to be updated during the reprogramming process.

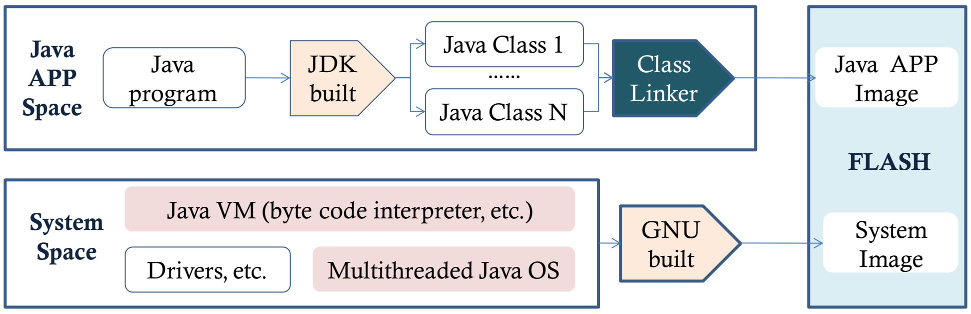


Figure 2.5: OS architecture of the simpleRTJ.

2.2.2. OS Scheduling Model

The scheduling model in the WSN OSes can be classified into two kinds: the event-driven scheduling and the multithreaded scheduling. In the past research work, there exist some debates and discussions about these two scheduling models [84-87]. In this section, the comparison between these two scheduling models is done firstly. And then the current event-driven scheduling OSes and the multithreaded scheduling OSes are presented.

2.2.2.1. Event-driven Scheduling and Multithreaded Scheduling

TinyOS, Contiki, SOS and openWSN are the event-driven scheduling OSes. In the event-driven OS, all the tasks are scheduled one by one. Each task runs to completion with respect to the others. The interruption is enabled during a task's run-time, but the preemption from one task to another is not allowed. Since the new task can be executed only after the previous task's context has been released, only one stack is needed in the event-driven OS. Thus, the memory consumption of the event-driven system is low. However, in the event-driven system, the time-critical task cannot be executed immediately after it is triggered. Instead, it needs to be deferred until the current task runs to completion. Due to this reason, the real-time performance of the event-driven OS is poor.

MantisOS, uCOS and simpleRTJ are the multithreaded OSes. In the multithreaded system, all the tasks (also called threads) can be executed concurrently. Threads can be switched, and before the switch operation the thread's run-time context needs to be saved so that its execution can be resumed next time. Therefore, every thread in the multithreaded OS should have a private stack. Due to this reason, the memory consumption of the multithreaded OSes is higher if compared with the event-driven ones. In the article [88], it is reported that the event-driven OS TinyOS achieves about a 30-fold improvement in the RAM memory requirement over the general purpose multithreaded embedded OS. Although the RAM memory consumption of the multithreaded OS is higher, its real-time performance is better if compared with the event-driven system, this is because the task preemption can be supported in this kind of system.

In Table 2.1, a comparison between the event-driven scheduling and the multi-threading scheduling systems is shown.

Features	**Event-driven Scheduling**	**Multithreaded OS Scheduling**
Task scheduling	Tasks are dispatched one by one	All threads can be executed concurrently
Preemption	Not enable	Enabled, overhead of context switch exists
Real-time performance	Poor, preemption not allowed	Well, threads can be preempted
Run-time stacks	All tasks share one global stack	Each thread has a private stack
Processor computation resources	Shared among all tasks in a cooperative way	Shared among all threads by thread switch

Table 2.1: Comparison of event-driven scheduling and multithreaded scheduling.

2.2.2.2. Event-driven OSes

TinyOS, Contiki, SOS and openWSN are the event-driven OSes. The structure of the event-driven system can be depicted as Figure 2.6, four components are composed: the event generators, the event channel, the event dispatcher as well as the system tasks. Event channel is required since the event generating speed can be more quickly than the event processing speed. After being posted into the event channel, the events will be extracted and dispatched one by one by the event dispatcher. Once an event is dispatched, the related task will be executed.

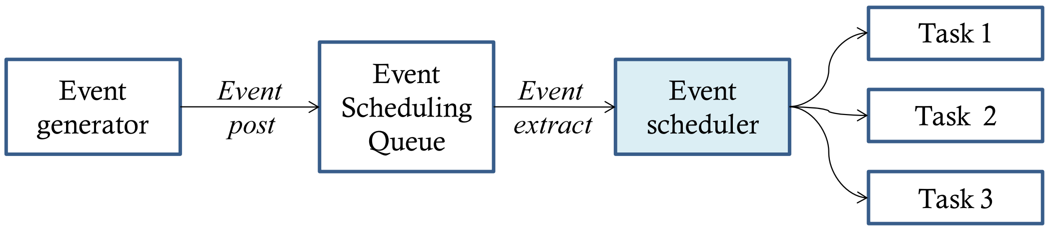


Figure 2.6: Event-driven scheduling structure.

Several topics need to be considered for the implementation of the event-driven scheduler, including the design of the event channel, the achievement of the concurrency, the improvement of the event-driven system's real-time performance, etc.

Event channel: Event channel is used to buffer the incoming events. In most event-driven WSN OSes including the Contiki, TinyOS, SOS and openWSN, the event queue mechanism is used for the implementation of the event channel. In TinyOS, only one queue is used and all the incoming events enter this queue by the FIFO (First Input, First Output) algorithm. Since FIFO is poor in the real-time scheduling, the sophisticated priority-based and deadline-based event queue mechanism is also implemented in TinyOS, and these mechanisms are provided optionally to the developers. In openWSN, only one event queue is use as well, but the priority-based scheduling algorithm is adopted. With the priority mechanism, the high-priority events can be dispatched more quickly. However, the priorities in openWSN are statically assigned, and cannot be updated dynamically. Thus, the low priority events can become starved if the high priority events keeps coming continuously. In Contiki, two-level scheduling mechanisms are implemented. The first one uses the FIFO event queue scheduling mechanism (similar to TinyOS), and is used to schedule the low-priority events. The second one uses the polling mechanism, and is used to schedule the high-priority events. If a low-priority event is generated, it will be posted into the FIFO event queue. If a high-priority event is generated, the polling flag in the related task will be set. The Contiki dispatcher will extract the event one by one from the event queue and then dispatch them. Before an event is extracted, it will firstly check whether some polling flags have been set or not. If yes, the corresponding time-critical tasks will be executed preferentially. Only after all the polling flags are cleared, the events in the event queue can be extracted and dispatched. By doing this, the high-priority tasks can be guaranteed to be executed in precedence of the low-priority tasks. In SOS, the FIFO event queue is also used. However, different from the Contiki and TinyOS, three level FIFO queues are used: the high-priority queue, the system queue and the low-priority queue. Once an event is generated, it will be posted into a given queue in terms of its emergence, e.g., the time-critical events will be posted into the high-priority queue, thus it can be processed earlier. Different event channel structures have different features, and they can be selected in terms of the WSN contexts. If non RT tasks exist in the system, the simple FIFO queue mechanism can be used. However, if the RT tasks are defined, the priority-based scheduling queue is more suitable to be used. Moreover, the dynamic priority updating mechanism other than the static assignment mechanism (e.g., the mechanism used in openWSN) can be applied to avoid the low-priority event being starved.

Concurrency-intensive operations: In the event-driven OS, tasks are not preempted. Every task should run to completion before the next task can be executed. Thus, two problems exist. One is the execution time of each task should not be too long. If not, the other high priority events can become stale. The other is the run-time execution of a task cannot be blocked. Otherwise, the system will be halt during the blocked time. In TinyOS, SOS and openWSN, the concurrency problems are solved by the split-phase approach. If the execution time of a task is too long, it should be split into small pieces. After the first piece is executed, the processor control will yield to the others. Likewise, if the execution of a task needs to be blocked, this task should also be split into two pieces from the blocking point: one is the piece of the code before the blocking call and the other is the piece of code after the blocking call. After the first piece of code runs to completion, the processor control will yield (without waiting). Once the result of the first piece of code is reached, an asynchronous notification will be delivered through an initial callback. On the receiving of this notification, the second piece of the code can be executed. By doing this, the processor resources will not be wasted during the task blocking time. Commonly, this split-phase operation is achieved by the state machine programming. In the Contiki, the protothreads [89] mechanism other than the split-phase mechanism is used to achieve the OS concurrency. Protothreads are a lightweight, stackless type of threads which provides a blocking context on top of the event-driven system, without the overhead of per-thread stacks. In the protothreads, a local continuations (LC) variable will be defined for each Contiki task (also called *process* in Contiki). If the task needs to yield the control or be blocked, it can save the current executing code address into the LC variable. The next time this task is called, it will not be executed from the starting address, but from the recorded LC address. By this means, the sequential control flow without complex state machines or full multi-threading can be realized inside the event-driven programming. The advantage of the protothreads over the split-phase approach is that the sequential code structure which allows for blocking functions can be provided. In the split-phase approach, tasks need to be split manually, and this is hard to be achieved when the control structures such as *if* conditionals or *while* loops exist in the task's program code.

Real-time Performance of event-driven system: Real-time tasks exist in many WSN applications. Since the preemption is not enabled in the event-driven OS, the real-time scheduling cannot be guaranteed in this system. Instead, only the soft real-time (SRT) scheduling can be supported. With the SRT scheduling, the time-critical tasks can be executed more quickly than the non-RT ones. To achieve the SRT scheduling, several mechanisms can be used, including the use of priority-based or deadline-based event queue (e.g., in SOS), the use of multiple level scheduling (e.g., the two levels of scheduling in Contiki), etc. By means of these mechanisms, the real-time performance of the event-driven system can be improved in a degree. However, the deadline of the RT tasks can still not be guaranteed. To achieve a RTOS, the task preemption should be supported in the system, and this can commonly achieved by the multithreaded system. In the next section, the multithreaded scheduling will be discussed.

2.2.2.3. Multithreaded OSes

MantisOS, uCOS and simpleRTJ are the multithreaded OSes. To implement a multithreaded scheduler, several topics need to be considered, including the allocation of the thread run-time contexts (thread control block, thread stack, etc.), the thread scheduling algorithm, the thread synchronization mechanism, the thread management, etc.

Thread control block (TCB) and thread stacks: Once a thread is created, the thread TCB and the thread stack need to be allocated. The TCB will be used to store the thread specific information, including the thread status, the thread stack pointer, the thread program counter, the pointer to the task/event bound with this thread, etc. Currently in many multithreaded OSes, the thread TCB are pre-reserved statically, including the simpleRTJ, mantisOS, etc. In this case, the maximum threads that can be created will be determined by the reserved number. The thread stack is used for the thread execution as well as the thread context saving. Before a thread is switched, its context needs to be saved so that this thread can resume its execution next time. The thread stack is commonly large in the size, e.g., in mantisOS the stack size is set to 120 bytes. Since every thread needs to have a private stack, the multithreaded OS is high in the RAM consumption. Therefore, the memory optimization is essential to be implemented for the multithreaded OSes in the resource-constrained WSN platforms.

Thread scheduling algorithms: The scheduling algorithms are used for distributing the computation resources among the parties that simultaneously and asynchronously request the resources. Many different scheduling algorithms exist, including the First Come First Served (FCFS), the shortest job first (SJF), the fixed priority pre-emptive scheduling, the round-robin scheduling (RR), the multilevel queue scheduling, etc. In *simpleRTJ*, the simple and starvation-free RR algorithm is used. A software timer is set with a configurable counter value 10 ms. Every time this timer is fired, the thread switch will be done. By doing this, the processor resources can be shared fairly among all the threads. In uCOS, the fixed-priority preemptive RMS algorithm is used. The task with the shorter cycle duration will have the higher priority. Anytime, the task that is active and has the highest priority will be executed. In the article [90], it is proved that the deadlines of the RT tasks can be guaranteed by RMS in case that the CPU utilization is below a specific bound. In mantisOS, the combination of the multilevel queue and RR scheduling algorithms are implemented (Figure 2.7). Five priority levels of queues are defined: priority_kernel, priority_sleep, priority_high, priority_normal, priority_idle. If a task is ready, it will be posted into a corresponding queue in terms of its priority, and in each queue the RR scheduling algorithm is used. Compared with the non-preemptive event-driven OSes such as Contiki and TinyOS, the RT performances of simpleRTJ and mantisOS can be better. However, the RR algorithm used in the simpleRTJ and mantisOS is not a real-time scheduling algorithm, thus these two OSes are not RTOSs (real-time OSes). In uCOS, the RT tasks can be schedulable, this is due to the RT scheduling algorithm that has been adopted in uCOS.

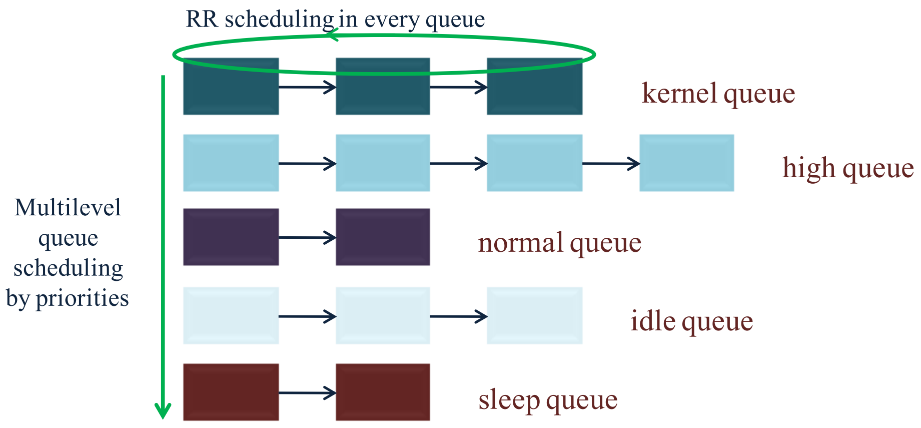

RR scheduling in every queue
kernel queue
high queue
Multilevel queue scheduling by priorities
normal queue
idle queue
sleep queue

Figure 2.7: Multi-level queue scheduling in mantisOS.

Thread synchronization: Since threads can run concurrently in multithreaded OS, the thread synchronization is needed to ensure that the specific portions of a program will not be executed by two concurrently-executing threads at the same time. Commonly, the implementation of thread synchronization includes the mechanisms of the mutex, the real/write lock, the semaphore, etc. Semaphore can be regarded as a variable which can be used to control the access to a shared resource in the parallel computing environment. It can be classified into two kinds: one is the counting semaphore which allows an arbitrary resource count, and the other is the binary semaphore of which the semaphore value is restricted to 0 and 1 (or locked/unlocked). In mantisOS and uCOS, the counting semaphore mechanism is applied. Two operations should be equipped to implement the counting semaphore: the signal and the wait. The wait operation will decrease the value of semaphore variable by 1 while the signal operation will increase the value of semaphore variable by 1. After the decrements operation, if the semaphore value becomes negative, the task which executes the wait operation will be blocked and added to the semaphore's queue. After the increment operation, if the semaphore value becomes positive, the next waiting task in the semaphore queue can be notified and removed to the ready queue. Commonly, the FIFO strategy is used for the task queuing in the semaphore queue. However, if the tasks have different priorities, the queue can also be ordered by priority. In this case, the highest priority task will be taken from the queue first. In simpleRTJ, the binary semaphore mechanism is used. Thus, a shared resource can be accessed by only one thread at any time.

Thread management: Thread management is essential when large numbers of threads exist in the OS, e.g., in the uCOS the supported thread number can be as large as 64 or even up to 256. In order to manage these threads efficiently (e.g., to get the highest priority thread from the ready threads quickly), the thread management mechanism needs to be implemented. In the uCOS, a lookup table is used for the thread management. If 64 threads are supported, a "8x8" lookup table will be used (Figure 2.8). All the tasks (or threads) in the uCOS have different priorities. Each thread corresponds to one entry in the lookup table. If a thread becomes ready, the corresponding entry in the table will be marked. By using this table and another constant checking table (named "OSTCBPrioTbl" in uCOS), the ready thread with the highest priority can be computed quickly without the necessity of scanning the whole thread lookup table. As a result, the RMS scheduling performance in the uCOS can be improved. In simpleRTJ, the supported thread number is not large (no more than 8 in the current version). Thus, the traversing scan approach is used to determine the next thread to be scheduled. In mantisOS, the supported thread number is larger than that in the simpleRTJ, but not so high as that in uCOS. Therefore, the multi-level thread queues are used for the thread management. With these queues, different types of threads can be distributed into different queues. Thus, the threads can be managed more efficiently.

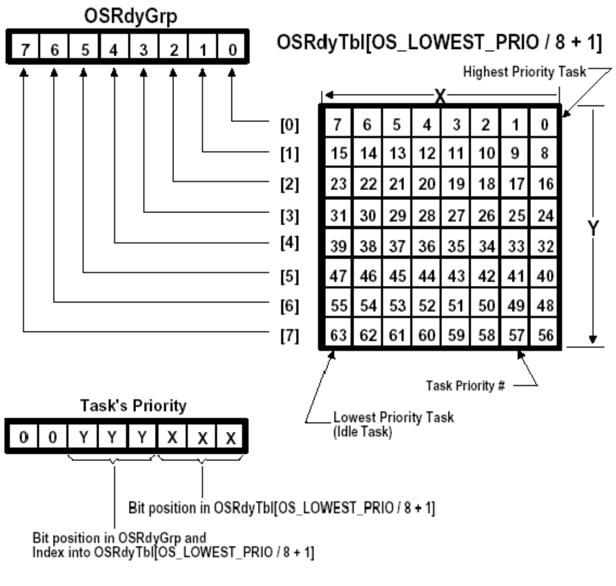


Figure 2.8: Thread lookup table in uCOS.

2.2.3. Memory Management

Memory management is important for the WSN OS since the memory resources on most WSN nodes are precious. In this section, different kinds of memory allocation mechanisms are presented.

2.2.3.1. Static Allocation

Static allocation has been used in several embedded OSes, including the TinyOS, the openWSN and the simpleRTJ. This approach has low overhead, but the memory resources cannot be reused. To improve the memory utilization efficiency, the dynamic allocation needs to be performed.

2.2.3.2. Bit Map Allocation

With the bit map allocator, the heap is divided into an array of fixed-size blocks. And the bit map table is used to record which blocks are allocated and which blocks are free. Upon allocation, the allocator scans the bit map table to find a set of continuous free blocks which are big enough for the objects, and then mark these blocks as allocated.

The allocation efficiency of this method is not high as the bit map scanning process is slow. Thus, this approach is not widely used in the conventional memory allocators. However, the development of the efficient bit-scan instructions in the processor makes this method be able to be used in some modern allocators, such as the jemalloc [91] and the TLSF [92].

2.2.3.3. Buddy System

Buddy system [93, 94] is an allocation algorithm which divides the heap space into partitions and gets the best-fit blocks by splitting the memory hierarchically into halves. Adjacent blocks which belong to the same hierarchical level are called buddies. This method is fast in the memory de-allocation, and the memory splitting and coalescing operations are also efficient. However, each block in the buddy system has an order, and the memory block size is proportional to 2^{order}. As a result, high internal fragmentation will occur after this approach is used. This makes the buddy allocator be more suitable for the large size object allocation, but not for the small size allocation on the WSN platforms.

2.2.3.4. Segregated Free List

Segregated free list (SFL) allocator [95] divides the memory space into segregated partitions. Each partition holds a set of specified size blocks, and a free list is used for the management of the free blocks in each partition. Upon allocation, a block is deleted from the matching free list. Upon releasing, the released block is added to the matching free list. Currently, this mechanism has been used in the OSes such as the Contiki, the SOS and the uCOS.

SOS memory allocator: The heap structure of the SOS is depicted in the Figure 2.9. The heap size is 1536 bytes and it is divided into three partitions. In different partitions, different kinds of blocks are used. And three segregated free lists are used to manage the free memory resources in these three partitions. Upon allocation, an appropriate partition will be chosen and a free block will be taken from the header of the free list. The advantage of this allocator is that the allocation can be performed with a constant response time. The drawback is that serious internal and external fragmentations can occur. For example, if a 36-bytes object needs to be allocated in Figure 2.9, the 128-bytes block will be used (internal fragmentation). In addition, if the 128-byte objects are requested to be allocated for 5 times, the allocation will be failed although enough free memory resources exist in the other partitions (external fragmentation).

Contiki memory allocator: One way to reduce the internal fragmentation in the SOS is to divide the memory heap into more partitions, and use each partition for the allocation of a particular type of object, such as the software timer, the event structure and the routing table entry. By doing this, the internal fragmentation problem in the SOS allocator can be eased. However, more partitions and more free lists need to be defined, and this will increase the RAM consumption in a degree. Currently, this approach has been used in Contiki.

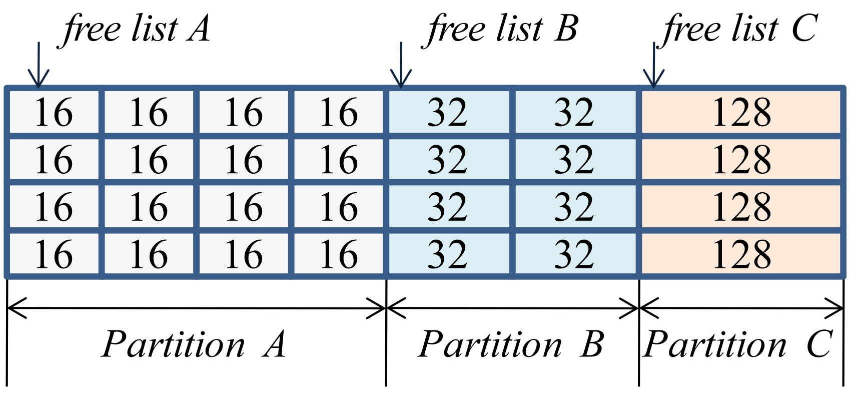


Figure 2.9: Segregated free list allocation in SOS.

The drawback of the SFL (segregated free list) allocation is that the pre-reserved size of each partition is difficult to be determined. This is because the required size of each partition can be varied in different contexts. For example, if a WSN node is configured to be an end-device, a partition with small size is enough for the allocation of the packet buffers. However, if the node is configured to be a router, a partition with larger size is needed (router needs to forward the packets for the end-devices). In case that the reserved partition size is too small, the memory overflow problem will occur. To avoid this problem, the partition size needs to be reserved to the maximum value for the worst case. But the memory insufficiency problems will occur. Due to these reasons, the flexibility of the SFL allocator is not good.

2.2.3.5. Sequential Fit

Sequential fit (SF) mechanism is different from the segregated free list (SFL) mechanism in that the memory heap is not divided into segregated areas. Instead, only one free list (other than several segregated free lists) is used to link all the freed memory. Since the size of each block can be different, the boundary tag technique needs to be used. Upon allocation, the free list will be searched to find a suitable free block, and the allocation response time is not constant. Currently, the SF allocator has been implemented for the memory allocation in mantisOS (Figure 2.10).

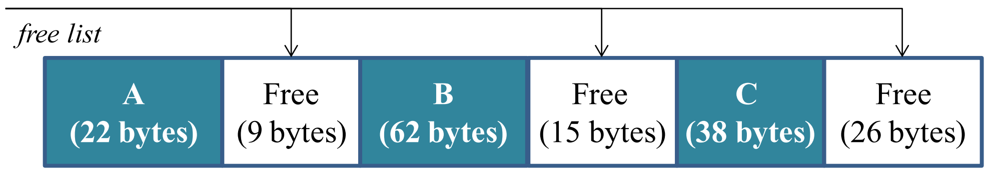


Figure 2.10: Sequential fit allocation in mantisOS.

The advantage of the SF allocation is that the internal memory fragmentation does not exist. Moreover, the partition pre-reservation problem in the SFL allocation can be avoided. However, the external memory fragmentations are prone to appear, e.g., if a 30-bytes object is required to be allocated in the Figure 2.10, the allocation will fail although the total free memory size is larger than 30 bytes. Currently in mantisOS, no mechanism is implemented to address the challenge of external memory fragmentation, and this decreases the memory utilization efficiency, especially when many small pieces of fragmentations exist inside the heap.

2.2.4. Application Development

2.2.4.1. Decoupling of Application Program

In order to provide a friendly application development environment to the users, the applications in the WSN need to be decoupled from the underlying systems. By this way, several advantages can be taken. Firstly, the users only need to focus on the application space and develop the applications without the necessity of considering the low-level software and hardware details. Thus, the application programming process can be simplified. Secondly, the WSN reprogramming performance can be improved since only the application image other than the whole software image is needed to be reprogrammed.

Currently in many WSN OSes, the applications can be decoupled from the systems, including the TinyOS, the Contiki, the SOS, the simpleRTJ, etc. In TinyOS, the application can be separated from the system by the VM Maté [103]. With Maté, the TinyOS application can be programmed by the byte code instructions, and a tiny size application image will be generated. In the Contiki, an ELF dynamic linker is implemented. With this linker, the application can be decoupled from the system and built into a loadable ELF module. In the simpleRTJ, an embedded JVM (EJVM) is developed. By means of this EJVM, the application can be independent from the system and be programmed by the popular and object-orient Java language. No matter which mechanism is applied, the WSN application programming complexity can be eased, and the reprogramming performance can be improved.

2.2.4.2. Application Reprogramming

The needs of providing new application services make it essential to reprogram the WSN applications. Since sensor nodes are prone to be deployed in the harsh environments where the human cannot access easily, the reprogramming commonly needs to be achieved through the wireless. As the WSN communication bandwidth is limited and the wireless transmission is high energy cost [104, 115, 116], the ways of reprogramming the WSN applications efficiently becomes a challenge. To address this challenge, the reduction of the reprogramming code size is significant.

The WSN reprogramming code size is related to the OS architecture. If the modular-based OS (e.g., the SOS) or VM-based OS (e.g., the simpleRTJ) are used, the reprogramming code size can be smaller. This is because only the application component needs to be updated for these OSes. As a result, the reprogramming energy cost decreased and the code disseminating success probability improved. In case that the monolithic architecture OS (e.g., the openWSN) is used, the monolithic software image needs to be updated during the reprogramming process, and the reprogramming code size can be larger than 100 KB. In this case, some optimization mechanisms are advisable to be applied to reduce the reprogramming code size. Currently, one popular optimization approach is the update of only the differential code changes (also called deltas), and this approach has been implemented in several different ways, such as in the Reijers [105], the Hermes [106], the zephyr [107], the Rsync [108] and the Remote incremental linking [109].

Besides the reduction of the reprogramming code size, the development of the code dissemination protocols is also essential. The current popular dissemination protocols include the XNP [110], the MNP [111], the Trickle [112], the Deluge [113] and the MOAP [114], etc. In the articles [78, 79], a survey to these approaches is made.

2.2.5. Energy Conservation

The energy resources on the WSN nodes are constrained, thus the power management is essential to be implemented to conserve the energy resource. Currently, many power management mechanisms have been developed, including the sleep/wakeup mechanism, the low duty-cycle control mechanism, the topology control mechanism, the data predication mechanism and the data compression mechanism.

Sleep/wakeup and low duty-cycle control mechanisms conserve the energy resources by decreasing the active working period of the WSN nodes. Sleep/wakeup mechanism can be used in both the application aspect and the WSN OS aspect. On the one hand, the WSN application can be configured to work in a periodical way. After the sensor data is sampled and the sensor packet is transmitted, the nodes can fall asleep. By doing this, more energy resources can be conserved when the nodes are in the idle time. On the other hand, the sleeping instruction can be called by the OS to make the nodes fall asleep when there are no active tasks left to be scheduled. Currently, this sleep/wakeup mechanism has been used in several WSN OSes like the TinyOS, Contiki, mantisOS, etc. Besides the sleep/wakeup mechanism, the low duty-cycle control is another approach to decrease the active period of the WSN nodes. Different from the sleep/wakeup mechanism, the low duty-cycle control approach achieves this objective from the network protocol aspect, rather than the application and the OS scheduling aspect. Currently, this approach has been used widely in the TDMA (Time Division Multiple Access) protocols [120-122]. With these protocols, the communication among the nodes can be synchronous, and the WSN nodes can enter the idle status for more time. As a result, more energy resources can be conserved.

Topology control mechanism and data prediction mechanism conserve the energy resources by reducing the communication redundancy and the data sampling redundancy respectively. Topology control is an approach used to adaptively choose a minimum subset of nodes to maintain the network connectivity. If a node is not selected, it can fall asleep to save the energy. In this manner, the communication redundancy can be reduced, and the network lifetime can prolong. Currently, this approach has been used in the research works [117-119]. Data predication approach reduces the energy cost by diminishing the redundant sampling to the sensor devices. It is reasonable to be used since the sensor data evolution process in some contexts can be predicted by a model. If the predication is within certain error bounds, the redundant sampling operations can be avoided. Currently, this approach has been used in the research works [123-128].

Data compression approach is also available to conserve the energy resources since the energy consumed during the compression and decompression processes can be much less than the energy cost during the large-size code wireless transmission process [115, 116]. Currently, different methods to compress the code size have been discussed in the articles [129-131].

2.2.6. OS Debugging

The OS run-time context is complicated since many tasks need to be executed concurrently. To develop the OS more efficiently, an effective debugging approach is needed. With the debugging support, the system run-time process can be tracked more clearly. As a result, some potential bugs in the OS coding can be found and fixed.

The traditional debugging approaches include the usage of "printf", the usage of the serial port and the usage of the breakpoint setting. However, these methods cannot work well for the debug on the resource-constrained WSN nodes. The "printf" is not useful as there are commonly no screens on the sensor nodes. Moreover, the execution overhead of "printf" is too high for the resource-constrained WSN nodes. The serial port debugging is also not ideal as the transmission speed of the debugging data is slow. Thus, the debugging data overflow problem will occur. The breakpoint method is useful to debug the code which is executed in logical sequence, but not useful to debug the complicated system in which many interruptions and preemptions occur concurrently. Moreover, once the breakpoint is met, the system run-time will be halt, and this may also cause some problems. For example, in a sensor network which uses the ZigBee protocol in the network layer. If a node is under debugging and a breakpoint is met, this node will be halted and no more exchange the packets with the others. In this case, the coordinator node may consider that this node has been lost and delete it from the network. Later, the node can no more be debugged effectively.

Due to the above reasons, it is essential to achieve a new OS debugging approach which will have the following features: 1). has low execution overhead, thus the execution of the debugging code will not influence the execution of the regular code on the sensor nodes. 2). be able to process the debugging data in high speed, thus the debugging data overflow problem can be avoided. 3). the system execution will not be halt during the debugging process.

2.3. Discussions and Conclusions

In this section, the feature comparison to the different WSN OSes is firstly presented. And then, the new OS design prospects are introduced.

2.3.1. Feature Comparison of WSN OSes

A comparison on the features of different WSN OSes is listed in the Table 2.2. TinyOS is a monolithic OS. In order to improve the application reprogramming performance, the VM Maté can be built on it. With the Maté, the TinyOS application can be programmed by the byte code instructions. Then, only the small-size application image rather than the monolithic software image needs to be updated during the reprogramming process. By so doing, the reprogramming energy cost can be decreased and the reprogramming success probability can be increased. Besides Maté, the TOSThread [132] is also developed for TinyOS. TOSThread can support the fully-preemptive threads in the TinyOS application level. It can be regarded as a natural extension to the concurrency model of historical TinyOS, and it targets at the objective of combining the ease of threaded programming with the efficiency of the event-driven scheduling. With the TOSThread, the programming of long-running computations and the preserving of timing-sensitive nature can be both achieved in the TinyOS applications. Contiki is also an event-driven OS. Similar to the TOSThread, a preemptive multithreaded scheduler is also implemented in Contiki. This scheduler is built as a library on top of the Contiki event-driven kernel. With this scheduler, Contiki can run in a hybrid scheduling model and the long-time computations can be preempted when needed. MantisOS and uCOS are both multithreaded OSes, and the thread preemption can be supported in these two OSes. In uCOS, the real-time scheduling algorithm RMS is realized. In mantisOS, the non-RT scheduling algorithm RR (Round-Robin) is used. Consequently, mantisOS does not become a RTOS.

In terms of the comparison results in the Table 2.2, it is shown that different OSes have different features. Commonly, event-driven OS has low RAM memory footprint, but it's poor in the real-time performance. Multithreaded OS consumes higher RAM memory, but it's better in the real-time performance. Therefore, the design and implementation of a new WSN OS which has good real-time performance yet keeps small memory footprint is significant.

Since the memory resources on the WSN nodes are constrained, the memory defragmentation mechanism is needed to be implemented so that the utilization efficiency of the precious memory resources can be improved.

WSN OSes	**OS concerns**			
	OS Architecture	**Execution model**	**RT scheduling**	**Memory management**
TinyOS	Monolithic	Event-driven	No	Static
TinyOS with Maté	Modular (only for application)	Event-driven	No	Static
TinyOS with TOSThread	Modular (only for application)	Event-driven for kernel / Multithreaded for APP.	No	Static
Contiki	Modular (only for application)	Event-driven	No	Dynamic SFL, defragmentation not supported
Contiki with multithreading	Modular (only for application)	Hybrid	No	Dynamic SFL, defragmentation not supported
SOS	Modular (ELF module)	Event-driven	No	Dynamic SFL, defragmentation not supported
MantisOS	Monolithic	Multithreaded	No	Dynamic SF, defragmentation not supported
uCOS	Monolithic	Multithreaded	Yes	Dynamic SFL, defragmentation not supported
simpleRTJ	JVM	Multithreaded	No	Static
openWSN	Monolithic	Event-driven	No	Static

Table 2.2 (a): Feature comparison of different WSN OSes.

WSN OSes	**OS concerns**				
	Decoupling of APP from system	**APP language**	**Energy Conservation**	**Network Protocols**	**Simulation**
TinyOS	No	NesC	Sleep/wakeup mechanism	TinyOS DYMO, 6LoWPAN, Active message, TDMA, IEEE802.15.4	TOSSIM [133]
TinyOS with VM Maté	Yes, by the VM	Byte code instructions	Sleep/wakeup mechanism	TinyOS DYMO, 6LoWPAN, Active message, TDMA, IEEE802.15.4	TOSSIM [133]
Contiki	Yes, by dynamic ELF linker	C	Sleep/wakeup mechanism	uIP, Rime, ContikiRPL	COOJA [134]
SOS	Yes, by dynamic ELF linker	C	not considered explicitly	Message, AODV, ICMP, etc.	N/A
MantisOS	No	C	Sleep/wakeup mechanism	Comm	XMOS [135]
uCOS	No	C	None	N/A	PC-based Simulator
simpleRTJ	Yes, by the JVM	Java	None	N/A	Graphical remote debugger
openWSN	No	C	Low duty-cycle control through IEEE802.15.4e	CoAP, HTTP, UDP, TCP, RPL, 6LoWPAN, IEEE802.15.4e	OpenSim PC-based Simulator

Table 2.2 (b): Feature comparison of different WSN OSes.

The sleep/wakeup mechanism has been used in most WSN OSes to achieve the energy conservation. This mechanism is effective, but not adequate for the WSN proliferation. To prolong the WSN nodes' lifetime and avoid the labor of bringing the nodes back for the energy recharging, more energy-efficient conservation mechanism is needed to be developed.

The decoupling of the application space from the system space is important for the application development in the WSN. With this decoupling, the WSN application reprogramming performance can be improved significantly. Virtual machine and dynamic loadable module are the popular mechanisms in the current WSN OSes to achieve this objective. Yet, these mechanisms are either not trivial in the memory cost (e.g., the code size of the JVM simpleRTJ can be more than 20 KB if implemented on the 8-bit AVR microcontroller), or high in the energy cost efficiency (e.g., the dynamic loadable ELF module needs to be resolved and re-linked on the target node before being executed). Therefore, it is essential to design and implement a new approach which can decouple the application space from the system space, yet remain memory efficient and energy efficient.

2.3.2. New OS Design Prospects

To address the challenges discussed above, a new WSN OS is prospected to be designed and implemented. This OS will have the following features:

- Can achieve real-time scheduling, but remains low memory cost.
- Memory fragments can be defragmented, thus high memory utilization efficiency can be achieved.
- Precious energy resources on the WSN nodes can be utilized efficiently. Consequently, WSN nodes can work in a long period without the necessity of bringing back to recharging the batteries.
- Application space is decoupled from the underlying system space, and the low-level details are shielded to the users. As a result, users can program the applications without large learn curve. Moreover, application reprogramming can be achieved with high performance.

In the Chapter 3, the new OS MIROS will be presented, MIROS achieves the above objectives by implementing the following mechanisms:

- A new hybrid scheduling model is used in MIROS. With this scheduling model, good real-time performance can be realized. Meanwhile, less memory resources are consumed (Section 3.1, Chapter 3).
- Memory fragments can be defragmented in the proactive or reactive way. With this defragmentation, memory utilization efficiency is improved (Section 3.2, Chapter 3).

- The combination of the software technology and the multi-core hardware technology is used to conserve the energy resource, improve the node run-time reliability and achieve new OS debugging approach (Section 3.4, Chapter 3).
- The new middleware EMIDE is implemented inside MIROS to decouple the application space from the system space. By so doing, a user-friendly application development environment can be provided to the users. And this will prompt the WSN technology to be used in widespread application domains (Chapter 4).

Chapter 3.
MIROS Design and Implementation

In this chapter, the design and implementation of the new hybrid, real-time, memory-efficient and energy-efficient WSN OS MIROS will be presented. MIROS targets to manage the constrained platform resources on the WSN nodes efficiently. Firstly, it implements the hybrid scheduling, both the event-driven scheduling and the multithreaded scheduling are realized. By doing this, the real-time scheduling can be achieved with low data memory cost. Secondly, it realizes the dynamic memory allocators. These allocators are improved on the base of the current WSN allocators. Compared to the other WSN allocators, they are more flexible to adapt to the run-time contexts. Moreover, the memory resources can be utilized more efficiently. Thirdly, the multi-core hardware technology is combined with the software technology to address some WSN OS challenges such as the conservation of the precious energy resource, the improvement of the node run-time reliability and the achievement of a new effective OS debugging approach. In addition to the above topics, the discussion on the MIROS software timers, the Inter-process communication (IPC) and the networking protocol is also presented. As for the services provided to the user application development, it will be discussed in Chapter 4 by means of the middleware EMIDE.

3.1. MIROS Hybrid Scheduling

3.1.1. Motivation of Hybrid Scheduling Design in MIROS

As discussed in the related works, one advantage of the multithreaded OS is that the real-time reaction can be achieved by performing the thread switch. However, every thread needs to have an independent run-time stack to realize this switch. Thus, the RAM consumption of the multithreaded OS is high, which makes it unsuitable to be used on the high memory constrained WSN nodes.

To address the challenge above, the thread stack optimization technique stack-size analysis [136] is proposed. With this technique, the memory cost of the thread stacks can be reduced. Traditionally, the size of the thread stack in the multithreaded OS is heuristically assigned. The stack overflow problem will occur if the estimated value is too small. To avoid this problem, the stack size needs to be assigned to a large value, e.g., in mantisOS a 120-byte stack is reserved for each thread. However, the memory waste problem will occur in this case. Ideally, the stack size needs to be reserved to the minimal but system-safe value, and this is what stack-size analysis approach targets to achieve. By using this approach, the application execution process is modeled as a control-flow graph. The application functions, their starting addresses and the local stack usages represent the nodes in the graph, and the branch instructions represent the edges. After the model is built, the stack usage of each thread can be calculated by the straightforward depth-first search. If a stack pushing instruction is observed during this search, the stack usage value will increase. In reverse, if a stack popping instruction is met, the stack usage value will decrease. By this way, the required size of each stack can be computed exactly, and the RAM resources can be avoided to be wasted. However, several cases exist in which the stack usage may not be boundable, including the interruption reentrancy, the recursive calls and the indirect function calls. If the interrupts are reenterable, the number of received interruptions will be difficult to be decided. If recursive calls exist, the cycle will appear in the control-flow graph. If indirect function calls exist, the target address of some functions cannot be known until run-time, thus disconnection will appear in the control-flow graph. To solve these problems, several solutions are proposed in the article [136] correspondingly. However, the final evaluation results show that these solutions are not ideal as it is impossible to know whether some typical cases will be reached or not during the program executing process. Consequently, the RAM resources will still be wasted by the thread stacks although the guesswork in determining the required stack size is eliminated.

The stack-size analysis mechanism reduces the size of each stack in the multithreaded OS. Besides this approach, another method to decrease the RAM consumption of the multithreaded OS is to reduce the total number of the stacks. Currently, this method has been used in the new OS MIROS. In MIROS, the hybrid scheduling model is adopted for the purpose of reducing the thread number.

3.1.2. Hybrid Scheduling Model in MIROS

The hybrid scheduler of MIROS is designed to combine the advantages of both event-driven system's low memory cost and multithreaded system's good real-time performance. Firstly, the event-driven scheduling model is used to keep low RAM consumption. However, the real-time reaction cannot be supported by the event-driven scheduler. Thus, the multithreaded scheduler is also implemented in MIROS, and it is used dedicatedly to schedule the RT tasks. Consequently, a hybrid scheduler, which consists of both an event-driven scheduler and a multithreaded scheduler, is realized in MIROS. By using this hybrid scheduler, the real-time response can be achieved. Meanwhile, the stack number can be decreased as all the non-RT tasks share only one run-time stack.

In Figure 3.1, the hybrid scheduling structure of MIROS is depicted. Two schedulers are implemented in parallel. The system tasks are classified into two kinds: the RT ones and the non-RT ones. The RT tasks require the preemption support, thus they are scheduled by the multithreaded scheduler. The non-RT tasks have loose restriction to the response time, thus they can be scheduled by the event-driven scheduler, and share only one stack. Two schedulers can switch to each other, but at any time only one scheduler can be active. The multithreaded scheduler has the priority higher than the event-driven one, thus it can preempt the event-driven scheduler when necessary. If all the RT tasks are inactive, the OS will run in the pure event-driven scheduling model. However, if any RT task becomes active, the event-driven scheduler will be suspended, and then the OS will switch to the multithreaded scheduling model.

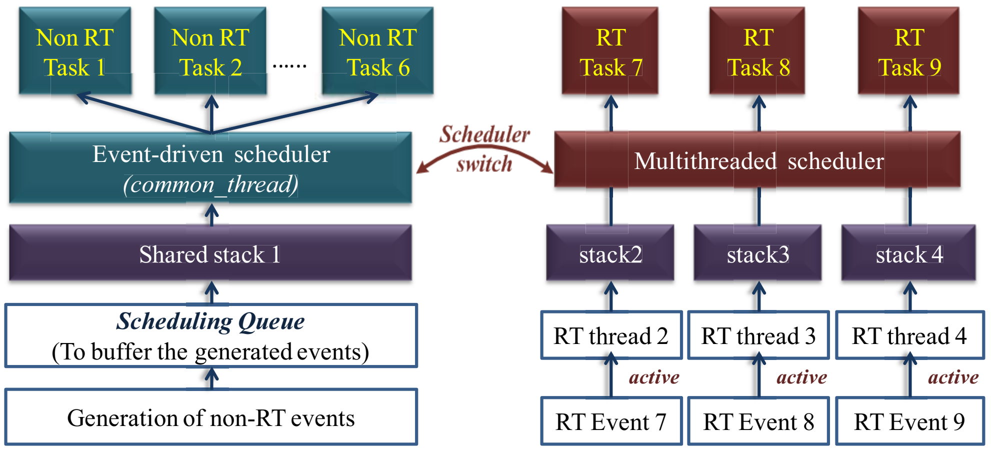


Figure 3.1: MIROS hybrid scheduling structure.

The advantage of this hybrid scheduling is that the real-time scheduling can be achieved with low RAM consumption. For the example in the Figure 3.1, there are nine system tasks (T1, T2, ..., T9) and three of them are RT ones: T7, T8 and T9. If the pure multithreaded OSes (such as mantisOS and uCOS) are used for the scheduling of these tasks, nine threads along with nine stacks are required. This is because in the pure multithreaded OSes, no matter a task is RT or not, a thread should be created for its execution. The main difference may be that the RT tasks are executed by the high priority threads while the non-RT tasks are executed by the low priority threads. However, the non-RT tasks have loose constraint to the response time, and the preemption is not needed among their execution. Therefore, these tasks do not need to be executed by threads, but can be scheduled one by one by the event-driven scheduler. By this way, a hybrid scheduling model is designed. With this hybrid scheduler, only four stacks rather than nine stacks need to be created for the execution of these nine tasks: three of them are used for the RT threads to execute the RT tasks (T7, T8, T9) while the left one is used for the event-driven scheduler to execute all the non-RT tasks (T1 to T6). By doing this, the stack number in MIROS decreases significantly. Since the stack is large in the memory size, the RAM consumption of the MIROS can be reduced greatly if compared with that in the pure multithreaded WSN OSes.

3.1.3. Real-time Scheduling Algorithm in MIROS

3.1.3.1. Scheduling Algorithms

Before a comprehensive scheduling algorithm is available, most time-critical applications were scheduled by the *table-driven cyclic scheduling* approach. The timeline is divided into a set of fixed length slots, and the tasks are bound statically to these slots based on their execution conditions [137]. This approach is easy to be implemented, and the run-time overhead is low as well. However, it is not flexible for the scheduling in the dynamic situations, e.g., a complete redesign of the scheduling table may be required if a new task is created. To solve this problem, the dynamic scheduling algorithm is developed, and EDF (earliest deadline first scheduling) [90] is one of the popular representatives.

EDF is a dynamic priority-based scheduling algorithm where priorities are updated dynamically and inversely proportional to the deadlines of the tasks. A priority queue is commonly used to store the active tasks. Whenever a scheduling condition meets (task finishes, new task released, etc.), this queue will be searched to find the task closest to its deadline, and then this task will be scheduled.

Besides the cyclic scheduling and dynamic priority scheduling, another approach is the static-priority scheduling. This approach can be considered as a midway between the cyclic scheduling and dynamic scheduling. A popular representative of this scheduling algorithm is the ***RMS*** (Rate Monotonic Scheduling) [90]. In the RMS, the priorities are statically assigned on the basis of the tasks' period duration: the shorter the period duration is, the higher the priority will be. In Figure 3.2, an example for the RMS and EDF scheduling is shown.

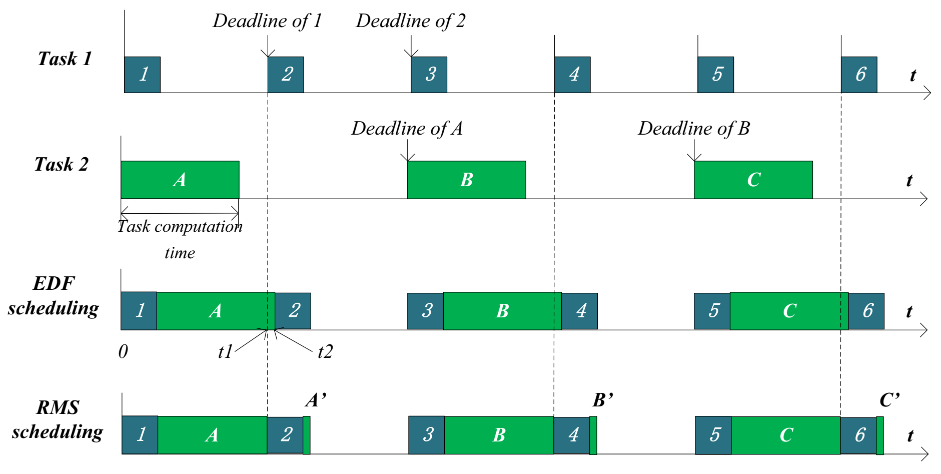


Figure 3.2: Example for RMS scheduling and EDF scheduling.

3.1.3.2. Schedulability Test

Both the RMS and EDF algorithms can be used to achieve the real-time scheduling in the RTOS. In [90], the *schedulability test* of the RMS is proposed. Provided that the RT tasks scheduled by RMS have the following properties:

- No resource sharing, or any kind of semaphore blocking, or non-blocking among the tasks.
- Deterministic deadlines are equal to periods.
- Context switch times and other thread operations are free and have no impact on the model.

Then, a feasible scheduling that will always meet the deadlines can exist by using RMS in case that the RT tasks' CPU utilization is below a specific bound as follows:

$$U = \sum_{i=1}^{k} \frac{C_i}{T_i} \le k(2^{1/k} - 1)$$

where U is the CPU utilization, C_i is the task computation time, T_i is the task release period, and k is the number of tasks. In case that $k \to \infty$, then $k(2^{1/k} - 1)$ is 0.693. This means that any RT task can meet their deadlines by using the RMS if the U is no larger than 0.693.

For the algorithm *EDF* [90], a feasible real-time scheduling can be achieved in case that:

$$U = \sum_{i=1}^{k} \frac{C_i}{T_i} \le 1$$

this means that EDF can guarantee that all RT tasks' deadlines can be met on condition that the CPU utilization of the RT tasks is no more than 100%. Compared to the fixed priority scheduling algorithm RMS, EDF can guarantee the deadlines in the system at higher loading.

3.1.3.3. MIROS Real-time Scheduling Algorithm

Different scheduling algorithms have different features. It depends on the task characteristics and the task timing requirements to select an appropriate algorithm. *EDF* uses the dynamic scheme, the dynamic mapping between the task deadline and the task priorities should be performed incessantly during the run-time. Thus, it is complicated to be implemented and the runtime overhead is also high. All these features prevent it to be used on many real-time kernels although it allows the full processor utilization. In MIROS, the static scheme *RMS* is applied for the multithreaded scheduler to schedule the RT tasks. The main reason to select the RMS is because it is simple to be implemented and the execution overhead is also low. Thus, it is more suitable to be used on the resource-constrained WSN nodes. Although the processor utilization of the RMS algorithm is not 100%, it is not a problem for the MIROS. This is because the hybrid scheduler is implemented in MIROS, and the left processor computation resources, which are not used by the MIROS multithreaded scheduler, can be utilized by the MIROS event-driven scheduler to schedule the non-RT tasks.

In order to explicate the hybrid scheduling workflow in the MIROS, an example in the Table 3.1 is used. In this example, ten tasks (*T1* to *T10*) are defined. The tasks *T10* and *T9* are the RT ones while the others are the non-RT ones. The two RT tasks are scheduled by the multithreaded scheduler while the other eight non-RT tasks are scheduled by the event-driven scheduler. The CPU utilization of the RT tasks (*T10* and *T9*) is 0.75. Since this value is within the schedulable bound (that is 0.828), these two RT tasks can meet their deadlines by using the RMS scheduling algorithm. As for the left CPU resource (that is 0.25), it can be used by the event-driven scheduler to schedule all the non-RT tasks. In Figure 3.3, the scheduling switch process of this example is illustrated. At time *t=5*, no RT tasks are active, thus the OS switches from the multithreaded scheduling model to the event-driven scheduling model, and then the non-RT tasks will be scheduled. At time *t=6*, the RT task *T9* becomes active. Then, the event-driven scheduler is suspended and the OS switches to the multithreaded scheduling model again. At time *t=8*, *T10* becomes active. As the priority of *T10* is higher than *T9*, *T9* is preempted by *T10*. After *T10* runs to completion at the time *t=9*, the thread switch is done, and then *T9* can resume the execution. Once *T9* is completed at the time *t=10*, the scheduler will switch to the event-driven scheduling model once more.

Tasks		**Run-time Stack**	**Priority**	**Scheduler**	**Period**	**Execution time**	**CPU utilization**	**Schedulable bound**	**Schedulable**
RT tasks	T10	Thread 1/ Stack 3	Highest	Multithreaded scheduler	4	1	U = 1/4 + 3/6 = 0.75	$k(2^{1/k}-1) =$ 0.828	Yes
	T9	Thread 2/ Stack 2	2nd highest		6	3			
Non-RT tasks	T1-T8	Stack 1 (shared by all non-RT tasks)	Lowest	Event-driven scheduler	N/A		0.25	N/A	N/A

Table 3.1: MIROS hybrid scheduling example.

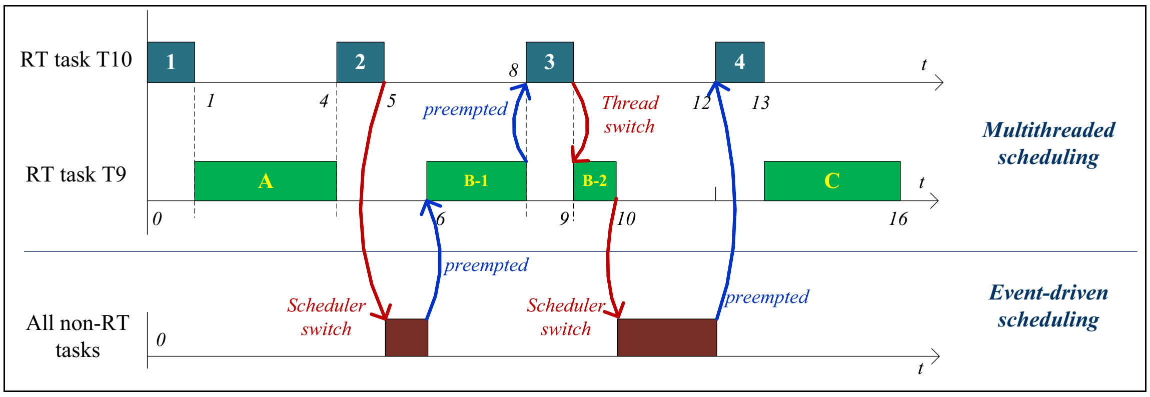


Figure 3.3: MIROS hybrid scheduling process.

3.1.3.4. Real-time Scheduling for Aperiodic/Sporadic Tasks in MIROS

The RMS and EDF are the real-time scheduling algorithms for the periodical RT tasks. In case that the aperiodic or sporadic RT tasks exist in the system, the other aperiodic/sporadic task scheduling algorithms that are developed on the base of the RMS algorithm, such as the Deferrable Server [138], the Sporadic Server [139], can be applied extendedly to the current MIROS multithreaded scheduler. And this extending apply will not have an influence on the MIROS hybrid scheduling mechanism described above, this is because the event-driven scheduler in the MIROS has the priority lower than the multithreaded one, and it can run only after the multithreaded scheduler is idle.

3.1.4. Implementation of MIROS Hybrid Scheduler

In this part, the implementation of the event-driven scheduler, the multithreaded scheduler and the scheduler switch mechanisms in MIROS will be presented.

3.1.4.1. MIROS Event-driven Scheduler

Implementation of MIROS event-driven scheduler: In most event-driven WSN OSes, the scheduling queue is used for the implementation of the event-driven scheduling. In these OSes, the generated events will be buffered in the scheduling queue, and then extracted and dispatched one by one by the event scheduler, seen in the Figure 3.4(a). Scheduling queue is needed as the event generating speed can be more quickly than the event dispatching speed. The scheduling principle can be the First Input, First Output (FIFO) or the priority-based, and the latter one can support better real-time performance.

MIROS event-driven scheduler is typical in that only the non-RT tasks need to be schedule by it. Thus, the real-time performance is not a critical design factor, and the flag polling mechanism other than the scheduling queue mechanism is used for the event scheduling. In Figure 3.4(b), the structure of the MIROS event-driven scheduling is depicted. Every non-RT task in the MIROS has one-bit flag. Once a task is activated, the related task's flag will be set. The event scheduler will poll the flags in loop. If a flag is found to be set, the related task will be executed. Task flags have the priorities that are statically assigned offline. The flags with the higher priorities will be polled in advance, thus the corresponded tasks can be executed earlier. If no flags are set, the sleeping directive will be executed, and then the node will fall asleep for the energy conservation. Compared with the scheduling queue mechanism, this flag polling mechanism is less flexible as the scheduling sequence is fixed, but it is more efficient and more suitable for the non-RT task scheduling in the MIROS.

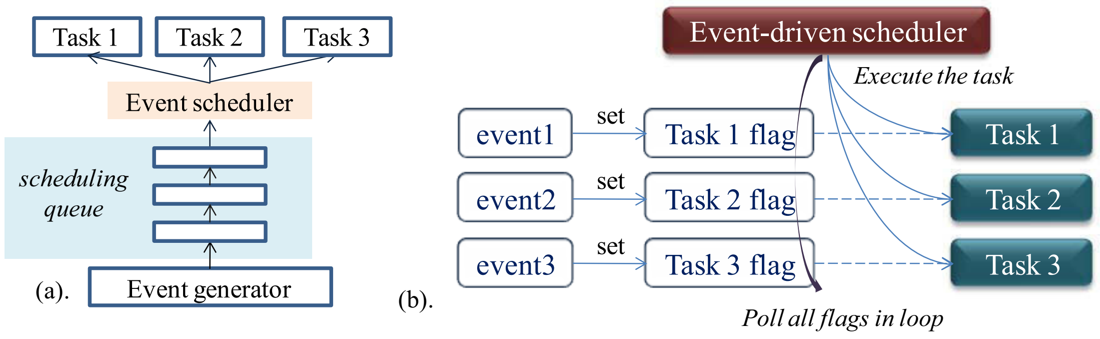


Figure 3.4: Event-driven scheduling structures.

3.1.4.2. MIROS Multithreaded Scheduler

Compared with the general multithreaded scheduler (MS), the *MS* in MIROS is typical in that only the RT tasks need to be scheduled by MIROS *MS*. Thus, the thread number in MIROS is small, and multithreaded scheduling structure can be simplified.

Thread TCB (thread control block) and thread stack: The thread TCB for the MIROS is shown in the Figure 3.5. Since the thread number in the MIROS is not large, all the thread TCBs are pre-reserved statically. As for the thread stack, it is allocated dynamically when the thread is created. And after a thread is deleted, the related stack memory space will be de-allocated.

```
/** @brief Thread TCB structure for multithreaded scheduler in MIROS */
typedef struct thread_s {
    struct thrd_tcb *next;
    uint8_t *thrd_sp;      /* stack's run-time address. */
    tsk_handler_t thrd_tsk;  /* pointer to the task executed by this thread. */
    struct thrd_tcb *semQ_next;  /* queue for the resource semaphore. */
    void *data;              /* data used for the task. */
    uint16_t thrd_period;    /* period of the task, will determine the priority of this thread. */
    uint8_t status;        /* "UNUSED, SUSPENDED, ACTIVE, etc." */
} miros_thread_t;
```

Figure 3.5: Data structure of MIROS thread control block.

Thread scheduling and synchronization: The RMS scheduling algorithm is implemented in MIROS for the thread scheduling. With the RMS, the schedulability of the RT tasks can be predicted off-line. The thread synchronization in MIROS is achieved by the semaphore mechanism.

Thread management: Thread management is a key topic for the implementation of the multithreaded scheduler. It is essential as the number of the threads scheduled by the multithreaded scheduler can be large, e.g., in the uCOS the thread number can be as large as 64 or even 256. To schedule these threads efficiently, a sound thread management mechanism is needed. Currently in the uCOS and the mantisOS, the "8x8" lookup table and the multilevel thread queues are used respectively for the purpose of managing the threads efficiently.

MIROS is different from the pure multithreaded OSes (uCOS, mantisOS, etc.) in that only the RT tasks need to be executed by threads, thus the thread number is small and no complicated thread management mechanism is needed. As a result, a simple single-link queue is used for the thread management. In this queue, all the MIROS threads are queued in the order of their priorities (from highest priority to the lowest priority). And then, the next thread to be scheduled by the RMS can be determined quickly by finding the first active thread in this thread queue.

3.1.4.3. Scheduler Switch in MIROS

In MIROS, two kinds of switches exist. One is the *thread switch* which occurs within the multithreaded scheduling system, and the other is the *scheduler switch* which occurs between the event-driven scheduler and the multithreaded scheduler. In order to make the switching process efficient and easy-managed, the event-driven scheduler in the MIROS is also implemented as a thread, named the "*common_thread*". And the *run* function of this *common_thread* is to execute the MIROS event-driven scheduler for the scheduling of all the MIROS non-RT tasks. By doing this, the MIROS hybrid scheduling can be managed as the pure multithreaded scheduling:

1). If all the RT threads are inactive, the next thread to be scheduled will be the *common_thread*. In this case, the OS will switch to the event-driven scheduling model, and all the non-RT tasks will be scheduled one by one.

2). If the *common_thread* is executing and any RT thread becomes active, the *common_thread* will be preempted, and then the OS will switch to the multithreaded scheduling model. By this means, the hybrid scheduling in MIROS can be implemented simply and executed efficiently.

With the mechanisms above, the hybrid scheduling in the MIROS can be implemented easily and the hybrid scheduling process becomes more efficient.

3.1.5. Related Works of Hybrid Scheduling

Currently, the hybrid scheduling structure has been used in several WSN OSes to address the different kinds of WSN challenges.

In TinyOS, the TOSThread [132] is developed for the purpose of combining the *ease* of threaded programming model with the *efficiency* of fully event-driven OS. TOSThread is achieved in the TinyOS application level (Figure 3.6a). It supports the application logic to be implemented in the user-level preemptive threads, and supports the lengthy computation program to be developed inside the TinyOS application.

In Contiki, the hybrid scheduling structure is also implemented (Figure 3.6.b). The same as the TOSThread in the TinyOS, the multithreaded scheduler in the Contiki is also built on the top of the event-driven scheduler. But different from that in the TOSThread, the multithreading in the Contiki is not used particularly for the user application. Instead, it is applied optionally in the low-level system, and serves the Contiki *processes* that need the preemption support (e.g., the lengthy computation cryptographic task [81]).

LIMOS [140] is another hybrid WSN OS. The LIMOS multithreaded scheduler is built on top of the event-driven scheduler as well. But different from the former two mechanisms, the scheduling model in the LIMOS can be configured in terms of the run-time contexts: 1). In case that the thread number inside each process is configured to be 1, LIMOS runs in the pure event-driven scheduling model. 2). In case that the process number in the system is configured to be 1, LIMOS runs in the pure multithreaded scheduling model. By doing this, LIMOS becomes customizable and context aware, and can thus adapt flexibly to different application contexts.

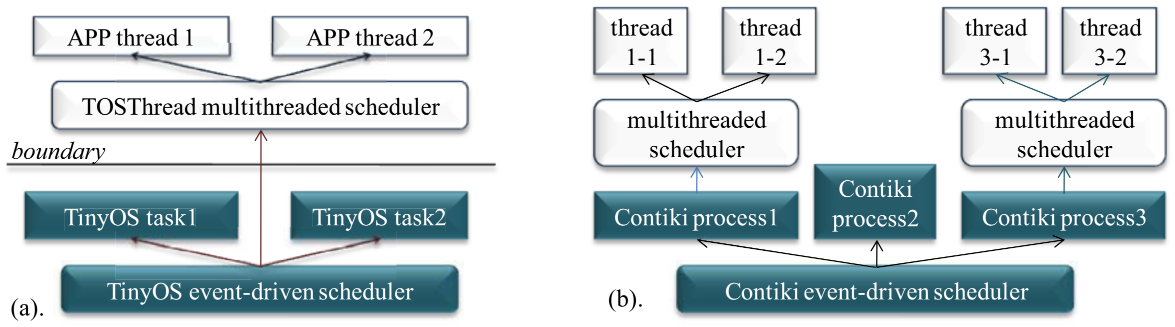


Figure 3.6: Hybrid scheduling in TinyOS and Contiki.

The hybrid scheduling structure in the MIROS is different from the mechanisms above in that the MIROS multithreaded scheduler is implemented in parallel with the event-driven scheduler, and the design purpose of the MIROS hybrid scheduling is to achieve the RTOS with low data memory consumption. A key drawback of the hybrid scheduling structure in the TinyOS, Contiki and LIMOS is that these OSes are still not real-time ones even if the hybrid scheduler are implemented. This is because the event-driven scheduling is still used in the native scheduling layer of these OSes. For example, if the Contiki *thread 1-2* is executing and the *thread 3-1* needs to be executed quickly (Figure 3.6.b), the preemption to the *thread 1-2* cannot be achieved immediately. This is because all the Contiki *processes* are still scheduled by the event-driven scheduler, and the *process 3* can be executed only after the *process 1* runs to completion.

3.1.6. Performance Evaluation

In this section, the memory consumption and the execution efficiency of the schedulers in the TinyOS, Contiki, SOS, mantisOS and MIROS are evaluated. The evaluation is done on the iLive node (Figure 3.7). ILive is equipped with the AVR ATmega1281 microcontroller; it has 128 KB of FLASH and 8 KB of RAM. Besides, 11 sensors are equipped on this node, including 1 temperature sensor, 1 light sensor, 1 air humidity sensor, 3 decagon sensors [147] and 4 watermark sensors [148]. For the software development tool, the AVR studio [149] is used.

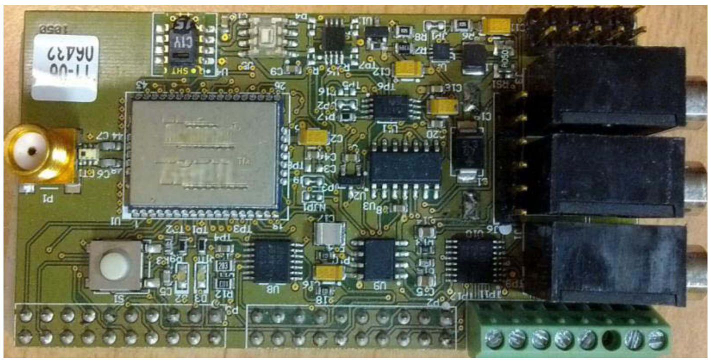


Figure 3.7: ILive sensor node.

3.1.6.1. Code Size of Different OS Schedulers

Code size of the event-driven schedulers: The code size of the different event-driven schedulers are shown in the Table 3.2. In the *TinyOS*, only one FIFO scheduling queue is used for the event scheduling, and the scheduler code size is small. In the *Contiki*, two level scheduling mechanisms are adopted: the FIFO scheduling queue for the asynchronous events and the polling mechanism for the high-priority events. As a result, the scheduler code size is larger than that in the TinyOS. In the *SOS*, three priority-based scheduling queues are used. Moreover, the indirect access and the module management mechanisms are implemented. Consequently, the complexity of the SOS scheduling structure increases and more code memory is consumed. In the *MIROS*, the simple flag polling mechanism is applied and the code size is small. In *mantisOS*, no event-driven scheduler is developed.

Code size of the multithreaded schedulers: The code size of the different multithreaded schedulers is shown in the Table 3.2. The *mantisOS* scheduler consumes more code memory than the others. This is because mantisOS is a pure multithreaded OS, all the tasks in this OS are executed by threads. As a result, the thread number is large. In order to manage these threads efficiently, the multilevel-queue (five ready queues and one sleep queue) scheduling mechanism is implemented, and this increases the complexity of the mantisOS scheduling architecture. In the *MIROS*, only the RT tasks need to be scheduled by the multithreaded scheduler, thus the thread number is small and the multithreaded scheduling structure is simple. In consequence, the code memory consumption is not high. In the *TOSThread*, the code size of the multithreaded scheduler is large, this is because a flexible boundary between the user code and the kernel code is implemented in TOSThread. With this approach, the TinyOS core can be kept unchanged, thread-safe and non-invasive, but the code size of the TOSThread becomes larger.

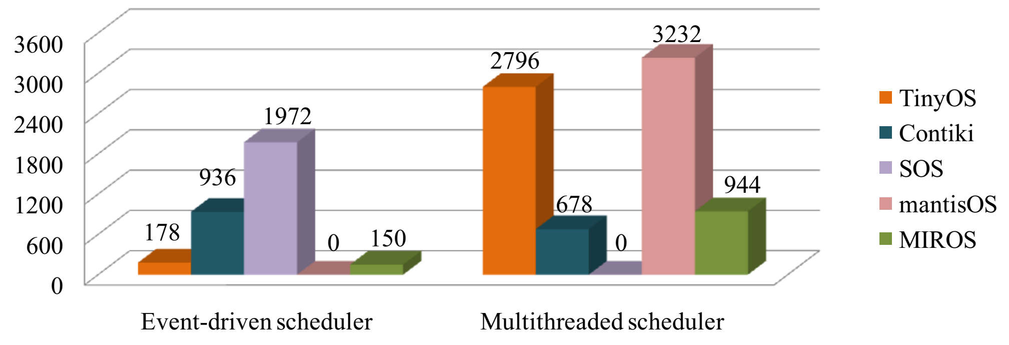


Table 3.2: Code size of different OS schedulers.

3.1.6.2. Data Memory Cost of Different OS Schedulers

The data memory consumption of the event-driven scheduler can be evaluated as follows:

$$E1 = Size(DATA/BSS) + Size(PCB) + Size(SQ)$$

$$= MDATA/BSS + SPCB*NPCB + SSQ*LSQ$$

in which $M_{DATA/BSS}$ represents the size of the DATA/BSS sections (the global variables, static variables, etc). S_{PCB} and N_{PCB} represent respectively the structure size of the *process control block* (In SOS, it is named as *module control block*) and the number of the defined *processes*. S_{SQ} and L_{SQ} represent respectively the size of each entry in the scheduling queue (SQ) and the queue length of the SQ.

The data memory consumption of the multithreaded scheduler can be evaluated as follows:

$$E_2 = Size(DATA/BSS) + Size(TCB) + Size(STK)$$

$$= M_{DATA/BSS} + S_{TCB}*N_{TCB} + S_{STK}*N_{THRD}$$

in which the S_{TCB}, N_{TCB}, S_{STK} and N_{THRD} represent respectively the structure size of the *thread control block* (TCB), the number of the thread TCBs, the size of the thread stack and the number of the allocated threads.

In Table 3.3, the date memory consumption of the different schedulers is listed. In the *MIROS*, the flag polling mechanism is used for the event-driven scheduling, thus no scheduling queue exists. In *TinyOS*, *Contiki* and *MIROS*, the multithreading is used either only for the application (TinyOS), or only for the system preemption-needed *processes* (Contiki), or dedicated to the real-time tasks (MIROS). Therefore, the number of the threads and the TCB in these OSes can be smaller if compared with those in the pure multithreaded OS like the mantisOS. Provided a scenario shown in the Table 3.3, the data memory size of the different OS schedulers can be computed.

WSN OSes	Scheduling mechanism	Data memory consumption		Scenario	Data size (bytes)	
		Event-driven	Multithreading		Event-driven	Multi-threading
TinyOS	Event-driven	$8 + 1*L_{SQ}$	N/A	$L_{SQ} = 15$, $N_{PCB} = 12$ $N_{TCB} = 8$, $N_{THRD} = 5$ $N_{TCB2} = 15$, $N_{THRD2} = 12$, $S_{STK} = 120$	23	/
	TOSThread	N/A	$22 + 16*N_{TCB} + \text{Size}(\text{STK})$		/	750
Contiki	Event-driven	$10 + 6*L_{SQ} + 8*N_{PCB}$	N/A		196	/
	Multithreading	N/A	$8 + 8*N_{TCB} + \text{Size}(\text{STK})$		/	672
SOS	Event-driven	$62+18*L_{SQ} +24*N_{PCB}$	N/A		620	/
mantisOS	Multithreading	N/A	$40 +22*N_{TCB2} + \text{Size}(\text{STK2})$		/	1810
MIROS	Event-driven	$4 + 4*N_{PCB}$	N/A		52	/
	Multithreading	N/A	$4 + 8*N_{TCB} + \text{Size}(\text{STK})$		/	668

Table 3.3: Date memory cost of different OS schedulers.

3.1.6.3. Execution Efficiency of Different OS Schedulers

The execution efficiency of the scheduling primitives can be evaluated by the clock cycles, and the evaluation results are shown in the Table 3.4 and Table 3.5.

Event-driven scheduling primitives	**Clock cycles** (AVR ATmega1281)					Clock cycles of one-byte memory copy
	TinyOS	Contiki	SOS	MIROS	mantisOS	
Event post	10	28	39	17	N/A	8
Task dispatching	46	36	53	26		

Table 3.4: Clock cycles of different event-driven scheduling primitives.

Multithreaded scheduling primitives	**Clock cycles** (AVR ATmega1281)				
	TinyOS TOSThread	Contiki multithreading	SOS	MIROS	mantisOS
Thread switch	77	93	N/A	102	99
Selection of next thread	26 (RR scheduling)	22 (RR scheduling)		5+9n (RMS scheduling)	72 (RR scheduling in multi-level queues)

Table 3.5: Clock cycles of different multithreaded scheduling primitives.

In the MIROS, the simple flag polling mechanism is used for the event scheduling, thus the clock cycles of the event posting and the task dispatching primitives are small. As for the thread scheduling, the static priority-based RMS algorithm is implemented in MIROS. To select the next thread, the clock cycles is (5+9n), where n represents the searching steps in the thread queue. In case that n is 3 (thread number is small in MRIOS), the clock cycle will be 32.

3.1.6.4. Discussions

The code size of the MIROS hybrid scheduler is not large, this is because the MIROS event-driven scheduler only needs to schedule the non-RT tasks while the multithreaded scheduler only needs to schedule the RT tasks. As a result, the implementations of both these two schedulers are simplified.

The code memory size and data memory size of SOS scheduler is larger if compared with that of the other event-driven OSes, this is because SOS is a dynamic module-based OS. In SOS, the complicated module management and indirect access mechanisms should be implemented.

More stacks exist in the multithreaded OS, thus the RAM usage of the multithreaded OS is commonly larger than that of the event-driven OS. Therefore, the decreasing of the stack number is significant for the optimization of the memory cost in the multithreaded WSN OS.

By means of the hybrid scheduling, the number of the stacks in the MRIOS decreases greatly. Consequently, MIROS becomes a real-time OS with low data memory usage. This makes it suitable to be used even on the resource-constrained WSN platforms.

Optionally, the MIROS multithreaded scheduler can be developed as a configurable component. If the real-time requirement is not needed by the applications, this component needs not to be built. In this case, the code size of MIROS can be reduced further.

3.2. Dynamic Memory Allocation in MIROS

As discussed in the related works of Chapter 2, segregated free list (SFL) and sequential fit (SF) are two basic dynamic memory allocation mechanisms in WSN. The drawback of SFL allocation is that the pre-reserved size of each partition is difficult to be decided. The drawback of the SF allocation is that the external fragmentation is prone to occur. To address these challenges, the heap-extendable SFL allocation and the memory-defragmented SF allocation are implemented in MIROS.

3.2.1. Heap-extendable Segregated Free List Allocation

SFL allocator needs to pre-reserve the memory partitions. If the size of a reserved partition is too small, the memory overflow problem will occur. In Contiki, the allocation will fail in case that this problem occurs. To avoid this problem, each partition should be reserved as large as it can be. However, the memory utilization efficiency will decrease by doing this. Moreover, the memory insufficiency problem may take place in this way. Since the WSN run-time contexts are varied, it is impossible to find an solution which can reserve all the sizes of all the partitions ideally.

To avoid the memory insufficiency problem, the size of each partition is not reserved to the largest value that may be needed. Instead, it is reserved to the moderate value which can meet the requirements of most WSN applications. In case that a partition is overflowed, the allocation will then not be failed, but continues in the extended heap space.

In Figure 3.8, the memory structure of MIROS is shown. After the different partitions are pre-reserved, the left memory space will be used both for the heap extending and the run-time stack. The heap and the stack increase in two opposite directions. If the allocation from a given partition is failed, this allocation can continuously be performed in the extendable heap. By so doing, the MIROS segregated free list allocator can adapt flexibly to diverse WSN run-time contexts, and the memory utilization efficiency can be improved.

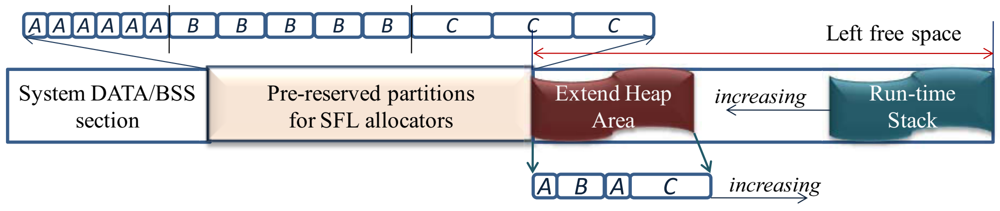


Figure 3.8: MIROS heap-extendable segregated free list allocation.

3.2.2. Memory-defragmented Sequential Fit Allocation

A key challenge for the SF allocation is that a set of external memory fragments can appear after a set of allocation and de-allocation operations. To solve this problem, the memory-defragmented mechanism is proposed in the MIROS SF allocator. Currently, two memory defragmentation approaches have been realized: the proactive memory defragmentation and the reactive memory defragmentation (concepts motivated by the proactive and reactive routing protocols).

3.2.2.1. Proactive Memory-defragmented Mechanism

Memory fragments are defragmented once they are appeared in the MIROS proactive memory-defragmented mechanism. With this mechanism, the fragments can be prevented to occur, and the new allocation can be performed immediately with constant response time. In Figure 3.9, four objects A, B, C, D are allocated. If a new object needs to be allocated, it will be performed from the starting address of the free memory space. If an allocated object needs to be de-allocated, the coalescence will be done to the adjacent chunks of this object, e.g., after object B is de-allocated, the memory areas of C and D will be coalesced in order to make the free memory space be continuous without fragments (Figure 3.9b). Since the addresses of some objects will change after the coalescing operation, all the allocated objects need to be accessed indirectly by the reference pointers. After the coalescence is completed, the update to these reference pointers can keep the allocated objects still be accessible.

The advantage of this allocation mechanism is that the fragments can be defragmented, thus the memory resources can be utilized efficiently. Moreover, the new allocation can be completely quickly with constant time. However, each time an object is de-allocated, the memory coalescence needs to be performed, thus the de-allocating overhead is high.

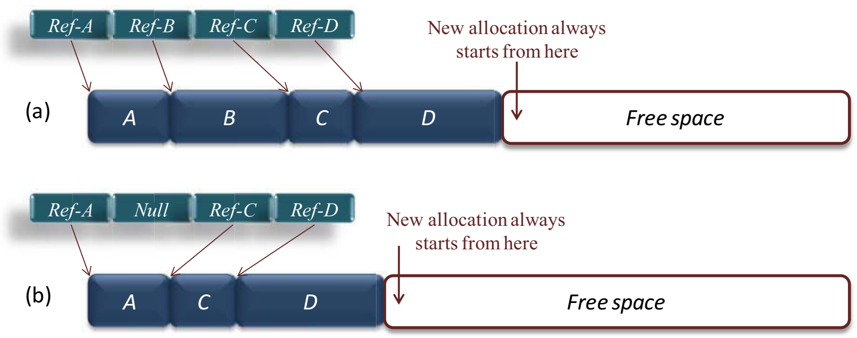

Ref-A
Ref-B
Ref-C
Ref-D
New allocation always starts from here
(a)
A
B
C
D
Free space
Ref-A
Null
Ref-C
Ref-D
New allocation always starts from here
(b)
A
C
D
Free space

Figure 3.9: MIROS proactive memory-defragmentation sequential fit allocation.

3.2.2.2. Reactive Memory-defragmented Mechanism

Reactive memory-defragmented mechanism is different from the proactive memory-defragmented mechanism in that the fragments are not defragmented every time the allocated object is released. Instead, they are defragmented only if the first time SF allocation is failed. That is, when there is not enough continuous free memory left for the new allocation. For example, in the Figure 3.10ab, if object B is released, objects A, C and D need not to be coalesced. And then, if a new 30-byte object E needs to be allocated, it can be done from the free 80-byte space. Later, if a new 60-byte object F needs to be allocated, the allocation will fail as there is no continuous free space that is larger than 60 bytes. However, the total free memory size (18+50=68 bytes) is larger than 60 bytes, thus the allocation can be performed successfully in case that the separated memory fragments are defragmented. And at this time, the defragmentation operation will be done to the different memory fragments. After this defragmentation, the allocation of object F will be tried another time and be done successfully (Figure 3.10d).

The same as proactive memory-defragmented mechanism, the reference pointers also need to be used in the reactive memory-defragmented mechanism. After the fragments are defragmented, the update to the related reference pointers must be done as well.

Compared with proactive memory-defragmented mechanism, the de-allocation overhead of reactive memory-defragmented mechanism is lower since the memory coalescence is needed rarely during the de-allocation process (needed only when two free areas are adjacent). However, the new object allocation overhead in the reactive memory-defragmented mechanism is higher as the allocator needs to search for an appropriate free block in the free list (In proactive memory-defragmented mechanism, the new allocation can be done directly from the starting address of the free space). As for which kind of defragmentation mechanism to be used, it depends on the run-time contexts.

3.2.3. Performance Evaluation

In Figure 3.11 and Table 3.6, the code size, execution efficiency and memory utilization efficiency of different dynamic allocators are evaluated. In Table 3.6, the symbol F represents the steps of searching for a free block in the Contiki SFL allocator (Contiki SFL uses the block flag mechanism other than the free list mechanism to manage the free memory), L represents the steps of searching for an available entry in the free list, and S represents the size of the memory to be shifted during the coalescing operation.

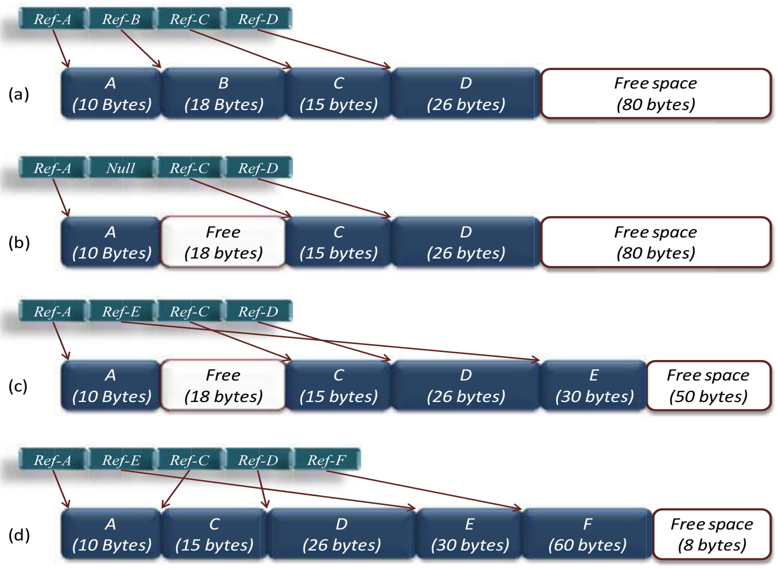


Figure 3.10: MIROS reactive memory-defragmentation sequential fit allocation.

From these evaluation results, it can be concluded that the code size and the allocation overhead of the dynamic allocators are not very high, thus the dynamic allocation approach is feasible to be used even on the resources-constrained WSN platforms. This result has also been proved in the article [151]. In this article, the authors insisted that the costs of the dynamic allocation are simply overestimated.

Different allocation mechanisms have different strong points and weak points, and they strike the tradeoff among the allocation response time, the execution overhead and the memory utilization efficiency. The allocation response time is concerned with the real-time performance, the execution overhead is concerned with the energy cost, and the memory utilization efficiency is concerned with the memory constraint problem. As for which approach should be selected, it depends on the practical contexts. If the memory resource on the WSN platform is not highly constrained (e.g., the SunSPOT node which has 512 KB RAM) and the real-time tasks exist in the system (e.g., the periodic signal sampling tasks), the MIROS SFL approach can be an ideal choice. This is because the SFL approach has fast allocation response time and low allocation and de-allocation overhead, thus the real-time requirements can be better satisfied and it will cost fewer energy resources. However, if the memory resource on the nodes is precious (e.g., the Mica2 node with only 4 KB RAM) and the application tasks have a loose requirement to the real-time guarantee (e.g., the precise agricultural application in which the nodes only need to sample the environmental data several times a day and transmit the sensing packets to the base station), the MIROS reactive SF approach can be used. With this approach, the memory fragments can be defragmented when needed. Consequently, the allocation failure resulting from the memory insufficiency problem can be eased. As for the MIROS proactive SF approach, its drawback is high de-allocation overhead, but its advantage is fast allocation time and defragmentation support. Therefore, it is suitable to be used for the real-time WSN applications in which the nodes are constrained in the memory resources (e.g., the MicaZ) but sufficient in the energy resources.

Nevertheless, the design and implementation of a new allocation mechanism that is fast in allocation time, low in allocation overhead, yet high in memory utilization efficiency still requires further study.

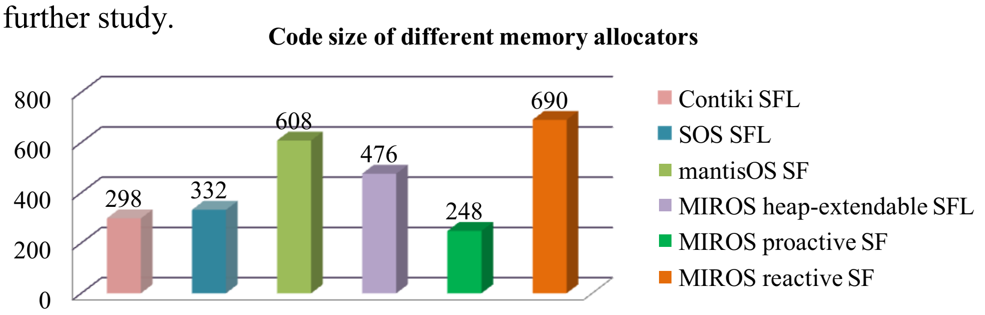


Figure 3.11. Code size of different dynamic memory allocators.

Allocation Mechanism	Allocation Status	Execution Efficiency valuated by the Cost of Clock Cycles)				Fragments (Internal or External)	Memory Utilization Efficiency
		Allocation		De-Allocation			
		Clock Cycles	Cost	Clock Cycles	Cost		
Contiki *SFL*	N/A	23 + 12**F*	Low	26 + 10**F*	Low	External	Low
SOS *SFL*		34	Low	38	Low	Both exist	Low
MantisOS *SF*		52 + 15**L*	Medium	37 + 16**L*	Medium	External	Medium
MIROS *SFL* (heap-extendable)	Regular	36	Low	32	Low	External	Medium
	In extended heap	45 + 16**L*	Medium	43 + 18**L*	Medium		
MIROS *SF* (proactive defragmentation)	N/A	36	Low	32 + 20**S*	High	External, but can be defragmented	High
MIROS *SF* (reactive defragmentation)	Regular	56 + 12**L*	Medium	45 + 15**L*	Medium	External, but can be defragmented	High
	Fragments need to be defragmented	92 + 12**L* + 15**S*	High	45 + 15**L*	Medium		

Table 3.6: Feature comparison of different allocators.

3.3. Timers, Inter-process Communication and Network Protocols

3.3.1. Software Timers

Software timer is the counter which requests some actions to be taken when the counter value meets a preset condition. It can be used for the timing control operations, such as the thread sleeping and the periodical tasks. Commonly, software timers are implemented on the base of the hardware PIT (periodic interruption timer).

The data structure of the MIROS software timer is shown in Figure 3.12. Timers can be classified into two modes: one-slot timer and periodic timer. The one-slot timer will be deleted once it is expired. Whereas the periodic timer will be reset and restarted once it is expired.

A timer queue is commonly used to manage the timers. Once a timer is started, it will be inserted into this queue. Every time the PIT is triggered, the timer queue will be checked to see whether the timers have been fired or not. If yes, the corresponded callback function will be executed.

```
typedef void (*time_cb_t)(void);

/* timer structure in MIROS */
typedef struct _Timer_t
{
      struct _Timer_t *next;
      uint32_t sysTimeLabel;
      uint32_t interval;        /* timer counter. */
      time_cb_t callback;       /* callback function when timer is fired. */
      uint8_t mode;             /* timer mode: TIMER_ONE_SHOT or TIMER_REPEAT. */
} Timer_t;
```

Figure 3.12: Data structure of MIROS software timer.

3.3.2. Inter-process Communication

Inter-process communication (IPC) is a mechanism implemented for the data exchange among the multiple tasks. It can provide an environment which allows the tasks to cooperate with each other, such as sharing the information, speeding up the computation and achieving the modular design. Commonly, the IPC can be realized by using the signal, socket, message queue, pipe, shared memory, file sharing and so on.

In MIROS, the message queue and "channel/pipes" mechanisms are used for the implementation of the IPC mechanism. And the MIROS IPC can be applied not only among the different tasks running on one core, but also among the different tasks running on different cores (on the multi-core WSN platform).

All the IPC operations in the MIROS are realized through the universal interfaces ―*send*” and ―*recv*”. In this manner, not only the programming complexity on the WSN nodes can be simplified, but also the application porting works among the different cores can be eased.

3.3.3. Network Protocols

The Atmel ZigBee and MAC IEEE 802.15.4 network protocols are currently ported and applied in MIROS.

ZigBee [4] is a low-cost, low-power, wireless network specification. It is used to create the personal area networks that are built from small, low-power digital radios. It can be widely used in the wireless control and monitoring applications. Due to its low cost feature, the long lifetime can be achieved with small batteries.

The Atmel ZigBee software [141] is fully compliant with the ZigBee standards. It provides an augmented set of APIs which, while maintaining compliance with the ZigBee standard, offer extended functionality to provide the convenience and ease-of-use to the developers.

The Atmel MAC software [142] is developed on the basis of IEEE 802.15.4. It follows a layered approach based on several stack modules. It has the following key features:

- Allows a highly flexible software configuration
- Supports different microcontrollers and platforms/boards, supports different IEEE 802.15.4 based transceivers and single chips, allows easy and quick platform porting
- Supports star networks and peer-to-peer communication
- Supports non-beacon and beacon-enabled networks

3.4. Multi-core Technology for Energy Conservation, Reliability Improvement and New OS Debugging Way

Multi-core sensor node is different from the single-core node in that two or more independent microcontrollers are integrated cooperatively on the same circuit die, and some circuitry can be shared among the different cores. Currently, most sensor nodes are still single-core ones, including the popular nodes such as mica, micaZ, TelosB and EYE. However, the development of WSN nodes trend towards the multi-core platform, this is because some WSN challenges such as the reliability and the context awareness can be addressed better by using the multi-core sensor nodes.

Compared with the single-core WSN platform, the multi-core platform can take advantages in several aspects:

1). Tasks can be distributed: Since different cores can work in parallel, the tasks on the multi-core WSN node can be split and distributed to the different cores. By doing this, some challenges existed in the single-core platform can be addressed well, e.g., in some cases, two hard real-time tasks may become active simultaneously and need to be processed immediately. If the single-core platform is used, these tasks may not be able to be completed within their deadlines. However, if multi-core WSN node is used, these two tasks can be distributed onto two different cores and be processed concurrently. By this means, the deadlines of these tasks can probably be guaranteed.

2). Be context aware, and thus be energy efficient: As different microcontrollers have different characteristics and own different advantages, the multi-core platform can self-configure to work in different working modes and adapt to the different run-time contexts flexibly. By this means, the multi-core nodes can become context aware, and the limited energy resources on the WSN platform can be utilized more efficiently.

3). Be more reliable: If only one core is embedded on the sensor node, this node will be failed in case that this core runs out of work. However, if multi-cores are embedded, the node reliability can be improved, e.g., if the master core works improperly, the slave core can monitor this and take some recovery measures, such as restarting the master core or substituting the master core to perform some tasks. By this way, the reliability of the WSN system can be improved.

In this section, the works of using the multi-core hardware technology to prolong the WSN nodes' lifetime, improve the node run-time reliability and achieve a new effective debugging approach are presented.

3.4.1. Multi-core Hardware Technology for Energy conservation

3.4.1.1. Concepts

WSN nodes are prone to be deployed in some harsh environments where humans cannot access, thus the lifetime of sensor nodes need to be long. If the nodes' lifetime is long enough, the labor work of bringing back the nodes at intervals for the battery changing can be avoided. However, most WSN nodes are powered with small-size batteries, and the energy resources are limited. Therefore, the energy conservation mechanism becomes significant in the WSN. In many WSN OSes such as Contiki and SOS, the sleep/wakeup mechanism is used to achieve the energy conservation. This mechanism is effective, but not adequate for the WSN proliferation. To conserve the energy resources more efficiently, the combination of both the software and the multi-core hardware technologies is used. With these technologies, a WSN node can be context aware and thus be more efficient in the energy utilization.

The concept of using the multi-core technology to reduce the energy consumption in the MIROS derives from the experimental results that different cores are energy efficient in execute different tasks. Commonly, the cores with powerful computation ability are more energy-efficient to execute the complicated tasks while the cores with the less computation ability are more energy-efficient to execute the lightweight tasks. In Table 3.7, the energy consumed by executing different tasks on the different cores is experimented. In this experiment, the 32-bit ARM core is more efficient to execute the complicated task, e.g., the signal processing. While the AVR core is more efficient to execute the simple task, e.g., the sensor data collection. Due to these features, a multi-core WSN node can be developed. On this node, different types of cores can be integrated. During the run-time of a task, the core which is the most energy efficient to execute this task can be configured to be active while the other cores can be configured to be inactive (fall asleep or be powered off). By this means, the WSN node can adapt flexibly to the different contexts and the energy resource can be utilize more efficiently.

Provided that N cores and M tasks exist on the WSN node. If these tasks are executed by the single core nodes, the energy consumed by the cores of these nodes will be:

$$E_{\text{core-1}}=\sum_{i=1}^{M}E_{\text{core1-task}i};\quad \ldots\ldots \quad E_{\text{core-}N}=\sum_{i=1}^{M}E_{\text{coreN-task}i}$$

However, if the tasks are executed by the multi-core WSN node, the consumed energy will be:

$$E_{\text{multi-core}}=\sum_{i=1}^{M}\min(E_{\text{core1-task}i}, E_{\text{core2-task}i}, \cdots, E_{\text{coreN-task}i})$$

Since $E_{\text{multi-core}} \leq \min(E_{\text{core-1}}, E_{\text{core-2}}, \ldots\ldots, E_{\text{core-N}})$, the multi-core node is more energy-efficient.

Platforms	Tasks for the measurements	Execution status (voltage: 3V)		Energy consumption (mJ)
		Current	Time cost	
Single-core AT91SAM7Sx	Temperature&light sensor device sampling	22.1 mA	896 ms	59.4
	Signal processing task/ pure instructions execution	19.7 mA	5.0 ms	0.296
	Flash programming (10 bytes)	20.9 mA	6.0 ms	0.376
	Sleep	0.2 mA	N/A	N/A
Single-core ATmega1281	Temperature&light sensor device sampling	15.9 mA	900 ms	42.9
	Signal processing task/ pure instructions execution	9.9 mA	268 ms	7.96
	Flash programming (10 bytes)	16.3 mA	10.1 ms	0.49
	Sleep	40 uA	N/A	N/A

Table 3.7: Energy cost of tasks execution on different microcontrollers.

3.4.1.2. Implementation of Multi-core WSN nodes

The cores on the multi-core node can be classified into two kinds: the working cores and the management core. The working cores are commonly very different from each other so that the multi-core node can adapt better to the diverse contexts. The management core is ideally the one which is low in the energy cost and high in the reliability, its responsibility is to configure the working cores to run in different status (active or inactive). By this way, the multi-core WSN node can work in different modes according to the different run-time contexts, and the energy resources can be utilized in the most efficient way.

One kind of multi-core WSN node implemented in our work is the E2MWSN board. In Figure 3.13 and Figure 3.14, the block diagram and circuit board of E^2MWSN are shown respectively. E^2MWSN consists of three cores: the low power 8-bit AVR ATmega1281 [144], the low power 32-bit ARM AT91SAM7Sx [145] and the ultra-low power iGLOO nano FPGA [146]. These three cores share the I^2C, UART and SPI resources on the board, and each one of them can be configured to run as the master of these communication buses. Commonly, the ATmega1281 and AT91SAM7Sx cores act as the working cores, and the iGLOO core acts as the management core.

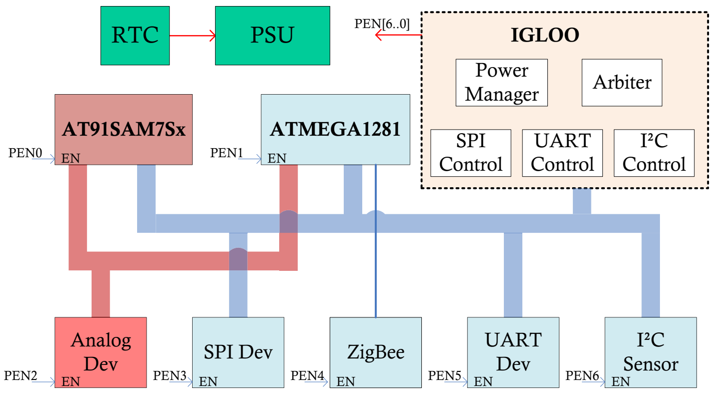


Figure 3.13: Block diagram of multi-core board E^2MWSN.

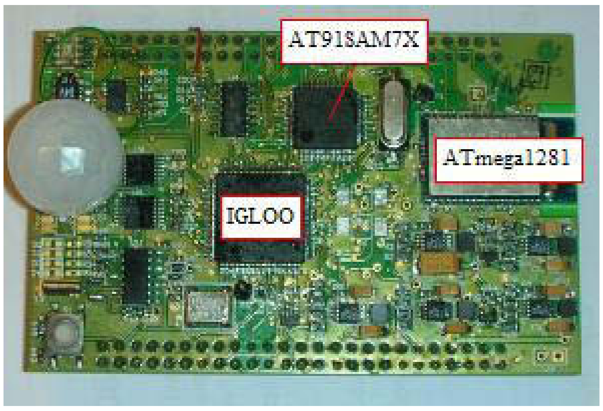


Figure 3.14: Prototype board of multi-core node E^2MWSN.

Different cores in the E2MWSN have different features. ATmega1281 core has limited computation ability. It commonly runs at only 8 MHz and delivers about 8 million instructions per second. Thus, it is more energy-efficient to execute of the simple tasks, e.g., the sensor data sampling. AT91SAM7Sx core is more powerful than ATmega1281 core. It can run at 48 MHz and deliver about 43 million instructions per second. Therefore, it is more energy-efficient to execute the complex computational tasks, such as the data compression, the data encryption and decryption and the signal processing. IGLOO core is a low-power nano FPGA from Actel. It is cheap, ultra low power, small form manufacture and good in the lead-time. It is designed to meet the demanding power and size requirements of portable and power-conscious embedded devices. With the configurability of the IGLOO nano FPGA, the circuit connection among the different cores on E2MWSN can be adjusted without any wired change. By doing this, E2MWSN can be configured to work in different modes. In Table 3.8, different kinds of E2MWSN working modes are listed.

Working modes	**Status of each core on E2MWSN (ON: power on. OFF: power off)**		
	AT91SAM7Sx	**ATmega1281**	**IGLOO**
Single-core AT91SAM7Sx	ON	OFF	OFF
Single-core ATmega1281	OFF	ON	OFF
Single-core IGLOO	OFF	OFF	ON
AT91SAM7Sx plus ATmega1281	ON	ON	OFF
Sleep mode (RTC module is on)	OFF	OFF	OFF

Table 3.8: Different working modes of multi-core node E2MWSN.

3.4.1.3. Performance Evaluation

To evaluate the energy conservation of E2MWSN platform, the energy consumptions of different tasks executed on different single-core platforms are measured, shown in the previous Table 3.7. From this measurement, it can be observed that: it is more energy-efficient to use the 8-bit ATmega1281 core to execute the simple task like the sensor device sampling. While it is more energy-efficient to use the 32-bit AT91SAM7Sx core to execute the complicated task like signal processing. As a result, the multi-core E^2MWSN node can be more energy-efficient if compared with the single-core WSN platform, this is because the multi-core E^2MWSN can adapt the working modes in terms of the tasks to be executed.

By experiment, it is computed in theory that: if powered by a pair of AA batteries and the temperature and light sensors sampling frequency is set to 3 minutes, the lifetime of multi-core E2MWSN is 1270 days. The lifetime of the single-core AT91SAM7Sx node is 382 days.

While the lifetime of the single-core ATMEGA1281 node is 825 days. Therefore, the lifetime of WSN nodes can be prolonged in case that the multi-core E2MWSN node is used.

3.4.2. Multi-core Hardware Technology for Reliability Improvement

In past research experience, it was proved that 30% to 50% Live nodes [152] failed after being deployed for two months, although these nodes worked properly for long periods when tested in the lab desktop. Since most WSN nodes are deployed in hostile environments where they are difficult to be brought back for repairing, the improvement of node reliability becomes a key research challenge. Currently in the MIROS, this multi-core hardware technology is also used to improve the node reliability.

One way of using the multi-core hardware technology to improve the node reliability is the usage of the redundancy approach. For example, on the EMWSN node, the sensor peripherals can be managed by both the AVR core and the ARM core. Commonly, the AVR core is in charge of the sensor data collection work. In the case that the AVR core fails, the ARM core can substitute it to continue this task, thereby improving node reliability.

In addition to the redundancy method, another way to improve the node reliability is to use a more reliable and ultra-low power core (management core) to manage a more powerful but less reliable and high power core (working core). Currently, this approach has been used on the multi-core node iliveT. On the iliveT, two cores are equipped: the 8-bit working core AVR ATmega1281 and the 4-bit management core nanoRisc. The AVR core is more powerful and can be used to perform WSN tasks, and MIROS runs on this core. The nanoRisc is low-end and cannot be competent for most WSN tasks. However, it is ultra-low power and highly reliable, and is thus ideal to be used as the management core to configure the work modes of the working core.

The following is an example scenario: the working core AVR wakes up every six hours to collect the environmental data, and then transmit the data packet to the local server. To achieve this task, a periodical six-hour timer is set on the management core nanoRisc. Every time the periodical timer is fired, the power will be supplied to the AVR core. Once powered, the AVR core starts collecting the environmental data, and then sends a handshaking signal from the GPIO ports to the nanoRisc. If the handshaking signal is received by the nanoRisc within a given time, the AVR core can be powered off or fall asleep. If not, the exception may occur on the AVR core. In this case, the nanoRisc will restart the AVR core by the power control (power off first, and then power on again). In this manner, the reliability of the data collection task can be improved.

The advantage of using this multi-core technology on the iliveT is that the reliability of the node will no longer be determined by the less reliable working core, but rather be determined by the highly reliable management core. In past experiments from two years ago, three single-core iLive nodes were deployed in the garden of ISIMA, and one of them failed only after working for two weeks. However, no nodes have failed thus far after the multi-core iliveT nodes were applied (and have been working properly for more than two years; the environmental temperature ranges from −15 degree to 35 degree). This result proves that the multi-core technology used on the iliveT is effective for reliability improvement.

Besides the improvement to the node reliability, the multi-core technology can also decrease the energy consumption of the iliveT nodes. If the single-core iLive node (only the AVR core) is used, the AVR core needs to fall asleep during the idle period. However, if the multi-core iliveT is applied, the AVR core can be powered off when it is idle, and be woken up later by the nanoRisc. Since the sleeping current of the nanoRisc core is much smaller than that of the AVR core (nanoRisc: 3.3 μA. AVR core: 130 μA), many energy resources can be conserved if the idle period is long. Although two cores are equipped on the iliveT, the manufacture cost of the iliveT does not increase significantly. This is because nanoRisc only needs to manage the working core, and can thus be chosen to be a low-end and inexpensive microcontroller.

3.4.3. Multi-core Hardware Technology for New OS Debugging Way

Due to the resource limitation on the WSN platforms, the OS debugging is difficult on the sensor nodes. In order to address this challenge, the multi-core node MiLive is designed and implemented. MiLive consists of two cores: the working core (the core which runs the WSN tasks) and the debugging core. The debugging core is active only during the debugging process, and its functionality is to assist the working core to afford most debugging works, including the buffering and analysis of the debugging data, the displaying of the resolved debugging information, etc. By this way, the debugging burden on the working core can be eased greatly, and the WSN debugging challenges resulting from the resource constraint can be addressed.

The MiLive board and its block diagram are depicted in the Figure 3.15. The AVR ATmega1281 core works as the working core, and the Raspberry Pi [150] core works as the debugging core. Raspberry Pi is selected as the debugging core because it is equipped with the powerful ARM BCM2835 microcontroller. This microcontroller can run in high frequency (700 MHz) and have abundant SRAM (512 M bytes), thus it is competent to afford the debugging tasks. With this debugging core, some traditional debugging problems, e.g., the data transmission overflow, can be avoided.

The GPIO (global input and output) ports are used for the connection between the AVR working core and the Raspberry Pi debugging core. GPIO ports are chosen because of two reasons: 1). the transmission speed of the debugging data from the GPIO ports can be high, thus the overflow problem can be avoided. 2). the overhead of transmitting from the GPIO ports is low. Thus, the execution of the debugging code will not influence the execution process of the regular code.

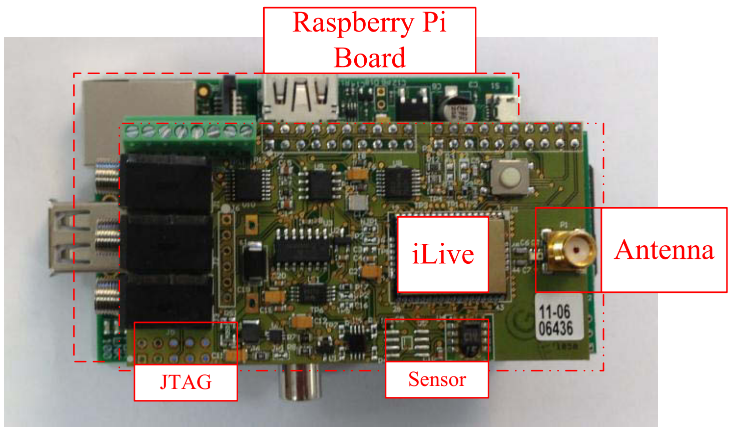


Figure 3.15: Multi-core board MiLive.

During the debugging process, the working core takes some actions, and then sends the corresponding debugging data from the GPIO ports to the debugging core. Later, the left debugging work will be undertaken by the Raspberry Pi debugging board, e.g., when a new object is allocated on the working board, 3-bytes debugging data "0xAC, 0x20, 0x1A" will be sent out from the GPIO ports. After the debugging core receives these data, it will resolve "0xAC" to the corresponding operation "OS_malloc", "0x20, 0x1A" to the allocated address 0x201A. And then, this debugging information ("New object allocation: OS_malloc 0x201A") will be sent to the WSN managers through the Internet. With this debugging mechanism, the run-time status of the WSN nodes can be monitored and debugged remotely.

3.5. Conclusions

In this chapter, the design and implementation of the new hybrid, real-time, memory-efficient and energy-efficient WSN OS MIROS is presented. MIROS uses the hybrid scheduling model to achieve the real-time scheduling with low data memory cost. In this manner, 63% data memory resources can be saved if comparing to the traditional multithreaded OS mantisOS. Besides the hybrid scheduling, MIROS implements the dynamic memory allocations, and the memory defragmentation functionality can be supported. By so doing, the probability of memory insufficiency problem can decrease, and this is significant to most WSN nodes on which the memory resources are high constrained. Since most WSN nodes are deployed in harsh environments where the human cannot access, the prolonging of the nodes' lifetime is significant. To prolong the WSN nodes' lifetime, MIROS uses the multi-core hardware technology. As a result, the lifetime of MIROS multi-core E2MWSN node can be 3.3 times longer than the single-core Live node (ARM AT91SAM7X microcontroller), and 1.5 times longer than the single-core iLive node (AVR ATmega1281 microcontroller). Moreover, MIROS uses the multi-core hardware technology to improve the node run-time reliability as well as achieve a new OS debugging way. With MIROS, the constrained platform resources (processor computation resource, memory resource, energy resource, etc.) can be managed efficiently, and the WSN proliferation can be better prompted.

Chapter 4.
Design and Implementation of MIROS Middleware

Programming and Reprogramming are the two important development processes for the WSN applications. Currently, these processes are hard for the WSN users because: 1). the hardware platforms in the WSN are diverse, different kinds of hardware platforms have been developed, such as the IMote, the Mica, the MicaZ, the sunSPOT, the TelosB and the XYZ. Therefore, it is complicated for the WSN users to program on a WSN node as they need to understand the diverse low-level hardware details. 2). different kinds of WSN OSes have been developed, including the TinyOS [51], the Contiki [52], the SOS [53], the mantisOS [55], the LiteOS [58], the RETOS [59], the nano-RK [60], etc. Each OS has its typical features and supports different application programming styles. Therefore, the learning curve exists for the application programming. 3). the WSN reprogramming is difficult. This is because most WSN nodes are prone to be deployed in the harsh environments where humans cannot access easily, thus the node reprogramming needs to be done remotely through the wireless, however, the wireless transmission is high energy cost and limited in the WSN wireless bandwidth. Therefore, it is significant to implement the mechanisms which can reprogram the WSN nodes with high success probability and low energy consumption.

In order to ease the application programming complexity and improve the application reprogramming performance, a middleware needs to be designed and implemented in the WSN. Middleware is intermediate software which locates between the WSN application layer and the low-level system space. It decouples the application from the system, implements a set of services in the system space, and provides an array of programming interfaces to the application. By using the middleware, the whole software space will be divided into two parts:

the user application space and the low-level system space, and two independent executable images will be generated (Figure 4.1b). With this space division, the persons who are the WSN experts can in charge of the system space and pre-burn the system image onto the WSN nodes. Then, the WSN users only need to focus on the application space without the necessity of considering the low-level system details. Consequently, the application programming process can be simplified. Moreover, only the application image other than the whole software image needs to be updated, thus the WSN reprogramming performance can be improved.

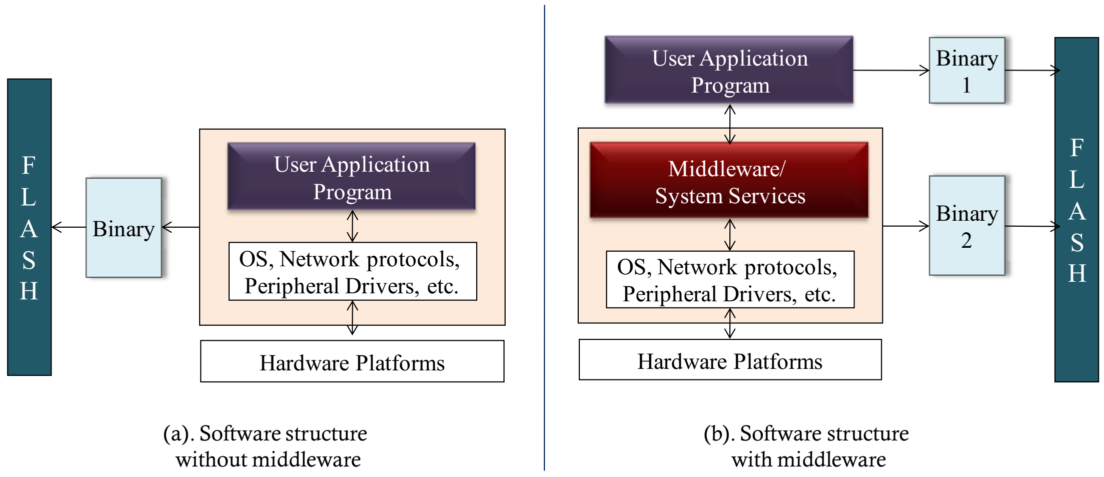

F L A S H
Binary
User Application Program
OS, Network protocols, Peripheral Drivers, etc.
Hardware Platforms
(a). Software structure without middleware
User Application Program
Binary 1
Middleware/ System Services
Binary 2
OS, Network protocols, Peripheral Drivers, etc.
Hardware Platforms
F L A S H
(b). Software structure with middleware

Figure 4.1: Software structure by using middleware

In this chapter, two kinds of middleware mechanisms are investigated. One is the use of the embedded Java virtual machine (EJVM). With the EJVM, the WSN application can be programmed by the popular and robust Java language. Moreover, the application byte code can be platform-independent. However, the memory consumption of the EJVM is high, and the byte code execution efficiency is low, making the EJVM not suitable for the high resource-constrained WSN nodes. To address this challenge, another new middleware EMIDE is developed. EMIDE is designed to have some similar functionalities as the EJVM, but be more efficient in the memory and energy consumption. Therefore it can be a substitution to the EJVM for applying on the high resource-constrained WSN nodes. Finally, the performance evaluation to both the EJVM and the EMIDE are evaluated.

4.1. Embedded Java Virtual Machine on WSN Platforms

4.1.1 Overview to Java Virtual Machine on WSN Platforms

EJVM is motivated to be used on the WSN platform because: 1). With the EJVM, the users can program the WSN applications by the popular, user-familiar, reusable, robust and prolific Java language without the need of considering the low-level system details. 2). Java language is an object-oriented paradigm. With Java, the application design, test and maintenance process can be simplified. 3). Java byte code is platform-independent. Thus, the challenge of programming in the heterogeneous WSN environments can be addressed well. 4). the WSN reprogramming performance can be improved as only the Java application image other than the monolithic software image is needed to be updated.

Currently, many embedded JVMs have been developed, including the JamaicaVM [61], the JamVM [62], the TinyVM [63], the Darjeeling VM [64], the simpleRTJ [56], the nanoVM [65], the Jwik [66], the Java Card VM [67] and so on. However, two key challenges exist for using these EJVMs on the WSN nodes: 1). most WSN nodes are constrained in the memory and energy resources while many EJVMs have non-trivial memory consumption and execution overhead. Thus, the selection of a suitable EJVM for a given WSN application becomes important. 2). most WSN applications have some typical requirements, such as the multitasking programming support and the real-time task support. However, some EJVMs cannot meet these requirements and need to be adapted or improved.

In this section, a survey to the current popular EJVMs is firstly done in the part 4.1.3. This survey can help the researchers to understand the features and challenges of different EJVMs, it can also be a guidance for the researchers to select a suitable EJVM for a given WSN environment. Before this survey is given, the basic concepts of the EJVM are presented by using the simpleRTJ as the example (part 4.1.2). In the part 4.1.4, the way of building an EJVM onto a given embedded OS is discussed. Finally, in the part 4.1.5, some conclusions to the EJVM are reached.

4.1.2 Basic Concepts of EJVM

SimpleRTJ is chosen as the example for the EJVM concept presentation because it is a clean room JVM implementation for the embedded devices. In this section, the EJVM concepts presented include the following aspects: the EJVM architecture, the Java application code structure, the EJVM run-time RAM structure, the byte code interpretation process, the interactive method between the application Java code and the system C code, the Java OS support, etc.

4.1.2.1. EJVM Architecture

The EJVM architecture of the simpleRTJ is shown in the Figure 4.2. Two component elements constitute the simpleRTJ: the simpleRTJ VM and the multithreaded JavaOS.

The simpleRTJ VM is in charge of interpreting the Java byte code. If no JavaOS exists, only the single-tasking Java application program can be executed by the simpleRTJ VM. In order to support the multitasking Java application, a multitasking JavaOS is needed. In the simpleRTJ, a multithreaded JavaOS is embedded. With this JavaOS, the multitasking Java applications can be achieved through the Java threads.

After the EJVM is applied, the Java application code can be separated from the lower system. Thus, the software space is divided into two parts: the application space and the system space. The application space is programmed by Java while the system space is programmed by C. Two spaces are built independently, and two independent images are generated. The system image is pre-burned to the WSN nodes by the WSN experts. Then, the WSN users only need to focus on the application space. After the Java application image is generated, it can be updated to the target sensor nodes through the wireless. Since only the application image needs to be transmitted, the reprogramming performance of EJVM-based system is better if compared with that of the monolithic software system (e.g., the TinyOS).

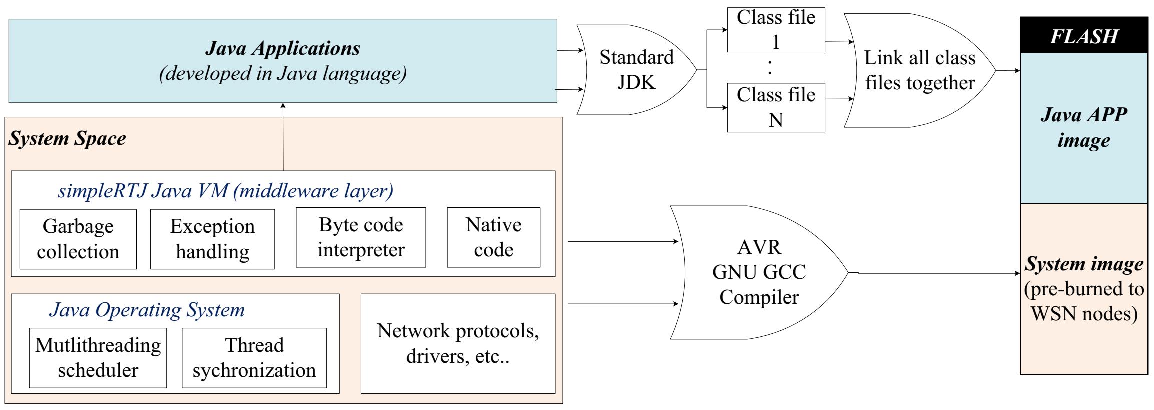


Figure 4.2: EJVM architecture of simpleRTJ.

4.1.2.2. Java Application Code Structure

The same as the JVM on the personal computer, the Java application in the simpleRTJ is built by the standard Java development kit (JDK). However, several differences exist. Firstly, the Java libraries in the simpleRTJ are optimized for using in the WSN application environments. With this optimization, the Java application image size will become smaller, thus the application reprogramming performance can be improved as only a small size code needs to be updated. Secondly, all the Java classes are pre-linked statically during the post-built process (Figure 4.3). By this static pre-linking, the separated Java classes are built up together to form a single one, and some dynamic-loading required information in the raw Java classes can be stripped. Compared with the general JVM which loads all the Java classes dynamically during the run-time, the pre-linked Java image is less flexible, because any change to the pre-linked program will cause the whole application to be re-built. However, it is more suitable for the resource-constrained WSN nodes since it can not only reduce the Java application image size, but also simplify the low-level JVM system architecture.

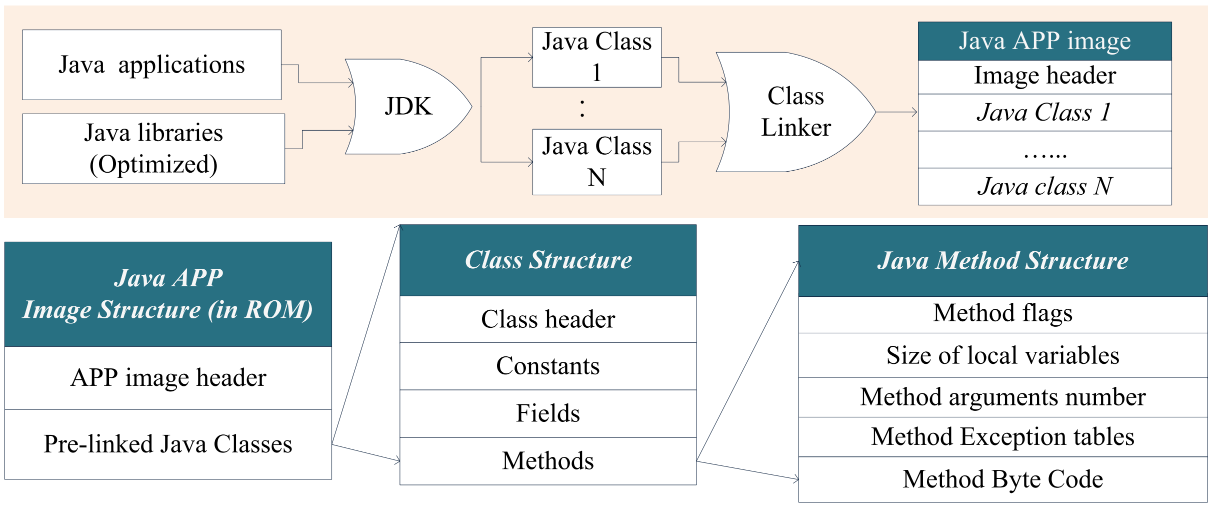


Figure 4.3: Java application image structure.

4.1.2.3. Java VM Run-time RAM Structure

The Java run-time data structure in the simpleRTJ is briefly depicted in the Figure 4.4. In the heap, the memory spaces for the Java static variables, the string tables, the thread control blocks (TCB) and the object references are reserved statically. For the left memory space, it is used by the Java object data and the Java method execution frame, and is allocated in two opposite directions.

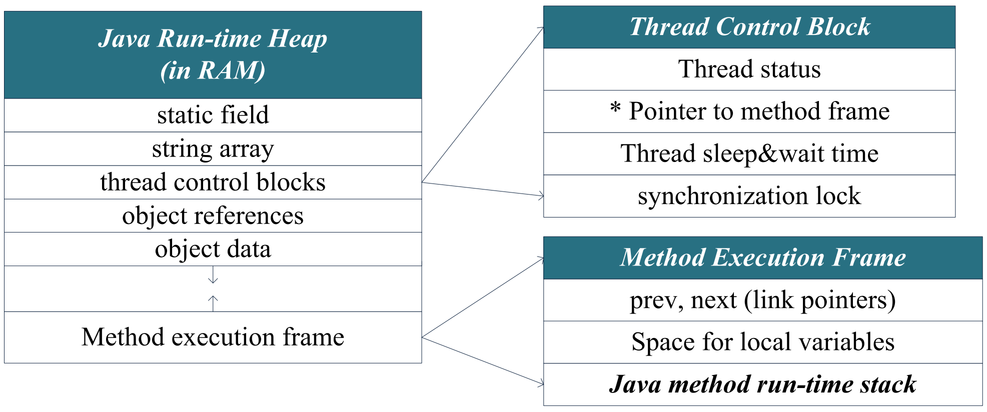


Figure 4.4: Run-time RAM structure of JVM simpleRTJ.

4.1.2.4. Bytecode Interpretation Process

The process of interpreting the Java byte code in the simpleRTJ is shown in the Figure 4.5. Before the execution of a Java method, the method arguments should be copied from the Java application image in the ROM to the method run-time context in the RAM. Moreover, the VM program counter (PC) and the VM stack pointer (SP) need to be initialized respectively to the byte code starting address and the run-time stack bottom address. After these initializations, the byte code can be interpreted one by one. Each Java byte code has a corresponding byte code handler (programmed in C). When the byte code is interpreted, the related code handler will be called.

After the byte code handler runs to completion, some further actions need to be taken in case that the handler's execution result is "invoke, native_invoke, new object creation, exception throw or method return" (Figure 4.6). The result "new object" is generated when a new Java object is defined. The result "invoke" is generated when the call from one Java method to another Java method is invoked. The result "native_invoke" is generated when the call from the Java native method to the underlying C programmed function is invoked.

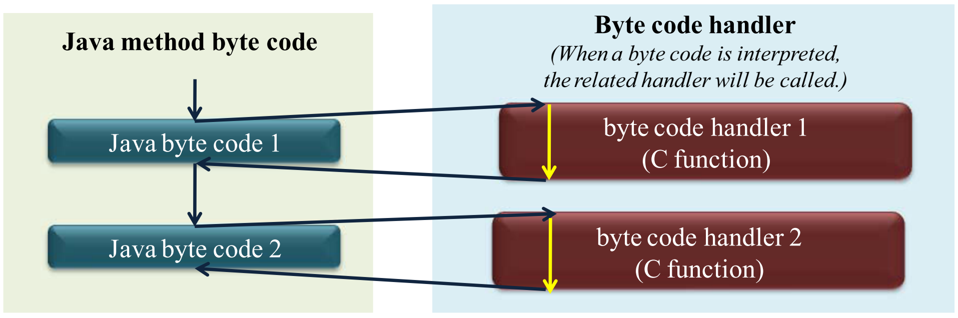

Java method byte code
Java byte code 1
Java byte code 2
Byte code handler
(When a byte code is interpreted, the related handler will be called.)
byte code handler 1 (C function)
byte code handler 2 (C function)

Figure 4.5: Java byte code and related byte code handlers.

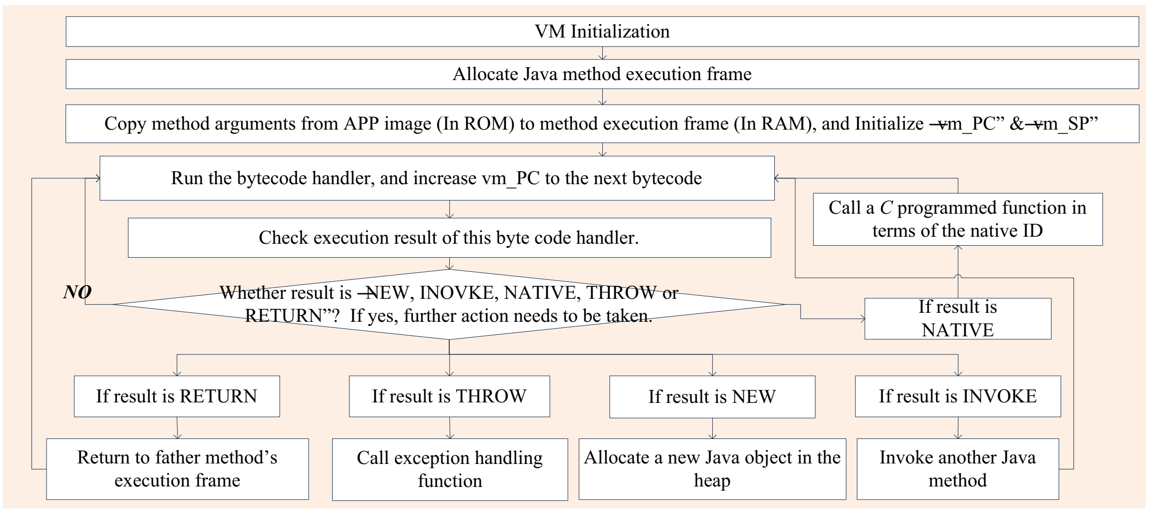

VM Initialization
Allocate Java method execution frame
Copy method arguments from APP image (In ROM) to method execution frame (In RAM), and Initialize "vm_PC" & "vm_SP"
Run the bytecode handler, and increase vm_PC to the next bytecode
Call a C programmed function in terms of the native ID
Check execution result of this byte code handler.
NO
Whether result is "NEW, INOVKE, NATIVE, THROW or RETURN"? If yes, further action needs to be taken.
If result is NATIVE
If result is RETURN
If result is THROW
If result is NEW
If result is INVOKE
Return to father method's execution frame
Call exception handling function
Allocate a new Java object in the heap
Invoke another Java method

Figure 4.6: Java bytecode interpretation process in simpleRTJ.

4.1.2.5. Interaction Between Java Application Space and System Space

Since the application is programmed by Java and the low system is programmed by C, the interaction way between these two spaces is an essential topic.

The access from the lower C code to the high-level Java code is achieved through the function interface "vm_run (method_t *method)", in which the "*method" represents a pointer to the Java method. All the Java methods' addresses are stored in the Java application image. After a Java method is loaded by the "vm_run" interface, its byte code instructions will start being interpreted.

The access from the high-level Java code to the lower C code can be achieved by the Java native methods. Java native method is used for the call from the Java method to the underlying C functions. All the Java native method are defined with an empty body block, and each Java native method has a corresponding C programmed native function in the low-level system space. After a Java native method is invoked, the corresponding system native function (C programmed) will finally be executed, e.g., the invoking to the Java native method "Thread.start" in the Java application space can finally lead to the execution of the corresponding C programmed function "java_lang_Thread_start" in the lower system space. Due to this reason, the Java native method is commonly used for the operation from the Java application program to the low-level hardware devices.

With the "vm_run" interface as well as the Java native method mechanism, the application Java space and the system C space can interact well with each other. These two mechanisms are important to be used if an EJVM needs to be built upon a C programmed JavaOS.

4.1.2.6. Java OS Support for Multitasking Java Programming

The JavaOS is essential to be used for the support of multitasking Java applications.

In simpleRTJ, a multithreaded Java OS is developed. With this OS, the multitasking Java applications can be achieved by using Java threads (Figure 4.7). By using the Java threads, several application tasks can be defined, and they can be executed concurrently by the thread switch. The switch algorithm used in the simpleRTJ is the Round Robin (RR). In terms of this algorithm, a switch request will be posted every time the RR timer is fired. By doing this, the computation resources can be divided among all the different threads.

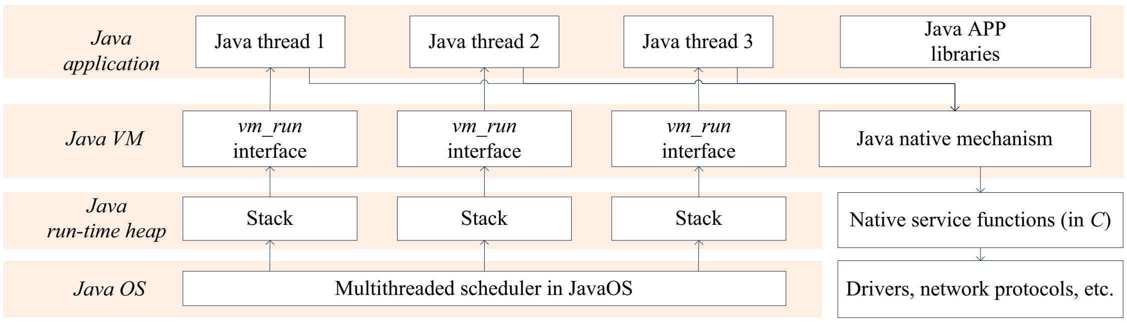


Figure 4.7: Multithreaded Java applications supported by the JavaOS.

Since the run-time context of each Java thread needs to be saved before the thread switch, every thread in the multithreaded OS should have a private stack. This stack will be used for the saving of the run-time context. By doing this, the thread can resume its execution next time. Due to this reason, the RAM consumption of the multithreaded OSes is higher if compared with that of the event-driven OSes.

In order to reduce the RAM consumption of the multithreaded JavaOS, the thread switch is done in the byte code granularity (atomic instruction) in simpleRTJ. This means that after a switch request is posted, the switch operation will not be performed immediately. Instead, it will be deferred until the current byte code handler runs to completion (that is, until the C run-time context of this handler has been released). By doing this, the byte code handler in each thread (programmed in C) will not be preempted during its midcourse execution (Figure 4.8). Thus, the handler's C run-time context, including the local variables, the processor registers (e.g., r0-r31 in the AVR ATmega1281 microcontroller) and so on, need not to be saved in the thread stacks. Instead, only the Java run-time context needs to be saved. By this means, the stack size of each Java thread can be decreased greatly, and the RAM consumption of the simpleRTJ become smaller. However, this advantage is reached at the cost of bringing down the simpleRTJ's RT performance (real-time reaction will be deferred). Thus, the current simpleRTJ is not a real real-time system. After the JavaOS is used, the simpleRTJ byte code interpretation process can be depicted as the Figure 4.9.

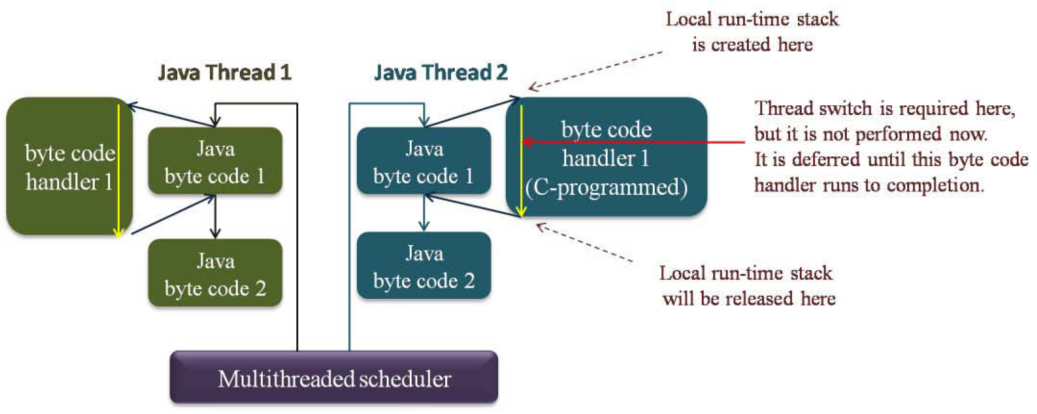

Local run-time stack is created here
Java Thread 1
Java Thread 2
byte code handler 1
Java byte code 1
Java byte code 2
Java byte code 1
Java byte code 2
byte code handler 1 (C-programmed)
Thread switch is required here, but it is not performed now. It is deferred until this byte code handler runs to completion.
Local run-time stack will be released here
Multithreaded scheduler

Figure 4.8: Java thread switch operation in simpleRTJ.

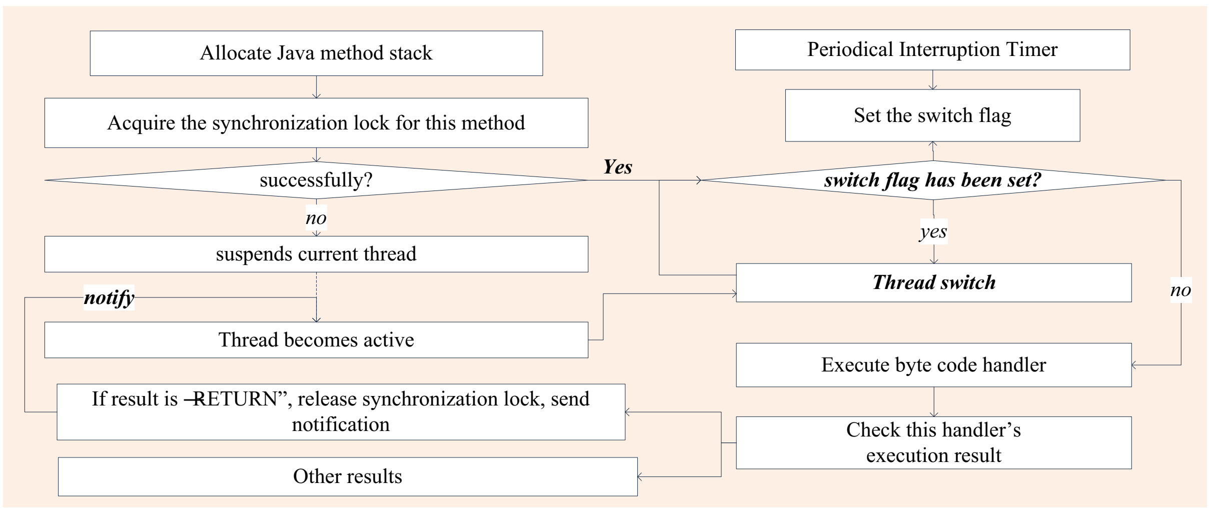

Allocate Java method stack
Acquire the synchronization lock for this method
successfully?
no
suspends current thread
notify
Thread becomes active
If result is "RETURN", release synchronization lock, send notification
Other results
Yes
Periodical Interruption Timer
Set the switch flag
switch flag has been set?
yes
Thread switch
no
Execute byte code handler
Check this handler's execution result

Figure 4.9: Multithreaded Java code interpretation process in simpleRTJ.

4.1.2.7. Exception Handling

If exception handling is supported, the occurrence of an exceptional condition during the computation process can be caught in time, and the system can be prevented to go into failure by launching the exceptional handlers. This is essential for the WSN node as sensor nodes are probably deployed in a hostile environment where it is unpractical for the humans to maintain manually.

In simpleRTJ, the Java language specified exception handlings are fully supported. With this mechanism, more robust application code can be generated.

Besides the exception handling, some other mechanisms such as the garbage collection, the run-time execution tracing, the remote debugging and so on, are also implemented in simpleRTJ.

4.1.2.8. Java Application Example

A Java-programmed WSN application example is shown in the Figure 4.10. This program can be used on the end-devices to perform the task of sensing data collection.

Since energy resources on WSN nodes are constrained, sensor nodes are configured to work in a periodical mode. Every time the node wakes up, it samples the sensor data, transmits the data packet, and then falls asleep. When the next working cycle arrives, the node wakes up again and performs the data collection work once more. In order to improve the packet transmission reliability, the acknowledgement (ACK) mechanism is used for the wireless communication. Once a packet is sent out, a timer (3 seconds) will be set. If the ACK packet is not received successfully within 3 seconds, the data packet will be retransmitted. The maximum retransmission count is set to 3. If all 3 retransmissions are failed, the node will give up and fall asleep. To improve the reliability of application code, the Java exception handling is used in this example.

```
/* A WSN application program developed by Java. */
import java.util.*;
class WSNDemo
{
   static void main(String[] args)
   {
      /* initialization. */
      boolean AckResult = false;    int retransmit_count = 0;    int dst_addr = 0x06A8;     int endpoint = 0x03;

      /* create a low-priority "common" thread to send the packet periodically */
      PktSendThr SendPktThr = new PktSendThr(COMN_THREAD, dst_addr, endpoint);
      /* create a high priority "time-critical" thread to monitor and process the ACK packet  */
      RcvAckThr AckRcvThr = new RcvAckThr( CRITICAL_THREAD);

      while(true)
      {
         /* sensor packet sending with ACK. Wait ACK for 3 seconds. */
         SendPktThr.start();        AckRcvThr.start();
         try{    Thread.sleep(3000);    } catch (InterruptedException Exc) {}

          /* wake up from sleep, come here when: A) sleeping is time out. B)  the ACK packet is received */
         AckResult = AckRcvThr.AckResult();         // check the ACK result
         if(AckResult == false)                 // result failed, packet needs to be retransmitted
         {    /* retransmit, maximum 3 times */
            if(retransmit_count < 3)           {     retransmit_count++;      continue;   }
             /* retransmit for 3 times and all transmissions are failed. Give up and fall asleep */
            retransmit_count = 0;
         }

         /* node fall asleep periodically (10 minutes), come here when: A) send success. B) all retransmissions fail. */
         try {     Thread.sleep(600000);   } catch (InterruptedException Exc) {}
      }
   }
}
```

Figure 4.10: WSN Java application in simpleRTJ.

4.1.3. EJVM Survey

In this part, a survey is made to the current popular EJVMs [56, 61-67], and the concerned VM features include the memory consumption, the multitasking and real-time support, the application pre-linking model, the exception handling (EH), the garbage collection (GC), the underlying JavaOS support, the open source as well as the VM specification. In Table 4.1, a comparison of these VM features to the different EJVMs is listed.

4.1.3.1. Memory Consumption

ROM consumption of different VMs is shown in the Table 4.1. The VMs [JamaicaVM, JamVM] have good VM performances. However, the VM code size is relatively too high, thus they are not suitable to be used on most WSN nodes, including the BTnodes, MicaZ, SenseNode, TelosB, T-Mote Sky, etc. The VMs [nanoVM, Jwik, JavaCard VM] has small code size, but poor VM capabilities.

The RAM consumption of the JVMs depends both on the VM architecture and the Java applications. The VMs [JamaicaVM, JamVM] need to be built upon the native general OS, e.g., the Linux. Thus, several MB RAM resources will be consumed, making these VMs only suitable for some resource-abundant embedded platforms, such as the powerPC and the strongARM. The other VMs [56, 63-67] have less RAM consumption so that they are available to be used on most WSN nodes. However, only simple Java applications can be supported if the RAM memory on the WSN platform is highly constrained.

4.1.3.2. Multitasking Application Programming

The support from JavaOS is essential for the JVM. With JavaOS, multitasking and real-time Java applications can be supported. For [nanoVM, Jwik, JavaCard VM], they run independently without any JavaOS. These VMs are commonly used to execute the single-thread non-RT Java applications. For [TinyVM, DarjeelingVM, SimpleRTJ], a lightweight multithreaded JavaOS is embedded inside. With the support of this JavaOS, the multitasking Java applications can be developed by using the Java threads. For [JamaicaVM, JamVM], they have strong VM performance. However, they need to be built on the native general OSes such as the Linux and the VxWorks. Thus, these VMs are only suitable to be used on the resource-abundant devices.

4.1.3.3. Real-time Performance

Real-time support is required by many WSN applications.

JamaicaVM is a JVM with full Java functionality and hard RT environments offered. The response time of JamaicaVM can be limited within 1ms. However, this high performance is achieved by the support of the native general RTOS (Linux, VxWorks, etc.) that it is built on. JamVM is similar to the JamaicaVM, a good RT performance can be realized by the support from the native general RTOS.

In simpleRTJ, the preemption can be achieved by the thread switch. Thus, the time-critical threads can be executed quickly. However, the non-RT round-robin algorithm is used for the thread scheduling in simpleRTJ. Moreover, the thread switch in the simpleRTJ is done in the byte code granularity. As a result, simpleRTJ is not a real real-time system. TinyVM and DarjeelingVM are similar to the simpleRTJ, a multithreaded JavaOS is implemented for the preemption support, and the thread switch is done in 256-bytecode granularity. That is, every time 256 bytecodes are executed in a thread, the thread switch will be performed. Since the thread preemption cannot be performed immediately, the RT performance of these VMs are also poor.

4.1.3.4. Application Linking Model

The Java application classes can be linked either in static pre-linking model or dynamic loading model.

JamaicaVM and JamVM use the dynamic loading model. The advantage of this model is that a Java class can be shared by several independent Java applications. However, the VM architecture becomes complicated if this model is used, and this increases the JVM memory consumption.

In the other EJVMs, the static pre-linking model is used. Compared with dynamic loading model, the pre-linked Java application code is less flexible, but the VM architecture complexity can be simplified. Furthermore, the application execution efficiency can be improved, and this is more suitable for the memory and energy resource-constrained WSN nodes.

EJVMs	Java APP linking model	Exception	GC	With stand-alone JavaOS	Multitasking Java APP support
Jamaica VM [61]	Dynamic loading	Yes	Yes	No, to be built on any RTOS (Linux, etc.)	Yes, by the native threads of the RTOS
JamVM [62]	Dynamic loading	Yes	Yes	No, to be built on the native OS	Yes, based on native POSIX threading
TinyVM [63]	Pre-linked	Yes	No	Yes, a multithreaded JavaOS embedded	Yes, creating threads by means of the JavaOS.
DarjeelingVM [64]	Pre-linked	Yes	Yes	Yes, multithreaded JavaOS	Yes, creating threads by the JavaOS.
SimpleRTJ [56]	Pre-linked	Yes	Yes	Yes, a multithreaded JavaOS	Yes, by threads in the JAVAOS.
NanoVM [65]	Pre-linked	No	Yes	No, run independently without JavaOS	No
Jwik [66]	Pre-linked	No	No	No	No
Java Card VM [67]	Pre-linked	Yes	No	No	No

Table 4.1 (a): Features comparison of current popular EJVMs.

EJVMs	Real-time APP support	VM specification	Code size (ROM)	Open Source
Jamaica VM	Hard RT response down to a few uS	RTSJ (RT Specification for Java)	Less than 1MB	No
JamVM	Support if built on a RTOS	Standard JVM specification	~ 220KB on PowerPC	Yes
TinyVM	Not really, atomic byte code execution	N/A	10KB on RCX micro-controller	Yes
DarjeelingVM	Not really, atomic byte code execution	N/A	~ 15KB	Yes
SimpleRTJ	Not really, switch in byte code granularity	JVM specification	18-23KB	Yes
NanoVM	No	N/A	< 10KB for AVR ATmega8	Yes
Jwik	No	N/A	< 10 KB	Yes
Java Card VM	No	JVM Specification	< 15 KB	No

Table 4.1 (b): Features comparisons of current popular EJVMs.

With the survey above, some conclusions can be drawn:

1). Different JVMs have different features, and strike the tradeoff between the performance and VM code size.

2). The VM code size, the multitasking programming ability and the real-time performance are the key factors for selecting an appropriate EJVM to a given WSN application. The code size will determine whether a JVM can be applied on the WSN nodes or not. The multitasking and real-time features will determine whether a JVM can meet the basic requirements of a WSN application or not.

3). The support from the JavaOS is important for the WSN EJVM. With the JavaOS, the multitasking or real-time Java applications are able to be supported. Since the general RTOSs (Linux, VxWorks, etc.) are too high in the memory consumption, it is essential to develop a small footprint JavaOS and embedded it inside the EJVM.

4). The current EJVMs like simpleRTJ, TinyVM and DarjeelingVM, are not real RT systems. This is because the thread switch in these VMs is done in byte code granularity. Although the RAM consumption of the EJVM can be reduced by doing this, the RT performance of these EJVMs is decreased as well.

From the conclusions above, it can be known that a key challenge for applying the EJVM on the WSN nodes is to develop a small memory footprint multitasking real-time EJVM. This challenge can be addressed in case that a small memory footprint multitasking RT JavaOS is developed. Since MIROS is a memory-efficient real-time multitasking OS, it can be used as the JavaOS for providing the multitasking and real-time support to the EJVM. As for the approach of building EJVM onto MIROS, it will be presented in the next section.

4.1.4. Integration of EJVM and MIROS

After the JavaOS is developed, the way to build an EJVM on this JavaOS is an essential topic. In this part, the example of building the nanoVM onto the MIROS will be used as the example for the discussion of this topic.

To build an EJVM on a JavaOS, the understanding of how the EJVM interacts with the JavaOS is needed. It has been discussed that the access from the Java application to the underlying OS space is achieved through the Java native method, and the access from the OS space to the Java application space is achieved through the vm_run interface "vm_run(*Java_method)". According to these principles, the nanoVM can be built upon the MIROS, with the system architecture shown in the Figure 4.11. As for the building of EJVM onto the other OSes, the same principle can be applied.

The key difference between the "simpleRTJ VM/simpleRTJ OS" system and the "nanoVM/MIROS" system is that in the latter system, when the real-time events are generated, the thread switch can be done immediately (byte code-granularity switch is not used). Thus, the "nanoVM/MIROS" system has better real-time performance.

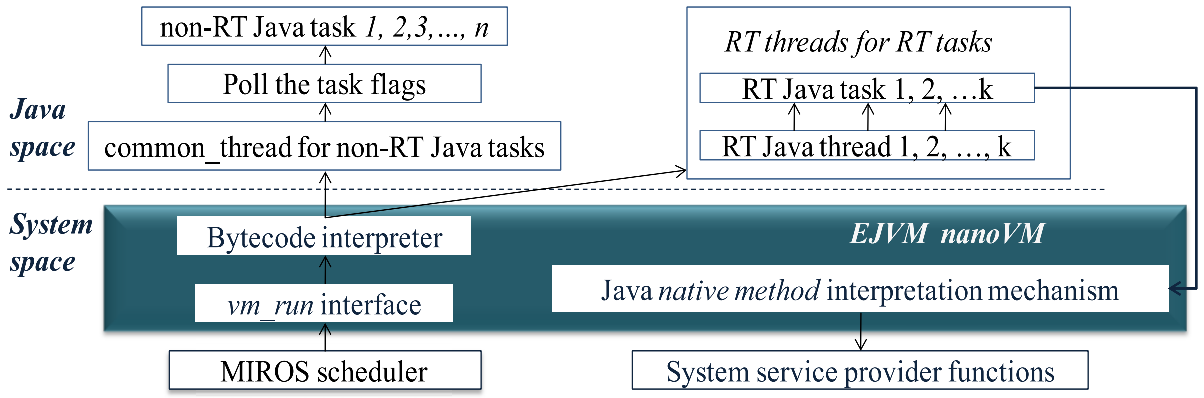

non-RT Java task 1, 2,3,..., n
Poll the task flags
Java space
common_thread for non-RT Java tasks
RT threads for RT tasks
RT Java task 1, 2, …k
RT Java thread 1, 2, …, k
System space
Bytecode interpreter
EJVM nanoVM
vm_run interface
Java native method interpretation mechanism
MIROS scheduler
System service provider functions

Figure 4.11: Software architecture of MIROS with nanoVM.

4.1.5. Discussion

With the EJVM, the WSN application programming can be simplified, and the WSN reprogramming performance can also be improved. Moreover, the multitasking and real-time Java applications can be supported in case that a multitasking real-time JavaOS (e.g., the MIROS) is developed. However, the execution efficiency of the Java byte code is lower if compared with the machine code. Thus, more energy resources will be consumed if the EJVM is applied, and this is an inherent drawback of EJVM. Fortunately, most WSN applications work in a periodic mode and the working duty cycle is short. Therefore, this low byte code execution efficiency will not have an obvious effect to the energy consumption. Nevertheless, if the EJVM is used to execute an application which is long in the working duty cycle, the energy resource limitation will become a bottleneck for the usage of EJVM.

To address this challenge above, a new middleware named EMIDE is developed. EMIDE takes some advantages from EJVM, such as the decoupling of application from underlying system and the providing of abstract application programming interfaces. However, it is different from EJVM in that it is designed to be efficient in both memory and energy consumptions. Consequently, EMIDE can be used as a substitution of the EJVM to be used on the tight memory and energy constrained WSN nodes. In the next section, the design and implementation of EMIDE will be presented.

4.2. Design and Implementation of Middleware EMIDE

In this section, the design concepts and implementation works of EMIDE will be presented. And then, the advantages of applying EMIDE will be concluded.

4.2.1. EMIDE Design Concepts

In order to utilize the advantages of the EJVM and the DLM, and avoid the drawbacks of these mechanisms, the new software EMIDE is designed and implemented. The same as the EJVM, EMIDE can separate the application space from the OS space. However, it is designed to be both memory and energy efficient, and thus can be used even on the high memory and energy constrained WSN nodes. In terms of its functional objectives, the design concepts of EMIDE are as follows:

Decoupling the application space from the system space: Like the EJVM and the DLM, EMIDE should be able to decouple the application space from the system space and function as a bridge between these two spaces.

Machine code or interpreted code: The byte code is used in the EJVM application while the machine code is used in the ContikiDL module. Although the byte code can be platform independent, more energy and time will be consumed during its interpretation process. Therefore, the machine code is chosen in the EMIDE. In this way, EMIDE can address the challenge of better using the tight resource-constrained WSN nodes.

Pre-linked code or dynamic loading code: The pre-linked application code is used in most EJVMs while the dynamic loading code is used in most DLMs. Compared with the dynamic loading code, the pre-linked code is smaller in size, and higher in execution efficiency. Moreover, the underlying system architectures can be simplified. Due to these reasons, the pre-linked approach is adopted in the EMIDE so that it can maintain a small memory footprint.

Application programming language: In most EJVMs, the language used in the application space (Java language) is different from the language used in the system space (commonly C language). As a result, two different run-time contexts should be created and this increases the complexity of the system architecture. In EMIDE, the application language is chosen to be the same as the system language. In so doing, the application code and the system code can call each other directly, thereby simplifying the system architecture.

Support of multitasking programming in the application: Multitasking programming is a fundamental requirement of the WSN application. In EMIDE, some mechanisms should be implemented to support the multitasking programming in the WSN application.

Support of callback from the system space to the application space: The callback from the system space to the application space is also needed in many cases, e.g., the low-level hardware ISR (interruption service routine) may need to be programmed in the application space. In these cases, the EMIDE should be responsible of this callback operation.

Prevention of the faults injected from the application space to the system space: Fault prevention is also needed in the EMIDE to prevent the faults injected from the application space to the low-level system space. The reliability of the EMIDE system can in this way be improved.

According to the design concepts above, EMIDE can be described as a mid-layer software that decouples the application code from the system code; acts as a bridge between the application space and the system space; supports multitasking programming in the application space; generates the pre-linked and machine-code application image; and is efficient in both the memory and energy cost. In Figure 4.12, the elementary structure of the EMIDE is briefly depicted.

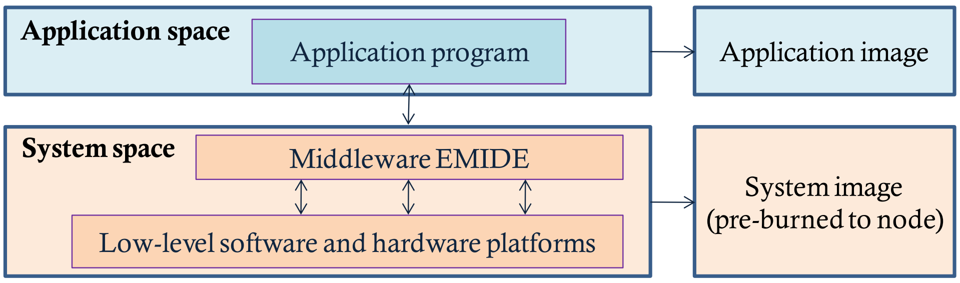


Figure 4.12: Elementary structure of EMIDE.

4.2.2. EMIDE Implementation

Several topics should be considered for the implementation of EMIDE. In this section, these topics will be presented, and the presentation is based on the AVR platform with GNU tool chains as the compiler.

4.2.2.1. Function Definitions

Two kinds of functions need to be defined in EMIDE: One is the service provider function (SPF) in the system space (similar to the C-programmed system native functions in EJVM).

This kind of functions will provide the system services that can be used by the application programs. The other is the service subscribe interface (SSI) in the application space (similar to the Java native methods in the EJVM). This kind of functions can be called in the application program to subscribe the system services provided by the SPF. Each SSI has a corresponding SPF, and the call to this SSI in the application space will lead to the related SPF in the system space to be executed (similar to the Java native mechanism in EJVM: a Java native method in the application will lead to the related system native function in the system to be executed).

The application development process of EMIDE can be depicted as the Figure 4.13. After the application program is built, a raw application binary will be generated. Then, all the SSI in this raw application binary will be pre-linked during the post-built process. As a result, all the application SSIs are linked to the system SPFs.

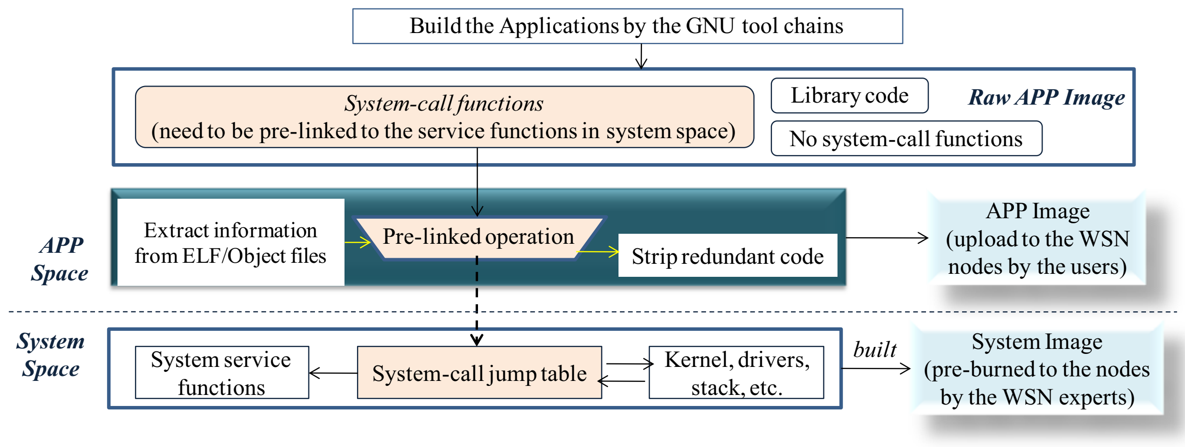

Build the Applications by the GNU tool chains
System-call functions
(need to be pre-linked to the service functions in system space)
Library code
Raw APP Image
No system-call functions
APP Space
Extract information from ELF/Object files
Pre-linked operation
Strip redundant code
APP Image (upload to the WSN nodes by the users)
System Space
System service functions
System-call jump table
Kernel, drivers, stack, etc.
built
System Image (pre-burned to the nodes by the WSN experts)

Figure 4.13: Application development process of EMIDE.

4.2.2.2. Resource Reservation

Although the application image and the system image are built independently, they run together on the same microcontroller (programmed on the same FLASH and run together in the same RAM). Thus, the RAM memory needs to be shared between these two images. In EMIDE, this sharing of the memory resource is solved by the pre-reservation mechanism. The FLASH and RAM memory required by the application image is pre-reserved, and this reservation is achieved by configuring the linker script.

4.2.2.3. Pre-Linking Mechanism

A common way to perform the link from the SSI to the SPF is to build the system space firstly. Then, a map file will be generated. In this map file, all the SPF functions' addresses will be listed. Thus, the pre-linking from the SSI to the SPF can be achieved through this list. The drawback of this method is that the application image is inflexible as any change to the system image can cause the application image to become invalided.

To make the application image be flexible, the SSIs in the EMIDE are not linked directly to the system SPFs. Instead, an intermediate function jump table is provided in the system space. By this table, the SSIs are linked indirectly to a given address in this table, and then this table will redirect the SSI call to the corresponding SPF function. By this way, the change to the system image will not affect the validity of the application image in case that the jump table is put at a fixed address and the item order in this table is not changed.

4.2.2.4. Variable Passing Mechanism

Besides the linking from the SSI to the relevant SPF, another problem to be solved is the passing of the local variables between the SSI and the SPF. EMIDE should act as a bridge to make up the variable value passing gap in case that such a passing gap exists between the application SPF and the system SSI functions.

After the application project and the system project are built independently, it is shown from the assembling code that the SSI function’s local variable values are put into the processor registers one by one (starting from [R25, R24, ...]). And at the beginning of the system SPF functions, the local variable values are extracted from the registers [R25, R24, ...] one by one. This indicates that no value passing gap exists between the SSIs and the SPFs, and this is because the same language and the same compiler are used in the application space and the system space. Therefore, the same regulation is followed although these two spaces are built independently. As for the passing of the return value from the SPF to the SSI, the mechanism is the same, and the registers [R25, R24] are used in this case.

4.2.2.5. Callback Mechanism

With the mechanisms above, the application program can access the functions in the system space, but the reverse operation (callback from the system space to the application space) is still not supported. However, this callback operation is needed in many cases. For example, the hardware interruption is triggered in the underlying system space, if an ISR needs to be programmed in the application space, a callback mechanism which redirects the function execution from the system ISR to the related application ISR needs to be implemented.

In EMIDE, this callback operation is achieved by a registration approach. If an application function needs to be called back from the system space, it needs to be registered by a registration function. After registered, the address of this application function will be passed to the system space. Then, the call back from the system function to this application function can be achieved.

4.2.2.6. Multitasking Programming Mechanism

The support of multitasking programming is important for the WSN applications. To achieve this objective, the registration mechanism is also used. Several independent application tasks can be defined in one application program and then be registered. After registered, the addresses of these application tasks will be passed to the underlying OS scheduler, and then these application tasks can be scheduled in turn.

Compared with the callback registration mechanism, the multitasking registration mechanism is different in two aspects. Firstly, different registration interfaces are used. Secondly, the callback functions' addresses are passed to the callback component in the EMIDE, while the multitasking tasks' addresses are passed to the scheduling component inside the underlying OS.

4.2.2.7. Fault Prevention Mechanism

In Java, the exception handling can be supported. However, in the C language, it cannot be supported. In order to prevent the potential faults being injected from the application program into the system run-time environment, the fault prevention mechanism needs to be implemented. In EMIDE, this is achieved by the health checking and the exception notification mechanisms.

Health checking includes the checking of the input parameters, the memory stack boundary, etc. The checking of input function parameters are done inside the SPFs. After a SPF function is called, the function parameters passed to this SPF will be checked firstly. Only after the input parameters are within a rational range, the SPF execution will continue. For example, the software timer structure should be defined as a global variable as it should keep alive until this timer is expired. If a timer is defined incorrectly as a local variable, the run-time fault can take place. To prevent this fault, the timer address will be checked before starting the timer. If the address of the timer locates within the stack space (memory area for the local variable) rather than the heap space (memory area for the global variable), this timer structure will be copied from the stack to the heap. Only after this operation, the timer can be started. The checking of memory stack boundary is similar to the checking of input parameters. At the beginning of every SPF function, the left stack size will be checked. If the left space is not enough, this SPF will not be executed. By doing this, the memory data can be prevented to be corrupted.

Once a fault is observed, the regular execution of the system will be suspended, and an exception notification will be sent to the managers. Later, the manager can reprogram a new correct program to the nodes.

4.2.2.8. Porting of EMIDE

EMIDE can be built on different WSN OSes. If it is built on the MIROS, the software architecture can be depicted as the Figure 4.14. Different from the EJVM, the access from the application to the underlying system is done through the pre-linked mechanism in the EMIDE (in EJVM, it is through the Java native method), and the access from the system space to the application space is achieved through the registration mechanism (in EJVM, it is through the vm_run(*Java_method) interface.)

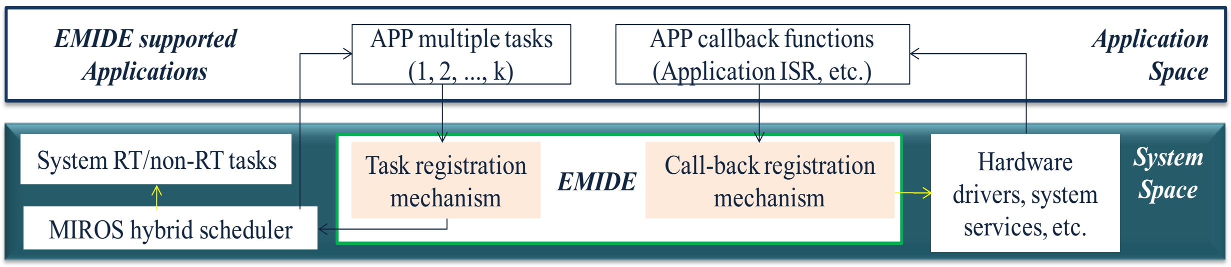

EMIDE supported Applications
APP multiple tasks (1, 2, ..., k)
APP callback functions (Application ISR, etc.)
Application Space
System RT/non-RT tasks
MIROS hybrid scheduler
Task registration mechanism
EMIDE
Call-back registration mechanism
Hardware drivers, system services, etc.
System Space

Figure 4.14: Software architecture of MIROS with EMIDE.

4.2.2.9. EMIDE Application Example

In Figure 4.15, an application example based on the EMIDE is shown. The functionality of this example is the same as that of the Java example in the Section 4.1. In the Section 4.3, the code size comparison of these two different mechanisms will be made.

```
/* application programs supported by EMIDE. */
int main(void)
{
    /* register a task to the system scheduler centre, to enable this task be scheduled by the scheduler in the system space. */
    appTask_register(packetSendTask, pktSendID);
    /* register a ISR callback function, to enable the hardware ISR programmed in the application space. */
    appCallback_register(packetRcvISR, pktRcvID);

    /* configure the WSN and start the network. */
    uint8_t Cset[3] = {11, 12, 13};        // channels that can be chosen.
    node_type('C');          // configure node to either coordinator, router or end-device.
    node_C_workingPeriod(1200);       // configure node working period to 20 minutes.
    sensor_allstart();      // open all the sensors.
    wsn_start();            // Starts WSN network.
}

/* Application ISR function, to process the Received wireless packet,  After registered, it can be called back from the system space. */
void packetRcvISR(dataInd_t *inPkt)
{
    sendToUsart(inPkt->data, inPkt->size);   // After packet is received, forward it to PC by USART
}

/* application Task, to send sensor packet with ACK mechanism. */
uint8_t packetSendTask(void *sensorPkt)
{
    static uint8_t pktRtrsCnt = 0; // the maximum retransmission counter.
    while(true)
    {
        dev_send(WIRELESS_TX_ID, sensorPkt, SENSOR_FRAME_SIZE, 0);   /* send the packet out. */
        startTimer(&AckWaitTimer);        yield_Thread(event->thrd);  /* Start timer to wait for ACK, and then yields the thread. */

        /* When timer is out or ACK is arrived, this execution will continue and the ACK result will be checked. */
        if(ACK_resCheck() == FALSE)
        {
            if(pktRtrsCnt < 3)   {    pktRtrsCnt++;        continue;     }            /* retransmission */
            pktRtrsCnt = 0;        stopTimer(&AckWaitTimer);        return FAILED;      /* all transmissions are failed. */
        }

        /* make the node to fall asleep. */
        node_sleep(600000);        // sleep for 10 minutes before next transmission
    }
}
```

Figure 4.15: EMIDE application example.

4.2.3. Discussion

Currently, less attention is paid to the pre-linked mechanism for the software development in the WSN, this is because pre-linking is considered to be inflexible. For example, in Contiki, the significance of decoupling the application from the underlying system is also realized. And to address this challenge, a dynamic linker [83] is developed. By this dynamic linker, the application program can be built into an ELF module and updated to the WSN nodes. Later, this module will be resolved and executed. Dynamic linking is chosen in Contiki because of its high flexibility. However, the experimental experience on the EMIDE proves that the flexibility of the pre-linked code is not a critical problem in case that an intermediate jump table is provided. More significantly, both the memory consumption and the code loading overhead of the pre-linked mechanism are much less than those of dynamic linking, thus this approach is more suitable for the high resource constrained WSN nodes.

In experiments, some advantages of using the EMIDE can be validated, as follows:

- Abstraction of the low-level details: With EMIDE, the users can make the application program easily without the necessity of pre-knowing the underlying software and hardware details (less learning curve).
- Some hardware development tools are no more obligatory: After EMIDE is used, some traditional hardware tools, e.g., the AVR emulator, are no more needed for the application development. With EMIDE, the application code size becomes small (hundreds of bytes). Thus, the new code can be reprogrammed through the wireless transmission (transmission of several wireless frames will be enough).
- New application can be deployed remotely through the webpage: With EMIDE, the application is decoupled from the system and becomes simple, thus it is practicable to program and reprogram the new applications remotely from the Internet by the webpage.
- Reprogramming becomes efficient in both the time and the energy consumption: With EMIDE, only the application code needs to be reprogrammed, thus the reprogramming can be completed in a short time, and less energy will be consumed. In the Atmel OTAU [71], the monolithic software image (commonly larger than 100 kilobytes) is updated for the WSN reprogramming. If Atmel ZigBee network protocol is used, the data payload of each wireless frame should be smaller than 95 bytes [72]. In this case, several thousands of wireless frames need to be transmitted for the new code reprogramming, and this process will take a long time. In the article [71], it is tested that in a network with 100 router, the entire OTAU reprogramming will take approximately 7 hours for a non-secure network, 11 hours for a secure network, and 14 hours for a high-security network. However, after the EMIDE is used, the reprogramming process can be completed in several minutes (only hundreds of bytes of code to be updated), and less energy will be cost.

4.3. Performance Evaluation

In this section, the performance evaluation to the EJVM and the EMIDE is done. SimpleRTJ and nanoVM are chosen as the EJVM examples. SimpleRTJ is chosen because it is a full-time EJVM. NanoVM is chosen because it is a tiny JVM with limited functionalities. Besides the EJVM (simpleRTJ, nanoVM) and the EMIDE, the Contiki dynamic linker ContikiDL [83] can also decouple the application from the underlying system, thus it is included in this evaluation as well. Moreover, the Atmel over-the-air-upgrade (OTAU) [71], which does not implement any middleware mechanism, is also involved in this evaluation.

The evaluation is performed on the iLive platform, and the AVR GNU tool chain is used as the compiler. As for the evaluation criterions, the software features, memory consumption, application code execution efficiency and application image size are selected.

4.3.1. Features Comparison

The features of different mechanisms are shown in the Table 4.2.

In the Atmel OTAU, no middleware is used and the application is not decoupled from the system. Thus, the application needs to be built with the system together. And several challenges exist for this mechanism: 1). the users need to be good at managing a huge software project, e.g., they need to master the skills of using the makefile and the development tool. 2). the application programming will be complicated as it will be more associated with the low-level system details. 3). the whole software image needs to be updated for the WSN reprogramming, thus the reprogramming performance is low. In terms of these reasons, it can be known that middleware is essential to be used for the application development on the WSN platforms.

With ContikiDL, the application can be decoupled from the system and built into an ELF loadable module. After the application module is loaded onto the WSN node, it will be resolved and re-linked. ContikiDL can improve the application reprogramming performance since only the application image needs to be updated. However, it has limited functionalities.

In most EJVMs and the EMIDE, the pre-linked code is used for the application image. Although pre-linked code is less flexible than the dynamic linking, it consumes less memory resources and supports higher execution efficiency.

Features	**Mechanisms**			
	EJVM (nanoVM & simpleRTJ)	**Dynamci linker ContikiDL**	**EMIDE**	**Atmel OTAU**
Application decoupled from system	Yes	Yes	Yes	No
Application programming language	Java	C	C	N/A
Application image format	Pre-linked Java bytecode	Dynamic loadable ELF file (should be dynamically linked before being executed)	Pre-linked machine code	N/A
Flexibility of application image	Average	Well	Average	N/A
Multitasking programming in Application	Yes, by Java threads	N/A	Yes, by *multitasking registration*	N/A
Mechanism to access to the system from the application	By Java *native* method	By dynamic linking	By pre-linking	N/A
Mechanism to callback from the system to the application	By *vm_run* interface	N/A	By *callback registration*	N/A
Exception handling support	Can be supported in simpleRTJ	N/A	Simple fault prevention mechanism	N/A
Garbage collection	Yes	N/A	N/A	N/A

Table 4.2: Feature comparison of EJVM, ContikiDL, EMIDE and Atmel OTAU.

4.3.2. Memory Consumption

The memory consumption of different mechanisms are listed in the Table 4.3. SimpleRTJ supports rich JVM features, but this is achieved at the cost of large VM code size. NanoVM has limited VM features, e.g., the multitasking application programming and the exception handling cannot be supported, but the VM code size is smaller. EMIDE has less memory consumption, this is because the pre-linked machine code is used, thus most EMIDE works are processed on the PC rather than the target WSN nodes.

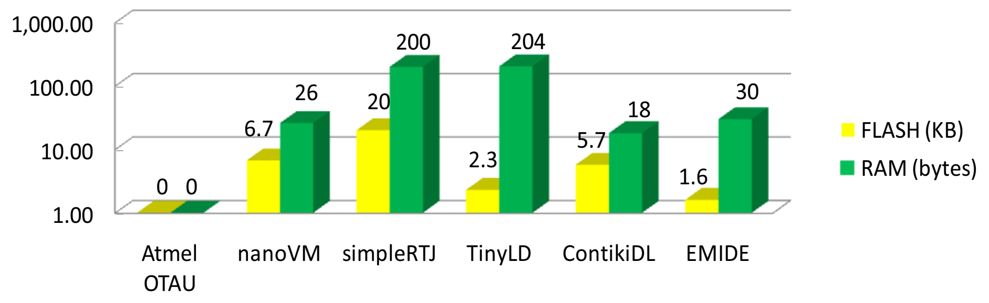


Table 4.3: Memory consumption of EJVM, ContikiDL, EMIDE and Atmel OTAU.

4.3.3. Application Execution Efficiency

In the ContikiDL, after the application ELF module is resolved and linked, the final executed code is the pure machine code, thus the execution efficiency is high. In the simpleRTJ and nanoVM, the executable application code is Java byte code. Compared with the machine code, the execution efficiency of the byte code is inherently lower. In order to compare the execution efficiency of the machine code and the byte code, the execution time between the application function and the corresponding system function in the simpleRTJ and the EMIDE is computed, and the results are shown in the Table 4.4. From this result, it can be known that the execution efficiency of the machine code is 34.6 times higher than that of the simpleRTJ Java byte code. Therefore, more energy will be consumed if simpleRTJ is used for the application execution.

EMIDE also uses the machine code, but the access from the application SSI to the system SPF is done through an intermediate function jump table, thus the application code execution efficiency is a little lower if compared with that in ContikiDL. And the proportion of the execution efficiency of these two mechanisms can be modeled as:

$$R = R_{EMIDE}/R_{ContikDLi} = (10{*}C_f{+}N_i)/N_i = 1{+}10{*}C_f/N_{i.} \qquad \textbf{\textit{(4.1)}}$$

where *Cf* mean the number of SSI in the EMIDE application program, *10* is the clock cycle of each item in the function jump table, *Ni* is the total execution clock cycles of application code (include the execution of the sub-functions). Assumed that *Ni* is equal to 900 and *Cf* is equal to 8, then *R* will be 1.089.

Application code types	Middleware mechanism	Execution information			
		Voltage (V)	Electric current (mA)	Time cost for 2000000 times execution (S)	Energy consumption (mJ)
Machine code	ContikiDL	3	10.5	14	441
Java bytecode	simpleRTJ	3	10.5	485	15277.5

Table 4.4: Comparison results of application code execution efficiency.

4.3.4. Application Image Size

Application image size is a key factor for the middleware as it will not only determine the success probability of application image updating, but also determine the energy consumption during the reprogramming process.

In order to compare the application code size of different mechanisms, the packet sending programs in the previous Section 4.1.2 (developed by Java, on the basis of EJVM) and previous Section 4.2.2 (developed by C, on the basis of EMIDE & ContikiDL) are built, and the result is shown in the Table 4.5.

If the Atmel OTAU is used, the application is not separated from the system, thus the monolithic software image needs to be reprogrammed, and the reprogramming code size is large. If EMIDE is used, the application code size is the minimum, this is because EMIDE uses the pre-linked machine code in which no interpretation or reference resolving information is included. If the ContikiDL is used, the application will be built into ELF module. Compared with the EMIDE application code, the ContikiDL ELF module is larger in the size, this is because the function and variable references should be contained in the application module for the future resolving. If the simpleRTJ is used, the application code size is relatively large. The main reason is because simpleRTJ supports the exception handling, thus a set of exception handling Java classes are linked inside the simpleRTJ application image. If the nanoVM is used, the Java application code size can be smaller if compared with that in the simpleRTJ, this is because nanoVM is not a full-time JVM, and less VM components are contained in the application image.

Since the code size of the EMIDE application is small, high reprogramming performance can be achieved.

Mechanisms	Application image format	Code size (bytes)
Atmel OTAU	Monolithic software image	114786
ContikiDL	Compacted ELF module	768
EJVM simpleRTJ	Java byte code	2472
EJVM nanoVM	Java byte code	876
EMIDE	Pre-linked machine code	182

Table 4.5: Comparison results of application image sizes.

4.3.5. Diverse Software Structures

With the MIROS, the EMIDE and the EJVMs, different kinds of software structures can be built, including the implementation of the EJVM onto MIROS, the implementation of EMIDE onto MIROS, etc. In this part, several kinds of software structures are evaluated from the aspects of multitasking application programming, real-time scheduling, automatic garbage collection, exception handling and code size, and the results shown in the Table 4.6.

Different mechanisms have different merits and drawbacks. It is not reasonable to find the best solution, but it is essential to find a suitable approach for a given WSN application. Commonly, if the memory and energy resources on WSN platforms are abundant, the full-time VM (e.g., simpleRTJ JVM) plus the MIROS can be a good combination. However, if the memory and energy resources are highly constrained, the EMIDE plus the MIROS can be a sound choice.

Structures	Features				
	Multitasking application programming	Real-time scheduling	Automatic garbage collection	Exception handling	Code size (KB)
nanoVM	No	No	Yes	No	7.8
simpleRTJ JVM/OS	Yes	No	Yes	Yes	22.3
nanoVM + MIROS	Yes	Yes	Yes	No	11.6
simpleRTJ JVM + MIROS	Yes	Yes	Yes	Yes	23.5
EMIDE + MIROS	Yes	Yes	No	No	4.8

Table 4.6: Feature comparison of different software structures

Chapter 5.
WSN Application Development

In this chapter, the development works of the MIROS applications will be introduced.

5.1. Data Collection Application

In this section, a WSN application which performs the task of environmental data collection is presented.

A popular functionality of WSN nodes is to collect the environmental data periodically and send the sensing data packets to the local server wirelessly. To improve the packet communication reliability, the ACK mechanism is commonly used. And the sensing data packet will be retransmitted if the ACK packet is not received within a given time. To prolong the lifetime of the nodes, the sleep/wakeup mechanism is used. Once the node is idle, it falls asleep to conserve the energy resources. In Figure 5.1, the flowchart of this data collection task is depicted.

After the sensor data is received by the local server, it can be either stored in the database for further analysis, or accessed through the Internet by the users (Figure 5.2). An online demo about this application can be accessed from the website: http://edss.isima.fr. (login: demo. password: demo).

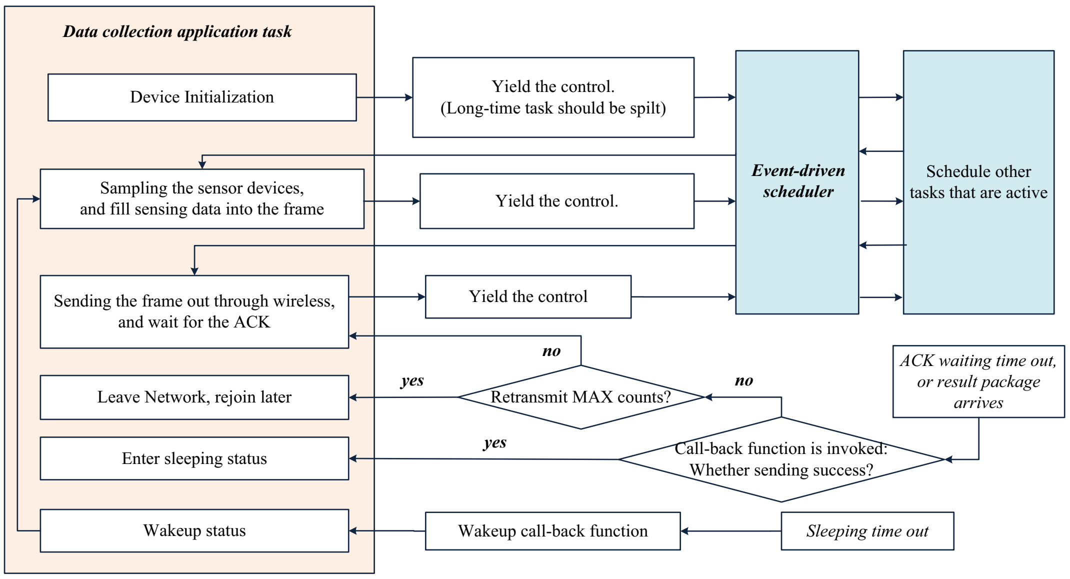

Data collection application task
Device Initialization
Yield the control.
(Long-time task should be spilt)
Event-driven scheduler
Schedule other tasks that are active
Sampling the sensor devices, and fill sensing data into the frame
Yield the control.
Sending the frame out through wireless, and wait for the ACK
Yield the control
no
ACK waiting time out, or result package arrives
yes
Retransmit MAX counts?
no
Leave Network, rejoin later
yes
Call-back function is invoked: Whether sending success?
Enter sleeping status
Wakeup status
Wakeup call-back function
Sleeping time out

Figure 5.1: Flowchart of WSN environmental data collection application.

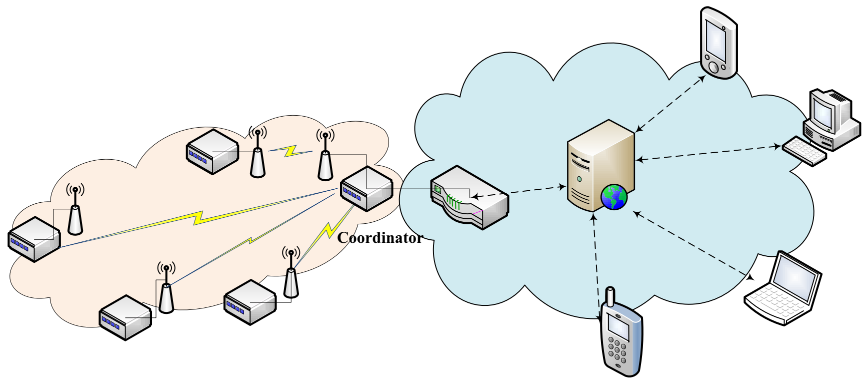


Figure 5.2: Network topology of WSN with Internet.

5.2. Related WSN Application Projects

In this section, some WSN application projects in our work will be introduced.

Precision agriculture: Currently, the iLive nodes have been deployed in the Montoldre, Allier, France for the precision agriculture applications. With the iLive nodes, the agricultural field information (temperature, soil moisture, etc.) can be monitored and collected.

Smart irrigation system: WSN technology has also been used to perform the smart irrigation in our works. Smart irrigation system (SIS) is needed as many planters lack the correct knowledge to undertake the suitable irrigation. With the SIS system, the plants can be irrigated scientifically and the irrigative resources can be saved. The iLive nodes are also used in the SIS applications. The nodes sample the sensing data of the soil moisture periodically. If the moisture value decreases to a given level, the irrigation will start automatically. In reverse, the irrigation will stop when the moisture value increases to a given level. More information about this SIS application can be referred from the article [73]. And an online project demo is also available from the website: http://edss.isima.fr/sites/smir/sis.

Smart environment explorer stick (SEES): The orientation and mobility capabilities are needed for the visual-impaired people (VIP) to walk autonomously. To achieve this objective, a new assistive device named SEES 'Smart Environment Explorer Stick' is designed and implemented in our work. More information about this project is presented in the article [74], and an online demo for this project is also available from the website: http://edss.isima.fr/sites/smir/sees.

Smart care: Smart care is a WSN project (STAR 'Système Télé-Assistance Réparti') dedicated to real-time remote continuous cardiac arrhythmias detection. With the STAR system, the health status of the patients can be cared remotely through the Internet (Figure 5.3). More information about this project can be referred from the articles [75, 76].

Wireless multimedia sensor networks: Due to the availability of inexpensive hardware such as CMOS cameras and microphones, the development of Wireless Multimedia Sensor Networks (WMSNs) has been fostered rapidly. With the WMSN, the multimedia environmental data, such as the audio streams, the images and the scalar sensor data, can be retrieved ubiquitously. Currently in our work, the WMSN technology is used for the plant diseases detection. More information about this application can be referred from the article [77]. And an online demo is available from the website: edss.isima.fr/demoforall/ (login: demo; password: demo).

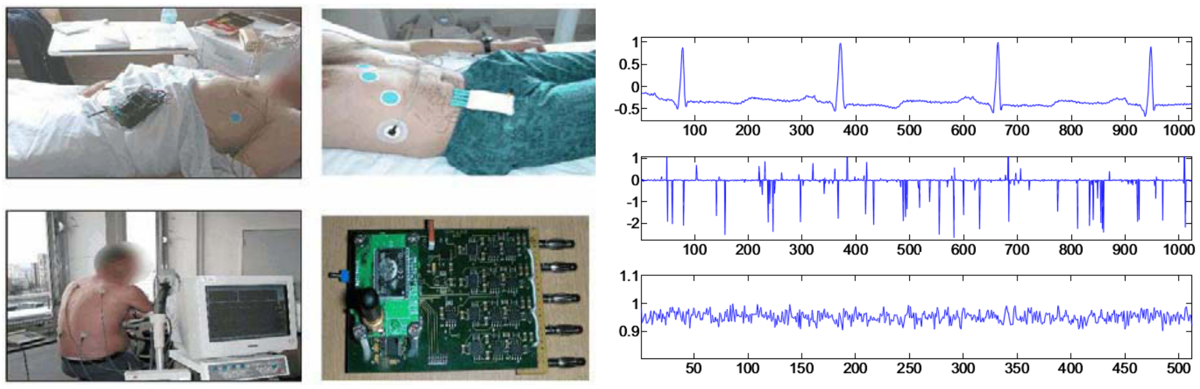


Figure 5.3: Smart care WSN project.

Chapter 6.
Conclusion and Ongoing Work

In this thesis, a real-time, memory-efficient, energy-efficient, user-friendly and fault-tolerant WSN OS MIROS is designed and implemented. MIROS achieves the real-time scheduling with less RAM cost. With this feature, the real-time WSN applications are feasible to be executed on the low-end high memory-constraint WSN nodes, e.g., after the MIROS is applied, the memory-constrained iLive nodes can be used to run the time-critical industrial engine control tasks. Besides this feature, MIROS also shows the advantages in having a long lifetime, being fault-tolerant and supporting efficient remote reprogramming. Consequently, the WSN nodes become practicable to be deployed in the harsh environments where the human-labor maintenances (e.g., the effort to bring the nodes back after the deployment for the power recharging, the application reprogramming and the fault reparation) are difficult. Furthermore, MIROS provides a user-friendly application development environment. In this manner, the WSN proliferation can be promoted to more application domains. Typically, MIROS can be applied in the contexts with the following rigorous conditions: (1) nodes are equipped with the low-end microcontrollers, and are constrained in the memory and energy resources; (2) a real-time guarantee is required by some tasks; (3) the deployment environment is harsh so that it is difficult to re-collect the nodes after they are deployed.

The ongoing work of MIROS will focus on the following topics:

Finite-state-machine MIROS: To improve the software reliability further, all the system services in MIROS will be developed by using the finite-state-machine programming style.

Every time the code related to a state is executed, the execution result will be checked. If the result is incorrect, the roll-back recovery will be performed. Only when the result is validated, can the execution transit into the next state. An execution error can thus be limited to spread widely, and the run-time failure probability can be decreased.

Establishment of the intra-communication network among the multiple cores: To simplify the management of the multi-core WSN platform, the communication among the multiple cores on the multi-core platform will be operated as the communication among the nodes in a network. For example, the communication among the cores on the multi-core node can be achieved in the networking modes: P2P (point to point), P2M (point to multi-point) and M2P (multi-point to point). Moreover, the ―channel/pipes” concept (similar to the ―socket/ports” concept in the TCP/IP protocol) will be applied in the MIROS IPC to abstract the low-level hardware communication details.

System on Chip (SoC) technology: Currently, different multi-core WSN nodes (EMWSN, MiLive, iliveT, etc.) have been developed to meet the different application contexts. This development method is complicated in the manufacture process and high in the maintenance cost. In the next generation of the multi-core hardware development, the SoC technology will be used. With the SoC, the working modes of the multi-core nodes can be reconfigured by uploading different firmware. Once reconfigured, a node can change the modes and adapt to the new application context. In so doing, one integrated multi-core node can be used to support different kinds of WSN applications. As a result, the manufacture cost can be reduced and the software development cycle can be minimized.

Parallel computing to improve the real-time performance: The multi-core technology is presently used to reduce energy consumption, to improve system reliability and to realize the new debugging approach. In future work, this technology will also be used to improve the system real-time performance. In the case that two or more hard real-time tasks are triggered simultaneously, these tasks will be distributed onto different cores and be processed concurrently. The deadlines of the real-time tasks can therefore be better guaranteed.

AADL for the OS simulation and verification: The AADL (Architecture Analysis and Design Language) is a model-based engineering language used to analyze the reliability, availability, timing, responsiveness, throughput, safety and security of the software and hardware architectures [80]. In the ongoing work, the AADL will be used in MIROS to achieve the objective of developing a formal verifiable, highly reliable, fail-detectable and self-recoverable system.

Automatic generation of the application code: With the middleware EMIDE, the WSN application code is decoupled from the system code. Thus, the WSN application development process becomes simple. As a result, it is feasible to generate the application code automatically by means of the GUI (Graphic User Interface) toolkit. With the GUI toolkit, the application programming complexity can be eased further. The WSN users select some graphic modules, connect them logically, configure the module parameters and then the prototype application programs can be generated.

Bibliography

[1] Proceedings of the Distributed Sensor Nets Workshop (1978). Pittsburgh, USA. Department of Computer Science, Carnegie Mellon University.

[2] Kumar, S. & Shepherd, D. "Sensit: Sensor information technology for the warfighter", Proc. of the 4th International Conference on Information Fusion (FUSION), pp. 3-9. 2001.

[3] IEEE 802.15 WPAN Task Group 4. http://www.ieee802.org/15/pub/TG4.html.

[4] ZigBee specification. Sponsored by ZigBee Alliance. Jan. 2008. http://www.zigbee.org.

[5] 21 Ideas for the 21st Century (1999). BusinessWeek, pp. 78-167.

[6] Y. Xu, J. Heidemann, and D. Estrin, Geography-informed energy conservation for ad hoc routing, in Proc. Mobicom, 2001, pp. 70–84.

[7] Mulligan, Geoff, "The 6LoWPAN architecture", EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors, ACM, 2007.

[8] Zach Shelby and Carsten Bormann, "6LoWPAN: The wireless embedded Internet-Part 1: Why 6LoWPAN?" EE Times, May 23, 2011.

[9] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, David Culler. "An analysis of a large scale habitat monitoring application". International conference on Embedded networked sensor systems, SenSys'04, pp. 214-226. New York, USA, 2004.

[10] Gilman Tolle, Joseph Polastre et al. "A macroscope in the redwoods". International conference on Embedded networked sensor systems, SenSys'05, pp. 51-53. New York, USA, 2005.

[11] Hakala, I. et al. "wireless sensor network in environmental monitoring-case fox house". Sensor Technologies and Applications, 2008. SENSORCOMM. Cap Esterel, France. Aug. 2008.

[12] Werner-Allen, G. et al. "Deploying a wireless sensor network on an active volcano". Internet Computing, IEEE . Vol. 10 , Issue 2. pp. 18-25. 2006.

[13] Fire Information and Rescue Equipment. Wireless Sensor Nets for Emergency Fire Responders.

[14] ALERT . http://www.alertsystems.org/.

[15] Body Sensor Networks. http://bsn-web.org/.

[16] CodeBlue: Wireless Sensor Networks for Medical Care. http://www.eecs.harvard.edu/mdw/proj/codeblue/.

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

[18] B. Zhou et al.. A wireless sensor network for pervasive medical supervision. In Proc. IEEE Int'l Conf. ICIT, pp. 740-744, 2007.

[19] O. Garcia-Morchon et al.. Efficient and context-aware access control for pervasive medical sensor networks. In 8th IEEE Int'l Conf. Workshop PERCOM, pp. 322-327, 2010.

[20] H Alemdar, C Ersoy. "Wireless sensor networks for healthcare: A survey". Computer Networks, Volume 54, Issue 15, pp. 2688–2710. October 2010.

[21] Benoît Latré et al. "A survey on wireless body area networks". Journal Wireless Networks. Volume 17 Issue 1, pp. 1-18, January 2011.

[22] M. Chen, S. Gonzalez, A. Vasilakos, H. Cao. "Body area networks: A survey". Journal Mobile Networks and Applications. Volume 16 Issue 2, pp. 171-193. April 2011.

[23] WiSA: Wireless Sensor and Actuator Networks for Measurement and Control. http://www.control.hut.fi/Research/WiSA/.

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

[25] Hou, L.; Bergmann, N.W. "A novel industrial wireless sensor network for condition monitoring and fault diagnosis of electrical machines". Australian Journal of Electrical and Electronics Engineering, v 10, n 4, p 505-514, 2013.

[26] Aghaei, Babak. "Using wireless sensor network in water, electricity and gas industry". International Conference on Electronics Computer Technology, v 2, p 14-17, 2011.

[27] A. S.-V. A. Bonivento, L.P. Carloni. Platform based design of wireless sensor networks for industrial applications. In Proceedings of Design Automation and Test in Europe (DATE), Munich, March 2006.

[28] Kouche, Ahmad El; Hassanein, Hossam; Obaia, Khaled. "Monitoring the reliability of industrial equipment using wireless sensor networks ". International Wireless Communications and Mobile Computing Conference, pp. 88-93, 2012.

[29] J. Åkerberg et al. Future research challenges in wireless sensor and actuator networks targeting industrial automation. In Proc. IEEE Int'l Conf. INDIN, pp. 410-415, 2011.

[30] I. Bekmezci et al.. "Energy efficient, delay sensitive, fault tolerant wireless sensor network for military monitoring". IEEE Sensors Applications Symposium, SAS, pp. 172-177, 2008.

[31] Lee, Sang Hyuk; Lee, Soobin; Song, Heecheol; Lee, Hwang Soo. "Wireless sensor network design for tactical military applications: Remote large-scale environments". IEEE Military Communications Conference MILCOM, 2009.

[32] Durišic, Milica Pejanovic; Tafa, Zhilbert; Dimic, Goran; Milutinovic, Veljko "A survey of military applications of wireless sensor networks". Mediterranean Conference on Embedded Computing, MECO 2012, pp. 196-199, 2012.

[33] Maciuca, Andrei; Stamatescu, Grigore; Popescu, Dan; Strutu, Mircea. "Integrating wireless body and ambient sensors into a hybrid femtocell network for home monitoring". International Conference on Systems and Computer Science, ICSCS 2013, p 32-37, 2013.

[34] Yu, Chong; Chen, Xiong. "Home monitoring system based on indoor service robot and wireless sensor network". Computers and Electrical Engineering, v 39, n 4, p 1276-1287, May 2013.

[35] Ling, Zhong; Deguo, Yang; Yan, Jiang; Haiwen, Feng "Design and realization of smart home environmental monitoring system based on wireless sensor networks". Advanced Materials Research, v 658, p 565-568, 2013.

[36] J. Zhang et al.. "Design of a wireless sensor network based monitoring system for home automation". In Proc. Int'l Conf. Future Computer Sciences and Application, ICFCSA, pp. 57-60, 2011.

[37] Mekikis, Prodromos-Vasileios; Athanasiou, George; Fischione, Carlo. "A wireless sensor network testbed for event detection in smart homes". IEEE International Conference on Distributed Computing in Sensor Systems, DCoSS 2013, p 321-322, 2013.

[38] Wei, Min; Rim, Keewook; Kim, Keecheon. "An intrusion detection scheme for home wireless sensor networks". Applied Mechanics and Materials, v 121-126, p 3799-3804, 2012.

[39] Tang, Jun; Zhang, Tingting. "Mid Sweden University: A survey of wireless sensor networks for home healthcare monitoring application". International Conference on Sensor Networks, p 240-243, 2013, SENSORNETS 2013.

[40] Suryadevara, N.K.; Mukhopadhyay, S.C.; Wang, R.; Rayudu, R.K.; Huang, Y.M. "Reliable measurement of Wireless Sensor Network data for forecasting wellness of elderly at smart home". IEEE Instrumentation and Measurement Technology Conference, pp. 16-21, 2013.

[41] J. Burrell, T. Brooke, and R. Beckwith. Vineyard computing: Sensor networks in agricultural production. IEEE Pervasive Computing, 03(1):38-45, 2004.

[42] Camalie Networks Wireless Sensing. http://camalie.com/WirelessSensing/Wireless Sensors.htm.

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

[44] T. Kalaivani et al.. A survey on Zigbee based wireless sensor networks in agriculture. In Proc. of the 3rd Int'l Conf. TISC, pp. 85-89, 2011.

[45] Y. Zhu et al.. Applications of Wireless Sensor Network in the agriculture environment monitoring. In Proc. Int'l Conf. APEE 2011., pp. 608-614, 2011.

[46] P. Patil et al.. Wireless sensor network for precision agriculture. In Proc. Int'l Conf. CICN, pp. 763-766, 2011.

[47] Yu, Xiaoqing; Wu, Pute; Han, Wenting; Zhang, Zenglin. "A survey on wireless sensor network infrastructure for agriculture". Computer Standards and Interfaces, v 35, n 1, p 59-64, January 2013.

[48] Yu, Xiaoqing; Wu, Pute; Wang, Ning; Han, Wenting; Zhang, Zenglin. "Survey on wireless sensor networks agricultural environment information monitoring". Journal of Computational Information Systems, v 8, n 19, pp. 7919-7926, October, 2012.

[49] K Sohraby, D Minoli, T Znati. "Wireless sensor networks: technology, protocols, and applications". Wiley. 2007.

[50] Jennifer Yick, Biswanath Mukherjee, Dipak Ghosal. "Wireless sensor network survey". Computer Networks. pp. 2292 – 2330. 2008.

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

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

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

[54] openWSN. https://openwsn.atlassian.net/wiki/pages/viewpage.action?pageId=688187.

[55] S. Bhatti, J. Carlson, H. Dai, J. Deng et al.. "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," ACM kluwer Mobile Networks & Applications Journal, August 2005.

[56] SimpleRTJ, a small footprint Java VM for embedded and consumer devices. RTJ Computing Pty. Ltd. Online at http://www.rtjcom.com/.

[57] J. Labrosse, MicroC/OS-II: The Real-Time Kernel, 2nd edition, CMP Books, June 2002.

[58] Q Cao, T Abdelzaher, J Stankovic. "The liteos operating system: Towards unix-like abstractions for wireless sensor networks". International Conference on Information Processing in Sensor Networks. St. Louis, MO. 2008.

[59] H Cha, S Choi, I Jung, H Kim, H Shin. "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks". 6th International Symposium on Information Processing in Sensor Networks. Cambridge, MA. April. 2007.

[60] A Eswaran, A Rowe, R Rajkumar. "Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks". 26th IEEE International Real-Time Systems Symposium. Miami, FL. Dec. 2005.

[61] JamaicaVM 6.1, User Manual: Java Technology for Critical Embedded Systems, February 2012.

[62] JamVM, a compact java virtual machine. http://jamvm. sourceforge.net/.

[63] TinyVM, an open source embedded JavaVM. http://tinyvm. sourceforge.net/

[64] N. Brouwers, K. Langendoen, and P. Corke, B. Darjeeling, a feature-rich VM for the resource poor. in Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (Sensys), pp. 169–182. 2009, NY, USA.

[65] NanoVM, Java for the AVR, http://www.harbaum.org/ till/nanovm/index.shtml.

[66] Jwik, Java programmable Wireless Kontroller. http://jwik. codeplex.com/.

[67] Sun Microsystems Virtual Machine Specification, Java Card Platform, v2.2.2, March, 2006.

[68] GNU linker. ld version 2. http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html.

[69] ELF. Executable and Linkable Format. http://www.cs.cmu.edu/afs/cs/academic/class/ 15213-f00/docs/elf.pdf.

[70] 8-bit AVR instruction set. http://www.atmel.com/images/doc0856.pdf.

[71] Atmel AVR2058: BitCloud OTAU User Guide. http://www.atmel.com/Images/doc8426.pdf.

[72] Atmel AVR2050: Atmel BitCloud Developer Guide. http://www.atmel.com/images/doc8199.pdf.

[73] Xunxing Diao, Kun Mean Hou, Hongling Shi and Zuoqin Hu. "A Sociable Wireless Smart Irrigation System (SIS) For Precision Agriculture". Workshop of New and smart Information Communication Science and Technology to support Sustainable Development (NICST), 18-20 September 2013, Clermont Ferrand, France.

[74] Muhammad Yusro, Kun Mean Hou, Edwige Pissaloux, Kalamullah Ramli, Dodi Sudiana, Zhang Lizhong and Hongling Shi. "Design and Implementation of SEE-Phone in SEES (Smart Environment Explorer Stick)". NICST'13, 18-20 September 2013, Clermont Ferrand, France.

[75] Hao Ding. "Key Concepts for Implementing SoC-Holter". PhD thesis of University Blaise Pascal. Oct. 2011.

[76] Haiying ZHOU. "wireless sensor networks dedicated to remote continuous real-time cardiac arrhythmias detection and diagnosis". PhD thesis of University Blaise Pascal.

[77] Kai Dang, Hong Sun et al. "Wireless Multimedia Sensor Network for plant disease detections". NICST'13, 18-20 September 2013, Clermont Ferrand, France.

[78] S. Brown et al. "Updating Software in Wireless Sensor Networks: A Survey". Technical Report UCC-CS-2006-13-07. 2006.

[79] Stephen Brown and Cormac J. Sreenan. "Software Updating in Wireless Sensor Networks:A Survey and Lacunae ". Journal of Sensor and Actuator Networks. 2013.

[80] AADL-Architecture Analysis and Design Language. http://www.aadl.info/aadl/currentsite/.

[81] F. Stajano. Security for Ubiquitous Computing. Wiley, 2002.

[82] David Gay, Philip Levis, David Culler, Eric Brewer. "nesC 1.1 Language Reference Manual". May 2003.

[83] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, ―Run-time dynamic linking for reprogramming wireless sensor networks,” in Proceedings of ACM SenSys, 2006.

[84] J. Hellerstein et al.. Events and threads. Lecturer Notes, November 2005.

[85] J. K. Ousterhout. ”Why Threads Are A Bad Idea (for most purposes)”, Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[86] R. von Behren Jeremy Condit et al.. Why events are a bad idea (for high-concurrency servers). In 10th Workshop on Hot Topics in Operating Systems (HotOS IX), May 2003.

[87] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, "A Comprehensive Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems", Journal of Networks, Vol. 3, No. 3, March 2008.

[88] Li, S.-F., Sutton, R., and Rabaey, J.: Low Power Operating System for Heterogeneous Wireless Communication Systems. In Proceedings of the Workshop on Compilers and Operating Systems for Low Power, Barcelona, Spain, 2001.

[89] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, ―Protothreads: simplifying event-driven programming of memory-constrained embedded systems,” in Proceedings of ACM SenSys, 2006.

[90] C.L. Liu, J.W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973, pp. 46-61.

[91] Evans, J. "A scalable concurrent malloc(3) implementation for FreeBSD". Proceedings of BSDCan, 2006.

[92] Masmano, M.; Ripoll, I.; Crespo, A.; Real, J.. "TLSF: a new dynamic memory allocator for real-time systems". In Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS '04). 2004. Washington, DC, USA.

[93] Kenneth C. Knowlton. "A Fast storage allocator". Communications of the ACM 8(10):623-625, Oct 1965.

[94] Knuth, D. E.. "Fundamental Algorithms: The art of computer programming". MA, USA. Addison-Wesley. 1973.

[95] Comfort, W. T. (1964). Multiword list items. Communications of the ACM, 7(6), 357-362.

[96] TinyOS TYMO routing protocol. http://tinyos.stanford.edu/tinyos-wiki/index.php/Tymo.

[97] Rime-A Lightweight Layered Communication Stack for Sensor Networks. Download from: http://dunkels.com/adam/dunkels07rime.pdf. Accessed on Jan. 2014.

[98] Winter, T.; Thubert, P. RPL: Ipv6 Routing Protocol for Low Power and Lossy Networks, Draft-ietf-roll-rpl-11; Available online: http://tools.ietf.org/html/draft-ietf-roll-rpl-06. Accessed on Jan. 2014.

[99] Tsiftes, N.; Eriksson, J.; Dunkels, A. Low-Power Wireless Ipv6 Routing with ContikiRPL. In Proceedings of ACM/IEEE IPSN, Stockholm, Sweden, 12–16 April 2010.

[100] T Watteyne, X Vilajosana, B Kerkez et al. "OpenWSN: a standards‐based low‐power wireless development environment". Transactions on Emerging Telecommunications Technologies. Volume 23, Issue 5, pages 480–493, August 2012.

[101] Shelby Z, Hartke K, Bormann C, Frank B. Constrained Application Protocol (CoAP) 12 March 2012.

[102] IEEE 802.15.4e Time Synchronized Channel Hopping. http://tools.ietf.org/html/ draft-ietf-6tisch-tsch-00.

[103] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In Proc. ASPLOS-X, San Jose, CA, USA, Oct. 2002.

[104] Q. Wang, Y. Zhu, and L. Cheng, ―Reprogramming wireless sensor networks: Challenges and approaches," IEEE Network Magazine, vol. 20, no. 3, pp. 48–55, 2006.

[105] Reijers, N., Langendoen, K.: ―Efficient Code Distribution in Wireless Sensor Networks". In: Proc. of the Second ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA'03), San Diego, CA, Sept. 2003. ACM (2003) 60-67

[106] Rajesh K. Panta, Saurabh Bagchi. "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks". IEEE INFOCOM. April 2009.

[107] Rajesh Krishna Panta, Saurabh Bagchi, Samuel P. Midkiff. "Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation". Proceedings of the 2009 conference on USENIX Annual technical conference. USENIX Association Berkeley, CA, USA.

[108] Jaein Jeong; Culler, D. "Incremental Network Programming for Network Sensors", In: Proc. of the 1st Annual IEEE Communications Society Conf. on Sensor and Ad Hoc Networks and Communications (SECON 2004). IEEE(2004) 25-33.

[109] Koshy, J., Pandy, R.: ―Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks". In: Proc. of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05). IEEE (2005). 354-365.

[110] Jaein Jeong, Sukun Kim and Alan Broad. "Network Reprogramming". Aug 12, 2003.

[111] Sandeep S. Kulkarni, LiminWang, "MNP: Multihop Network Reprogramming Service for Sensor Networks". Distributed Computing Systems. Columbus, OH. 2005.

[112] Philip Levis, Neil Patel, David Culler, and Scott Shenker. "Trickle: A Self Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks". First Symposium on Networked Systems Design and Implementation (NSDI '04), March 29–31 in San Francisco, CA.

[113] Jonathan W. Hui, David Culler. "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale". 2nd international conference on Embedded networked sensor systems. 2004.

[114] T Stathopoulos, J Heidemann, D Estrin. "A Remote Code Update Mechanism for Wireless Sensor Networks".

[115] V. Raghunathan, C. Schurghers, S. Park, M. Srivastava, ―Energy-aware Wireless Microsensor Networks", IEEE Signal Processing Magazine, March 2002, pp. 40-50.

[116] G. Pottie, W. Kaiser, ―Wireless Integrated Network Sensors, Communication of ACM, Vol. 43, N. 5, pp. 51-58, May 2000.

[117] D. Ganesan, A. Cerpa, W. Ye, Y. Yu, J. Zhao, D. Estrin, ―Networking Issues in Wireless Sensor Networks", Journal of Parallel and Distributed Computing, Vol. 64 (2004) , pp. 799-814.

[118] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler and J. Anderson, ―Wireless Sensor Networks for Habitat Monitoring", Proc. ACM Workshop on Wireless Sensor Networks and Applications, pp. 88-97, Atlanta (USA), September 2002.

[119] A. Warrier, S.Park J. Mina and I. Rheea, ―How much energy saving does topology control offer for wireless sensor networks? – A practical study", Elsevier/ACM Computer Communications, Vol. 30 (14-15), pp. 2867-2879, 15 October 2007.

[120] K. Arisha, M. Youssef, M. Younis ―Energy-aware TDMA-based MAC for Sensor Networks", Proc. IEEE Workshop on Integrated Management of Power Aware Communications, Computing and Networking (IMPACCT 2002), New York City (USA), May 2002.

[121] J. Li, G. Lazarou, ―A Bit-map-assisted energy-efficient MAC Scheme for Wireless Sensor Networks", Proc. International Symposium on Information Processing in Sensor Networks (IPSN 2004), pp. 56-60, Berkeley (USA), April 2004.

[122] V. Rajendran, K. Obracza, J. J. Garcia-Luna Aceves, ―Energy-efficient, Collision-free Medium Access Control for Wireless Sensor Networks”, Proc. ACM SenSys 2003, Los Angeles (USA), November 2003.

[123] D. Chu, A. Deshpande, J.M. Hellerstein, W. Hong, ―Approximate Data Collection in Sensor Networks using Probabilistic Models”, Proc. of the 22nd International Conference on Data Engineering (ICDE06), p. 48, Atlanta, GA, April 3-8, 2006.

[124] A. Jain, E. Y. Chang, Y.-F. Wang, ―Adaptive Stream Resource Management Using Kalman Filters”, Proc. of the ACM International conference on Management of Data (SIGMOD2004), Paris (France), pp. 11-22, June 13-18, 2004.

[125] B. Kanagal and A. Deshpande, ―Online Filtering, Smoothing and Probabilistic Modeling of Streaming Data”, Proc. of the 24th International Conference on Data Engineering (ICDE 2008), Cancún, México, April 7-12, 2008.

[126] Q. Han, S. Mehrotra, N. Venkatasubramanian, ―Energy Efficient Data Collection in Distributed Sensor Environments”, Proc. of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04), pp. 590-597, March, 2004.

[127] D. Tulone, S. Madden, ―PAQ: Time series forecasting for approximate query answering in sensor networks”, Proc. of the 3rd European Conference on Wireless Sensor Networks (EWSN06), pp. 21-37, February 2006.

[128] D. Tulone, S. Madden, ―An energy-efficient querying framework in sensor networks for detecting node similarities”, Proc. of the 9th Intl. ACM Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM06), pp. 291-300, October 2006.

[129] C. Tang, C. S. Raghavendra, ―Compression techniques for wireless sensor networks”, Chapter 10 in book Wireless Sensor Networks, pp. 207-231, Kluwer Academic Publishers, 2004.

[130] M. Wu, C. W. Chen, ―Multiple Bit Stream Image Transmission over Wireless Sensor Networks”, Chapter 13 in book Sensor Network Operations, pp. 677-687, IEEE & Wiley Interscience, 2006.

[131] Z. Xiong; A. D. Liveris, S. Cheng, ―Distributed source coding for sensor networks”, IEEE Signal Processing Magazine, Vol. 21 (5), pp. 80-94, Sept. 2004.

[132] K. Klues, C.-J. M. Liang, J. yeup Paek, et al., ―TOSThreads: Safe and Non-Invasive Preemption in TinyOS,” in Proceedings of ACM SenSys, 2009.

[133] P. Levis, N. Lee, A. Woo, S. Madden, and D. Culler. Tossim: Simulating large wireless sensor networks of tinyos motes. Technical Report UCB/CSD-TBD, U.C. Berkeley Computer Science Division, March 2003.

[134] Fredrik Österlind. "A Sensor Network Simulator for the Contiki OS", SICS Technical Report, 2006.

[135] Hector Abrach, Shah Bhatti et al.. "Mantis-system supports for multimodAl neTworks on in-situ sensors". Conference On Embedded Networked Sensor Systems (SenSys), pp. 336-337, California, USA, 2003.

[136] Adam Torgerson. "Automatic thread stack management for resource-constrained sensor operating systems." http://pdf.aminer.org/000/310/128/automatic_management_of_operating_system_resources.pdf.

[137] Locke, J. 1997. Designing real-time systems. In IEEE International Conference of Real-Time Computing Systems and Applications (RTCSA ’97), Taiwan (Invited talk).

[138] Lehoczky, J. P., Sha, L., and Strosnider, J. K. 1987. Enhanced aperiodic responsiveness in hard real-time environments. In Proceedings of the IEEE Real-Time Systems Symposium, pp. 261–270.

[139] Sprunt, B., Sha, L., and Lehoczky, J. 1989. Aperiodic task scheduling for hard real-time system. Journal of Real-Time Systems 1: 27–60.

[140] Hai-ying Zhou; Feng Wu; Kun-mean Hou. "An Event-driven Multi-threading Real-time Operating System dedicated to Wireless Sensor Networks". International Conference on Embedded Software and Systems. ICESS '08. Sichuan, China. 2008.

[141] Atmel AVR2052: Atmel BitCloud Quick Start Guide. 2012. http://www.atmel.com/Images/doc8200.pdf.

[142] Atmel AVR2025: IEEE 802.15.4 MAC Software Package -User Guide. 2012. http://www.atmel.com/Images/doc8412.pdf.

[143] Hongling Shi. "Development of an Energy Efficient, Robust and Modular Multicore Wireless Sensor Network". PhD thesis of University Blaise Pascal. January 2014.

[144] ATmega1281. Atmel-Corporation. (2012b). 8-bit Microcontroller with 64K/128K/256K Bytes In-system programmable Flash 2549O–AVR–05/12 (pp. 1-448).

[145] AT91SAM7S. Atmel-Corporation. (2011). AT91SMA ARM-based Flash MCU6175L-ATARM-2-8-Jul-11 (pp. 1-781).

[146] IGLOO. Actel-Corporation. IGLOO nano Low-Power Flash FPGAs with Flash*Freeze Technology (pp. 1-120). 2009.

[147] Decagon Sensor Introduction, http://www.decagon.com/products/sensors/soil-moisture-sensors/.

[148] Watermark Sensor Introduction, http://www.irrometer.com/sensors.html.

[149] AVR studio tool. http://www.atmel.com/tools/atmelstudio.aspx.

[150] Raspberry Pi Foundation. (2013). Raspberry Pi. 2013, from http://www.raspberrypi.org/.

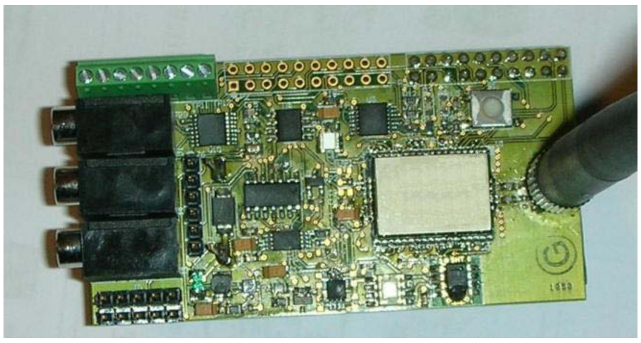
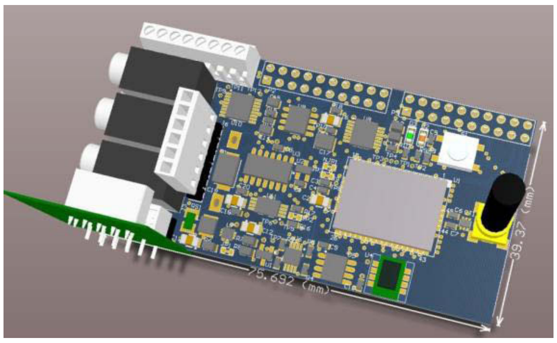
[151] Willson, P.R.; Johnstone, M.S.; Neely, M.; Boles, D. Dynamic storage allocation: A survey and critical review. In Proceedings of International Workshop on Memory Management, Kinross, UK, 27–29 September 1995.

[152] Hou, K.M.; Sousa, G.D.; Chanet, J.P.; Zhou, H.Y. LiveNode: LIMOS versatile embedded wireless sensor node. J. Harbin Inst. Technol. 2007, 32, 139–144.

Appendix

In this appendix, some WSN nodes designed and implemented by the SMIR team, LIMOS laboratory, CNRS UMR 6158, FRANCE will be listed.

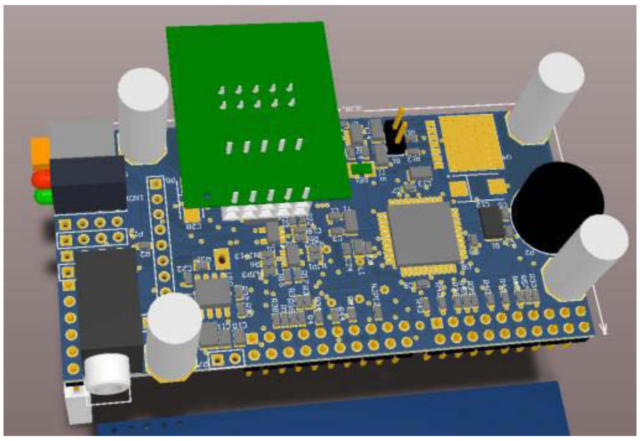
■ iLive node:

Features:

- Ultra Low energy consumption;
- Two AA standard batteries for 5 years (1 sample per day);
- Dimension 76mm*40mm
- Four Watermark sensors
- Three Decagon sensors
- One temperature sensor
- One air humidity sensor
- One light sensor
- One RS232/USB slave port
- One IEEE802.15.4 ZigBee wireless access medium.

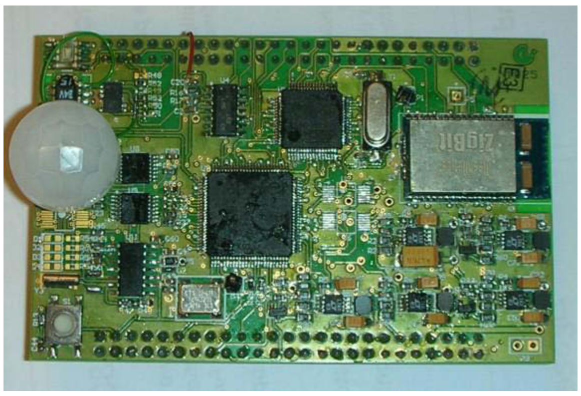
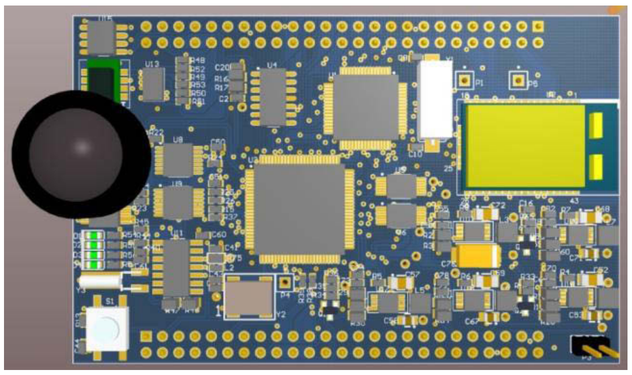
■ Smart Irrigation System (SIS) node:



Features:

- Low Energy Consumption (1 year Lifetime)
- Reliability and Safety
- 9-Volt Alkaline Battery
- Dimension 79mm*40mm
- One DC-9V Electro-valve to control flow of irrigate water
- One Buzzer
- One Watermark sensors
- One Decagon sensors
- One soil temperature sensor
- One RS232/USB slave port
- Optional (one air temperature sensor, one air humidity sensor, one light sensor, one IEEE802.15.4 ZigBee wireless accessmedium)

■ Multi-core E2MWSN node:

Features:

- Ultra low power consumption
- Dimension 85mm*54mm
- One light sensor
- One temperature sensor
- One air humidity sensor
- One Passive Infrared Motion Detector
- Three Decagon soil moisture sensors
- Two RS232 ports
- One USB slave port
- One RTC
- SD Card
- IEEE802.15.4 ZigBee wireless access medium

■ Multi-core Edge Router:

Features:

- Low Cost
- Higher Performance (ARM11 700 MHz ARM1176JZF-S core, 256M SDRAM)
- Dimension 86mm*54mm
- 2x USB 2.0 HS Host ports
- 10/100 Ethernet Port
- Composite RCA (PAL & NTSC), HDMI, LCD Panels via DSI
- SD / MMC / SDIO card slot