Design and Implementation of a Dynamic Module and Self-Maintaining Mecahnism in Wireless Sensor Network Operating System

Li-Chi Feng, Sung-Nan Yao, Yu-Yao Wang, Bing-Chuen Fang
Department of Computer Science and Information Engineering
Chang Gung University
lcfeng@mail.cgu.edu.tw

Abstract

The purpose of sensor node is to detect the around environment, and to process the gathered data or deliver the data to data-gathering machine through sensor network.

In many applications, sensor nodes are distributed over wide environment. It is very difficult to retrieve them for maintenance or software update when the application of sensor nodes change or system occur unusual behavior. How to make sensor node have the self-maintenance capability is becoming an important issues.

Besides, the ability of sensor network OS at present is too simple to deal with new sensor network application in the future.

In this paper, we design and implement the first sensor network operating system which has self-maintaining ability. Our system has the following features: light-weight dynamic module mechanism, fault detection, self-recovery and self-update ability. The experiment results show that our system can operate correctly and efficiently.

Keywords: Sensor Network Operating Systems, Dynamic Module, Recovery, Self-maintaining, Software Update

1. Introduction

The purpose of sensor node is to detect the around environment, and to process the gathered data or deliver the data to data-gathering machine through sensor network that is composed of sensor nodes.

The application of sensor network is very wide. It can apply to house, industry, medical-treatment and some other relevant environments monitoring and control.

For example, we can install the sensor in our building to monitor the emergence of the fire. The system will notify relevant units or trigger the alarm when the fire takes place. In medical application, the sensor can be used to monitor the body temperature or health status of patients. If necessary, the system will notify the nearest medical personnel to help.

The number of sensor node in a sensor network application may be huge. It can range between several hundreds to several ten thousands units and will be distributed over wide environment. It is very difficult to retrieve these nodes for maintenance when the application change or system occur unusual behavior. It is becoming an important issues that how to make these sensor nodes have the self-maintaining ability.

In this paper, we design and implement the first sensor network operating system which has self-maintaining ability. Our system has the following features: light-weight dynamic module mechanism, fault detection, self-recovery and self-update ability. These features make our system highly available and can provide a reliable, energy- efficient sensor network application environment. The empirical results show that our system can operate correctly and efficiently, the performance has not been influenced by new-added self-maintaining mechanism.

The paper is organized as follow: Section 2 is related work. Our system design is briefly stated in section 3. Section 4 describes the design of our module format and dynamic module mechanism. The content of section 5 is experiment. Section 6 is system evaluation. This paper is concluded in Section 7.

2. Related Work

TinyOS[2] is an event-driver system designed by UC Berkeley. Virtual machine, Maté[3], is regarded as uppermost component of TinyOS. It is treated as a monitor taking care of software activities above OS and performing dynamic loading/updating of software module. However, the biggest shortcoming lies in its low efficiency, and Maté can not change the kernel components.

SOS[7] sensor network operating system that Chih-Chieh Han et al. proposed in 2005 can swap software while system running. But without domain protection mechanisml, SOS could not prevent kernel from being malicious damage. Consequently, sensor node might crash.

3. System Design Goals

In order to meet the future trend of sensor network development, we think that a well designed sensor network operating systems should have several key functionality as described below.

First, the system must support compact dynamic module for the effective wireless transmission when the system need a newer or correct software component for fault recovery or normal software upgrade.

Second, the system must have the ability to detect at least some system fault and to trigger appropriate recovery procedure. For example, if it is a kind of transient failure, we only need to re-initialize the failed software component (self-recovery). If it is a real bug, system upgrade is necessary (software update).

Third, we also need a well design module management system. Once the system fault is detected, the self-recovery or self-update procedure is started. To keep continuing running we should not stop the running system as best as we can, that means update have to be done while running.

We summary the design goals of our system briefly as following:

- Light-weight dynamic module mechanism
- Fault detection ability
- Self-recovery and self-update ability.
- Acceptable system performance

Fault detection ability can prevent our system from invalid access or external faulty module destruction. In fault detection, we seclude each module into a separate memory region. Each module can access its own memory region only, so called domain. By means of ARM 7 MMU hardware support, the domain solution is feasible. Only the module that owns this domain could write/read the corresponding memory region. Operating system kernel is the unique module that can read/write all domains' memory region.

4. Dynamic Module Mechanism

Linux is a good operating system, but it is huge for most popular sensor node architecture.

eCos(embedded Configurable operating system)[10] is an open source, full function embedded operating system which support many different hardware platform. Due to its clear structure and elegant design, more and more researchers engage into its development.

In original eCos, only source-level component is supported. At run time, the entire system is compiled and linked as static image that can be downloaded into embedded platform. That is no way to dynamically change any module or device driver.

Due to the elegant and compact design of eCos, we want to use it as the starting point of our sensor network operating system design.

So we must develop a dynamic module format and mechanism on eCos.

4.1. The Format of Module Execution File (LW-ELF)

ELF is the most popular object file format in UNIX world. In order to replace module in sensor node by dynamic loading/unloading, the object file must use the wireless device to transfer. The size of module is closely proportioned to the transmission time and power consumption. In order to down-sizing the ELF object file format, we develop an ELF-like object file format named LW-ELF.

We concentrate on reduction about section, relocation and symbol data to generate LW-ELF.

4.1.1. Reduce Section Data

Section data includes section header table and section string table. Section header table could find section string table, symbol table, symbol string table and relocation table in object code.

To trim section data, we regard sections in object code as the same block to reduce the use of section header entry and section string table. As Figure 1 shows, we put text section、BSS section and section that will be used into the Object Code block of LW-ELF.

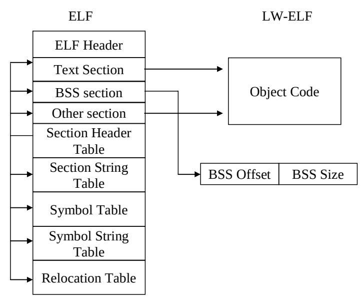

Figure 1. The method to trim section data.

4.1.2. Reduce Relocation Data

This kind of data can divide into four kinds of type. They are defined PC, ABS and undefined PC, ABS. Defined PC and ABS indicate that these machine code use the data in object code are able to find target address. However, undefined PC and ABS indicate that these machine code use the data unable to find in object code, must obtain target address through linker of OS.

The trimming methods used by LW_ELF are as follow.

- Defined PC

Operand of general branch machine code is the offset of target section. These sections based offset need relocation entry and symbol entry to obtain object code based offset.

To this kind of data, we calculate object code based offset and update branch machine code to reduce the time to relocate and the number of relocation entry and symbol.

- Defined ABS

The address of defined ABS stores the offset of target section. While loading object code into operating system, OS must calculate the absolute memory address according to the data of relocation. We calculate the object code based offset and record it in ABS defined relocate entry to save the size and time of symbol table.

- Undefined PC

Undefined PC means that the branch target is an undefined symbol must depend on OS to find the exact address of this symbol by symbol name. This kind of data is usually a memory address of a function; we will call it undefined function as follows.

In order to reduce the space accounted for of symbol, we replace symbol string with ID. As Figure 2 shows, A want to branch to B that is in operating system, we use Fun Undefined Relocate Entry to record the relocation data of A, use Fun Undefined ID Entry to record the data of B.

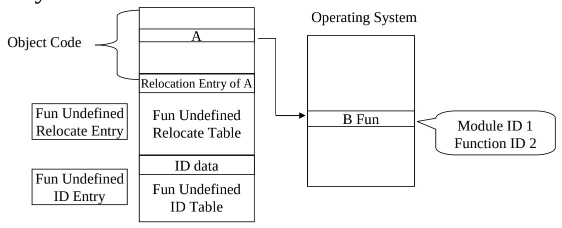

Figure 2. The method to trim undefined function

The structure of Fun Undefined Relocate Entry records the object code based offset of A and the index to Fun Undefined ID Table.

Fun Undefined ID Entry records the Fun ID and Module ID of this symbol.

- Undefined ABS

Undefined ABS is just like undefined function, we use GV Undefined Relocate Entry and GV Undefined ID Entry to record the data needed by undefined ABS.

- Module-exported Function

Module provides function for operating system such as the initial function of module must let operating system know. We use Fun Reg. Entry to offer this kind of information.

- Module-exported GV(Global Variable)

Module provides global variables for operating system; we use GV Reg Entry to offer this kind of information.

Final LW-ELF format is illustrated in Figure as below.

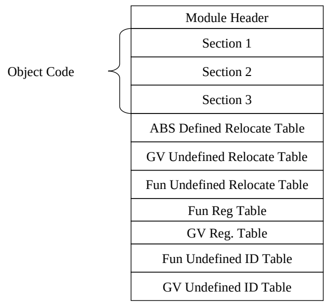

Figure 3. LW-ELF format

4.2. Automatic Tool of Format Transformation

We implement two tools, the one transforms ELF into LW-ELF, the other reads LW-ELF to obtain the information in it.

Read ELF file in buffer at beginning, and analyze the data in Header to obtain all information. After that, read Fun ID Table and GV ID Table of the operating system, trim section, relocation and symbol data, and transform relocation data into our designed format to produce LW-ELF format finally.

The main purpose of handling flow to read is to inspect if producing LW-ELF fit our demand or not. This tool will read LW-ELF in buffer, resolve LW-ELF header to find other information. At the end, it will print the data of LW-ELF Header, Object Code, Relocation, Fun Reg and GV Reg.

4.3. The Design of Module Manager

In our system, module can be device driver or AP, and will load into our system through Module Manager. Figure 4 illustrate our system infrastructure.

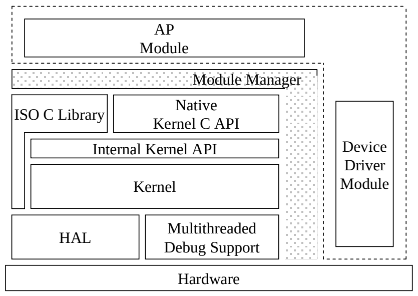

Figure 4. System infrastructure

4.3.1. Module Run-time Structure

The information of module will be recorded by module structure as a table (show as Figure 5). In order to find module structure fast, the way to store them is array and the search for them is through Module ID. Module ID 0 initialized specifically for kernel, the other IDs are assigned to other modules by developer.

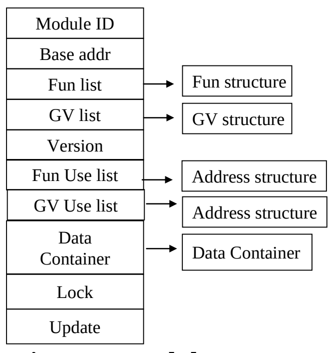

Figure 5. Module structure

The attribute of *Data_container* structure is a important area to store the configuration data , which is critical to this module. While self-maintenance or self-recovery mechanism occurs, this data structure will be used to find the saved data kept before. Data Container is the pointer to point *Data_container* structure.

There is an additional Lock attribute as a synchronous mechanism flag that will be used while self-maintenance or self-recovery mechanism occurs. Update tells kernel that module have updated or recovered. The update mechanism will be explained in section 3.4.

5. Self-maintaining Mechanism

The self-maintenance mechanism in our design is divided into two parts: self-recover and self-update.

The time to trigger module self-recover is when the system find out some failures. The main method is to re-initialize this fault module to get back to the normal working state fast. Some study have showed that this method can usually solve transient failures [12] fast.

The time to trigger self-update mechanism is self-recover mechanism can not resolve failure or module manager thread receive the new version module. Figure 6 is the self-update flow. Module manager thread will inspect whether the system receive new version module or not every ten minutes. When receiving new version module, the module manager thread will decide whether it can update or not. If the version of received module is newer than that of system, it will enter update step.

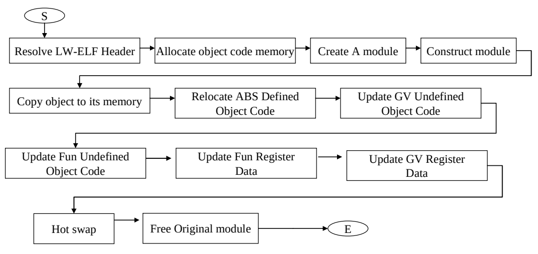
S
Resolve LW-ELF Header
Allocate object code memory
Create A module
Construct module
Copy object to its memory
Relocate ABS Defined Object Code
Update GV Undefined Object Code
Update Fun Undefined Object Code
Update Fun Register Data
Update GV Register Data
Hot swap
Free Original module
E

Figure 6. The self-update flow

The key question of this part is how to get back to the original execution environment after module update or recover. Object code of module is possible updated or recover at any time. So, the data the module use can not put into object code. It must store in *Data_Container* we design and implement.

The most important part of self-update and self-recover mechanism is hot swap. The environment of old module is transplanted into new module in the operation of the system, and replaced old module with new module. This step must deal with synchronization, will use lock in module structure to control only one can use this module.

When module thread should change old, however, old Module has already been used by kernel or other module, it will not be updated unless kernel or other module release lock of this module; when module is update, kernel or other module must wait module thread to release the lock of this module.
After finishing hot swap, module thread will sets update field in module structure as 1 to tell kernel while use this module, it must execute the reload function to read the data in *Data_container* and complete whole flow of update. Finally, module thread will free the memory space used by old module structure.

6. System Evaluation

First, we will explain how we experiment on SCAN II. After this, we will describe our experiment and the result.

6.1. Experiment Method

The implementation of Zigbee/802.15.4 protocol stack on SCAN II is a project of our laboratory. Due to the delay of protocol stack implementation, we can not use wireless device to test module transmission. We suppose that one original and one newer LW-ELF files have already existed in the memory of SCAN II, and module manager know the memory address of them.

We connect Multi-ICE to SCAN II, use ADS to download our OS image and serial port driver module. Then the OS image is executed.

When system initialize, module manager will read serial port driver module, the format of that is LW-ELF, add this module into system, and create module thread responsible for module update. Finally, it will create the thread of simple shell to communicate with user.

6.2. The Design and Result of Experiment

To verify that our method can reduce the size of ELF, we compare the size of LW-ELF and ELF first. After this, we will describe the experiment of self-recovery and self-update mechanism. Finally, we will compare our system with other sensor network OS.

6.2.1. The Comparison Between the Size of LW-ELF and ELF

The goal to trim ELF is to reduce the size of module to lower the power during transmission. We compare the size of serial port driver module that is ELF with LW-ELF. The result is show in table 1. According to table 1, the original ELF serial port driver module is 5.3K; the size of LW-ELF transformed from our tools 2.4K. The size of ELF has been reduced to 45%. Besides, simple shell can be reduced to originally 47%. It shows that LW-ELF is effective to reduce the size of ELF and help to low the power of wireless device.

Table 1. The comparison of size of ELF with LW-ELF

Application	ELF	LW-ELF
Serial Port Driver	5.3K	2.4K
Simple Shell	2.3K	1.1K

6.2.2. Self-maintaining mechanism

We try to experiment our self-maintaining mechanism by means of infusing the tested module with NULL memory access purposely. This type of fault scenario is occurred when request memory from kernel while running out of memory, the memory address returned might be NULL. In our system, NULL memory-address lead off the interrupt vector table. What if access is done, system will be under great risk of crash. In our experiment, the module is serial driver and there is an application called small shell we designed supported by serial driver. Small shell is a virtual terminal device catch user input line(s) then show on screen.

System will detect serial driver while running the code section that access NULL memory-address, then trigger Recovery Manager Thread re-initializing the serial driver module. In the period of time, kernel might gather enough memory then resume working properly. If the module access NULL memory-address intentionally, Recovery Manager Thread will try to update the module with another version stored on specific memory section. What if the newer could not be found, Recovery Manager Thread will kill small shell necessarily to prevent influencing system. During the re-initialization or update procedure, small shell catch user input then output on the screen still. That is the application supported by abnormal module will keep working normally even the module is re-initialized or updated ever.

About overhead, the average time cost during re-initialization is 10 ms and 20 ms during update.

6.2.3. Performance of Dynamic Module Manager

The testing application is an x modem application we wrote, used to transmit data, and the uses serial port driver as lower level support. The host end used x modem application to transmit data of the size is 200K through serial port to sensor node. Sensor node will execute x modem application to receive data.

The contrasting group is an original eCos without module mechanism. The experimenting group is the one with module mechanism.

Figure 7 is the average result of ten times testing. According to the result, the contrasting group will complete receiving in 40.02 seconds; the experimenting group still complete that in the same time and the receiving data is correct. This is because the design of self-recovery mechanism is very simple and makes few overhead of system.

Another statistics measured is for tty module. Our testing method is to input a piece of 200K-size-of data and take down the time cost. Finally, constructing files (256 bytes for each) up to 400 K totally, we have the time action cost. Figure 8 shows the statistics of experiments above, composite with serial driver module.

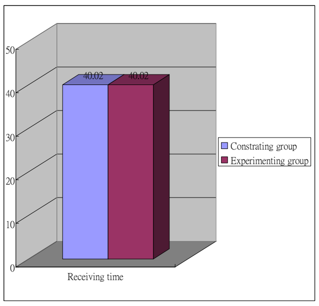

Figure 7.Comparsion of X-Modem transmission time measurement

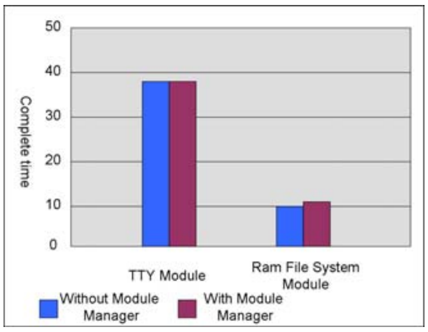

Figure 8. Execution time comparison between before and after dynamic module mechanism.

6.2.4. Comparison with Other Sensor Network OS

Due to unable to find common platform to compare each sensor network OS, we compare with each of them using static ability.

Table 2. Comparison our system with other sensor network OS

	Our System	SOS	TinyOS
Infrastructure	Multi-Threading	Event-Driven	Event-Driven
Software update mechanism	Module	Module	Script Language (Mate)
Replace system component	Yes	Yes	No
The invoking relationship between module and kernel	Direct	Indirect	None
Self-recovery ability	Yes	No	No

Table 2 compare the difference between our system and other sensor network OS, SOS and TinyOS. SOS and TinyOS are event-driven system, there is a deficiency to apply in the future. The virtual machine mechanism of TinyOS has shortcoming to exchange the component of system; however, our system is able to exchange the components of system. Though SOS can exchange the components of system, the invocation between module and kernel is indirect, making some overhead of system; the invocation of our system is direct, the performance of ours is better than indirect invocation. At last, our system has self-recovery ability, SOS and TinyOS lack that.

7. Conclusion and Future Work

Recently, more and more people are engaged into the study and development of wireless sensor network. The number of sensor node in a sensor network application may be huge. These huge number of nodes may be distributed over wide environment. It is very difficult to retrieve these nodes for maintenance when the application change or system occur unusual behavior.

This paper focus on this problem, we design and implement the first sensor network operating system with self-maintaining ability. Our system is highly available and can provide a reliable, energy-efficient platform for various sensor network applications. The empirical results show that our system can operate correctly and efficiently.

In the future, we will finish the ZigBee/802.15.4 protocol stack implementation and integrate it with our system. Furthermore, to make our sensor network OS more perfect, we will do research on low power mechanism of wireless protocol and sensor network OS.

8. Reference

[1] Berkeley Sensor node. http://www.tinyos.net/scoop/special/hardware

[2] Berkeley TinyOS. http://www.tinyos.net

[3] Phil Levis and David Culler, "Maté : a Virtual Machine for Tiny Networked Sensors " **,** *ASPLOS*, Dec 2002.

[4] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors", *In Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Tampa, Florida, USA, November 2004.

[5] A. Boulis, C.C. Han, and M. B. Srivastava, " Design and Implementation of a Framework for Programmable and Efficient Sensor Networks" , *MobiSys* 2003.

[6] Jaein Jong, David Culler, "Incremental Network Programming for Wireless Sensors", *IEEE SECON 2004* (Oct 2004).

[7] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler and Mani Srivastava, **"**A dynamic operating system for sensor networks" *,NESL Tech Report TR-UCLA-NESL-200502-01*, 2005.

[8] Linux Loadable Kernel Module HOWTO. http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Module-HOWTO.pdf

[9] ELF. http://www.x86.org/ftp/manuals/tools/elf.pdf

[10] eCos. http://sources.redhat.com/ecos/

[11] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, and Robert W. Wisniewski, Providing Dynamic Update in an Operating System", *USENIX 2005* , pp. 279-291, Anaheim California April 2005

[12] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, Henry M. Levy. *Recovering Device Drivers* , in *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[13] GCC, http://gcc.gnu.org/

[14] ARM, http://www.arm.com/

[15] Hynix HMS30C7202, http://www.magnachip.com/ENG/Products/MCU/32Bit/HMS30C7202_ref.html

[16] Eduardo Souto, Germano Guimaraes, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz "A message-oriented middleware for sensor networks," *2nd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, October 18th - 22nd, 2004, Toronto, Ontario, Canada.