
The RETOS Tutorial

Resilient, Extensible, and Threaded Operating System
for Wireless Sensor Network

Version 1.1.X / June 15th, 2007



Mobile Embedded System Lab
Department of Computer Science, Yonsei University, Korea



Contents

Chapter One	Setting Up the RETOS Development Environment for MSP430	4
	1.1 Installing Basic Development Environment	4
	1.2 USB Driver Installation & Configuration	6
	1.3 RETOS Installation	8
Chapter Two	Running the RETOS Kernel and Application Programs	10
	2.1 RETOS Kernel Installation	10
	2.2 Compiling and Installing Application Programs	12
	2.3 Example of Injecting an Application Program	15
	2.4 Kernel Module Installation	17
Chapter Three	Writing RETOS Application Programs	20
Chapter Four	Serial Communications	22
	4.1 Mote Side	23
	4.2 Host PC Side	24
	4.3 Examples	25
Chapter Five	Radio Communications	28
	5.1 Interface	29
	5.2 NSL: The Network Support Layer	32
	5.3 Examples	35
Chapter Six	Threading	40
	6.1 Interface	40
	6.2 Examples	43
Chapter Seven	Thread Scheduling	44

Chapter Eight	Sensing	48
	8.1 Interface	48
	8.2 Examples	49
Chapter Nine	Miscellaneous System Calls	52
Chapter Ten	Detecting Application Errors	54
	10.1 Static Error Detection	54
	10.2 Dynamic Error Detection	56
	10.3 Restrictions	58
Chapter Eleven	Module Programming	60
	11.1 Overview	60
	11.2 Controlling Application Context	60
	11.3 Managing Function Table	61
	11.4 Managing System Events	62
Appendix		65
	Appendix A System Calls	66
	Appendix B Module Calls	76
	Appendix C The PNS Module	84
	Appendix D RMTTool: The RETOS Monitoring Tool	86
	Appendix E ETA (Elapsed Time Arrival) Field	96
	Appendix F Code Dissemination	98
	Appendix G Hardware Supports	100
	Appendix H RETOS Paper List	101

1

Setting Up the RETOS Development Environment for MSP430

You should establish a cross-compile environment for developing sensor applications using RETOS for sensor node devices based on micro controllers. In this section, we describe how you can establish the RETOS development environment. A process for injecting RETOS kernel into sensor nodes, writing, compiling, and running RETOS applications is described in Chapter 2.

Followings are the requirements for establishing a development environment for RETOS and injecting the RETOS kernel and applications into sensor node devices. (For TmoteSky and Telos motes with TI MSP430 processors)

- A x86 PC with MS Windows
- At least 1 USB port for injecting THE RETOS kernel and applications to the motes

1.1 Installing a Basic Development Environment

To establish THE RETOS development environment, following applications are required

- MS Windows + Cygwin on x86 PC
- MSP430 GCC compiler (msp430-gcc)
- MSP430 serial programmer (msp430-bsl)
- MoteList Utility (to check the list of TmoteSky or Telos motes connected to PC)

First of all, you should download Cygwin(<http://www.cygwin.com/>), MSP430 GCC, serial

programmer and Motelist to install them. 1) you can download a MSP430 GCC installation file at <http://mspgcc.sf.net> 2) and a zip file for the MSP430 serial programmer (BSL: Boot Strap Loader) at <http://moteiv.com/support/msp430-bsl.zip> (this BSL is specially patched for TmoteSky and Telos motes, so you must download it through this web link) 3) you can also download MoteList at <http://moteiv.com/support/motelist.zip>. Then, unzip msp430-bsl.zip and motelist.zip into the mspgcc/bin directory. After you finish installing all these three software, proceed to the next step.

Configuring PATH for msp430-bsl and MoteList will be useful since these are frequently used commands. You can do this by adding the following line at .bashrc file in your home directory (if .bashrc doesn't exist, make a new one)

e.g.) If msp430-bsl is installed at C:\Program Files\mspgcc\bin, add a following line at the end of .bashrc file.

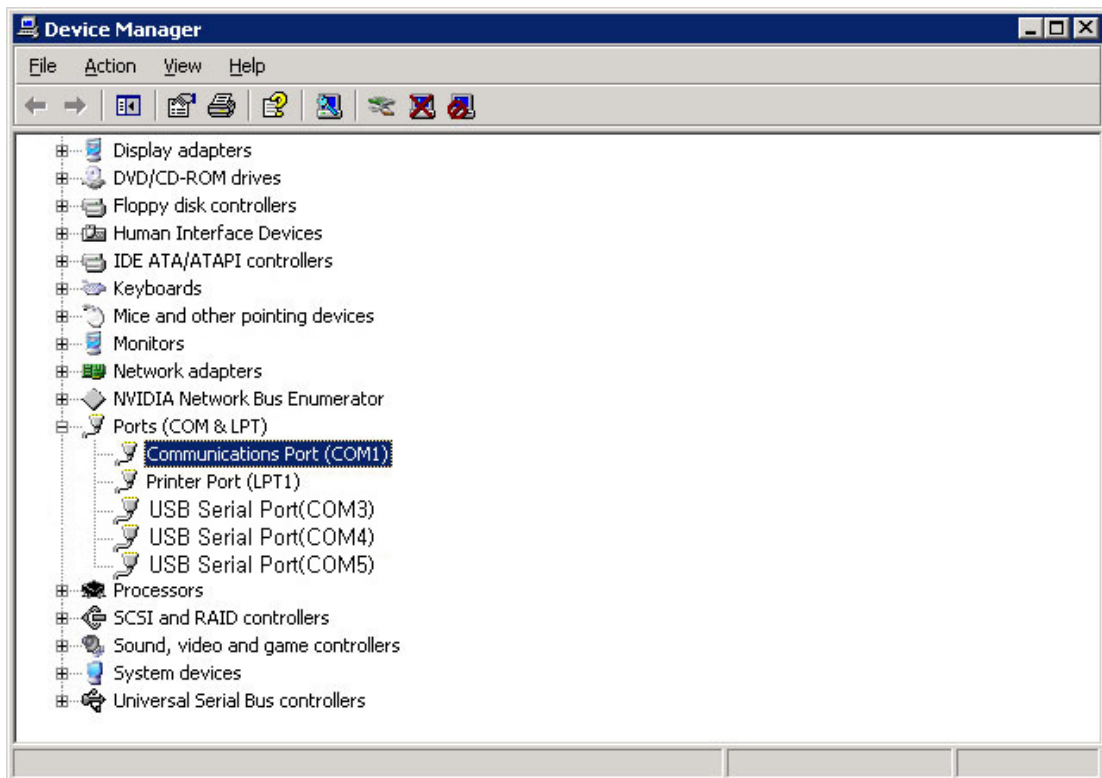
```
■ export PATH="${PATH}"/:/cygdrive/c/Program\ Files/mspgcc/bin
```

[TIP]

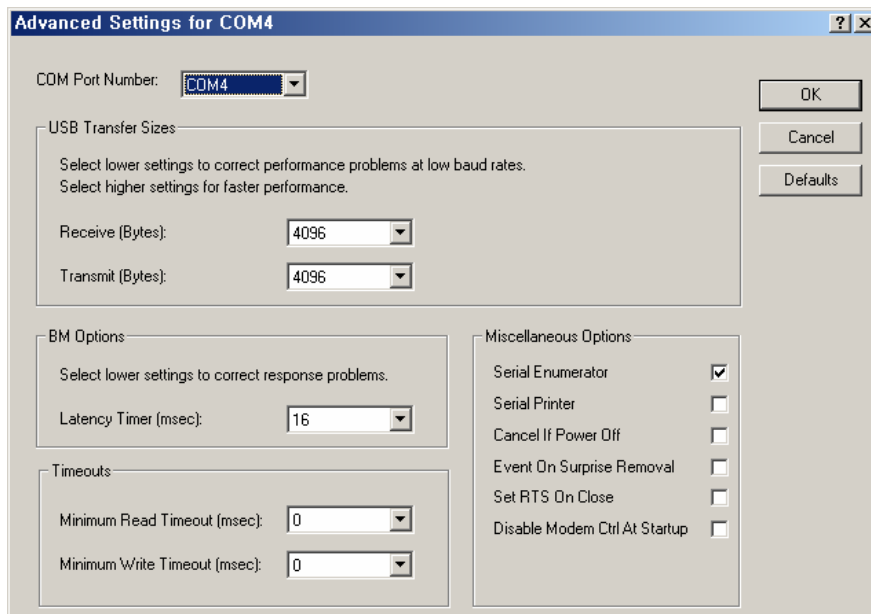
- If the development environment for TinyOS is already established on your system, you don't have to install Cygwin again. Check if msp430 cross compiler is already installed.
- If you have two cygwin1.dll file on the same computer, the cygwin will complain about the compatibility issue. The mspgcc installation version automatically installs additional cygwin1.dll into mspgcc/bin directory even though you have already installed the cygwin. Therefore, you should delete cygwin1.dll in mspgcc/bin directory to make everything work fine.
- The motelist utility requires VCP (Virtual COM Port) driver. The following chapter describes how to install VCP driver.

1.2 USB Driver Installation & Configuration

TmoteSky and Telos include USB to Serial Converter, so the RETOS and its applications can be injected through it. Virtual COM Port driver should be installed to recognize your sensor motes and program the system on host PC. You can download it from FTDI Chip web site. (<http://www.ftdichip.com/Drivers/VCP.htm>) If you connect your sensor motes to your host PC after installing the VCP driver, each mote will be mapped to each COM port as in the following figure.



In this figure, three motes are connected to COM3, COM4 and COM5, respectively. This Virtual COM Port allows several motes to connect to the host PC and work simultaneously. Port numbers can be modified by changing COM Port Number at Port Setting > Advanced, and you can see the port number changed after reconnecting the mote of which you just changed the port number.



[Caution] Cygwin released before Oct. 2006 is compiled to recognize maximum 15 serial devices. Virtual COM ports should be mapped to COM1 to COM15 since the RETOS application installation tool injector uses the Cygwin standard library.

```

RETOS: retos/kernel
[netropy:skylooker:/kernel/] $> motelist
Reference  CommPort  Description
-----
M4A2L304   COM3       tmote sky
M4AD49PR   COM4       tmote sky
M49ZI1RC   COM5       tmote sky
[netropy:skylooker:/kernel/] $>

```

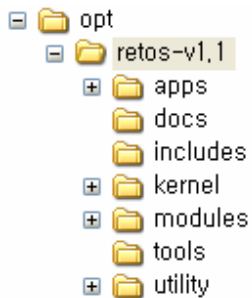
You can check if each mote is connected by typing “motelist” command.

1.3 RETOS Installation

You can install RETOS after finishing the previous section. You finish RETOS installation by downloading the RETOS kernel binary and application development tools and decompressing them into certain directory.

Following is the directory structure of RETOS v1.1.X when you install it at /opt.

- /apps: Makerules for application sources and compilation
- /docs: User tutorial
- /includes: Header files for application compilation
- /kernel: Kernel binary
- /modules: Module binaries
- /tools: Tools for building and installing applications
- /utility: Makerules for host PC application sources and compilation



2

Running the RETOS Kernel and Application Programs

Unlike common operating systems (i.e., TinyOS, uC-OS/II) for micro-controller hardware, the kernel and application binaries work separately on RETOS. Therefore, if you want to run application programs on RETOS,

1. The RETOS kernel should be injected on a sensor mote (writing a kernel image on the Flash ROM)
2. You can install and run several kinds of applications on the motes with the RETOS kernel after compiling your applications.

After installing the application on the motes with RETOS, it will be run automatically. If you press the RESET button (see following figure if you are using TmoteSky), system will be restarted and check if any application program is on the Flash ROM and if it does, run it. Following describes how to inject the kernel image and develop applications on hardware.

2.1 RETOS Kernel Installation

You can download the kernel on TmoteSky or Telos by using the Virtual COM serial port. You need to install the following tools to download the kernel on your motes.

- MSP430 GCC Toolchain v3.2.x
- USB Virtual Com Port Driver

Injecting the RETOS kernel uses USB, so sensor motes should be connected to the host PC through USB.

Following steps are required for injecting RETOS kernel into the Telos motes. (in this case, “retos/” is the root directory for RETOS)

```

RETOS: retos/kernel
[netropy:skylooker:/kernel/] $> make tmote_install COM=com4 ADDR=1
make[1]: Entering directory `/opt/retos/kernel'
make[2]: Entering directory `/opt/retos/kernel'

#####
# for your sense                                     #
#                                                     #
#               R E T O S                             #
#                                                     #
# Copyright(c)2007 Yonsei Univ. All Rights Reserved.  #
#####
                                                    vl.1

MSP430 Bootstrap Loader Version: 1.39-telos-7
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
33294 bytes programmed.
Reset device ...
make[2]: Leaving directory `/opt/retos/kernel'
make[1]: Leaving directory `/opt/retos/kernel'
[netropy:skylooker:/kernel/] $>

```

1. Connect your mote to USB
2. Move to “retos/kernel” directory
3. Run “make tmote_install” (if you use TmoteSky)

You can use ADDR to set a specific address of your mote. This address acts like an IP address of TCP/IP and is used for operations such as sending packets to the specific address through RF communication channel. If you want to inject a kernel with different address, set ADDR as following line.

- make tmote_install ADDR=2

(If you want to get the address of your mote in application, you can use the identifier named `LOCAL_ADDR`. This will be explained later in the section for making application programs)

If you type “make `tmote_install`” without setting `COM` field, a mote on `COM3` will be tried to be injected by default. Therefore, if you want to use another port, set `COM` field. For example, if you want to use `COM4`, type “make `tmote_install COM=3`” since `COM4` is “`/dev/ttyS3`”.

After finishing above steps, the RETOS kernel will be installed on your mote. Flash ROM will be refreshed after this, so previously installed applications won’t exist anymore on your mote.

Sensor node is hardware without MMU and privileged mode. RETOS has a software functionality which detects application malfunctions, but there can still be other undetected errors or kernel bugs which make system malfunction. (e.g. after running certain applications, they can’t be reinstalled) In this case, you have to re-inject the kernel so that your mote is initialized.

RETOS supports various hardware platforms. Following table is the list of hardware platforms that RETOS supports now.

Vender	Platform	Command
MoteIV	Tmote Sky	make <code>tmote_install</code>
MoteIV	Telos Rev.b	make <code>telos_install</code>
MoteIV	Telos Rev.a	make <code>telosa_install</code>

2.2 Compiling and Installing Application Programs

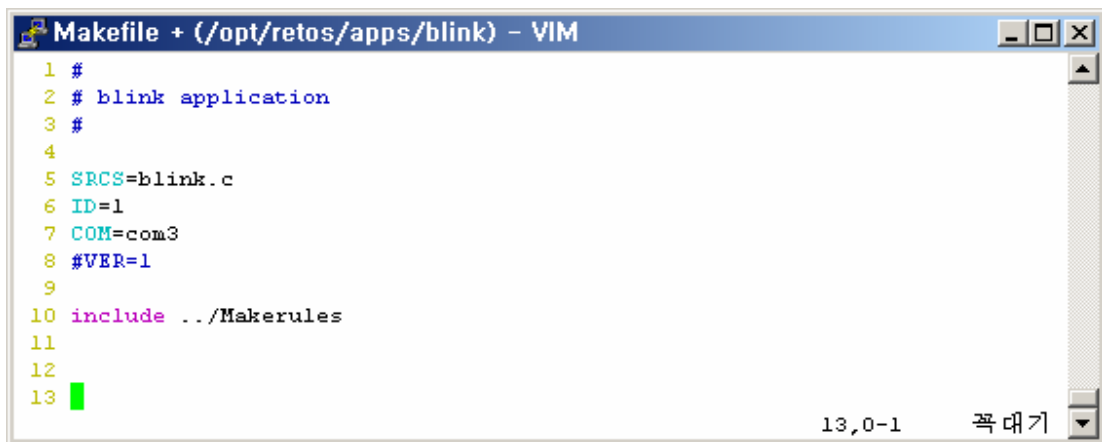
Following list is the programs you need for compiling and installing the RETOS application programs. If you established your development environment properly as described in Chapter 1, these programs should be already installed on your PC.

- MSP430 GCC Toolchain v3.2.x
- RETOS Application Injector (included with RETOS release)

Following list describes the general steps for writing, compiling and injecting RETOS application programs

1. make a directory in “retos/apps” to write your application program (e.g. blink)
2. Write “Makefile” in that directory. This will be used when you compile your application program. (i.e. retos/apps/blink/Makefile)
3. Write source code of your application program
4. Compile your application program (make tmote)
5. connect your mote to PC (check its COM port number)
6. Inject your application program (make injector)

To compile your application program, you need “Makefile”. Makefile should include the list of source files, application ID and serial port number to inject your application program.



```

1 #
2 # blink application
3 #
4
5 SRCS=blink.c
6 ID=1
7 COM=com3
8 #VER=1
9
10 include ../Makerules
11
12
13 █
  
```

13,0-1 꼭대기

Above figure is an example of “Makefile” for the application, “blink”. Followings are the descriptions of each field.

- **SRCS:** Write source files(.c files) sequentially. In “blink” application, there is only one source file, so “SRC=blink.c” is written. If you use more than one source files, write as following. “SRC=blink1.c blink2.c”
- **ID:** The ID of your application program (File ID, FID). An application program ID

acts like a file name in RETOS. If both the currently running program and your program to be installed have an identical ID, currently installed program will be deleted and your new program will be injected instead. Therefore, if you want to run several application programs at the same time, their file IDs should be distinguishable. You can use the numbers from 1 to 255 for your application program file ID.

- **COM:** A serial port number which you use when you inject your program. You can set this value with “make” command. For your convenience, you might prefer to write the frequently used port number in your “Makefile”. If you want to use COM3, write “COM=2” or “COM=com3”.
- **VER:** This means the version of your application program. Increasing this value whenever you modify or release your application program is recommended for code dissemination since it disseminates application programs with higher version. [refer to appendix section for code dissemination] You can use 1 to 255 for a version number. If the version value is 0, corresponding application program can’t be disseminated by code dissemination algorithm, so this can be used to avoid code dissemination. If you don’t set this value, it will be 1 by default.
- **include ../Makerules:** Do not alter this line since this is essential for compiling and injecting application programs.

Fields such as SRCS, ID and COM in “Makefile” still can be determined when you enter “make” command. For example, you can describe source files like `make tmote SRCS=”blink1.c blink2.c”` when you compile your application program or describe a port number and program file ID like `make injector COM=4 ID=4` (when you use COM5) when you inject your application program.

2.3 Example of Injecting an Application Program

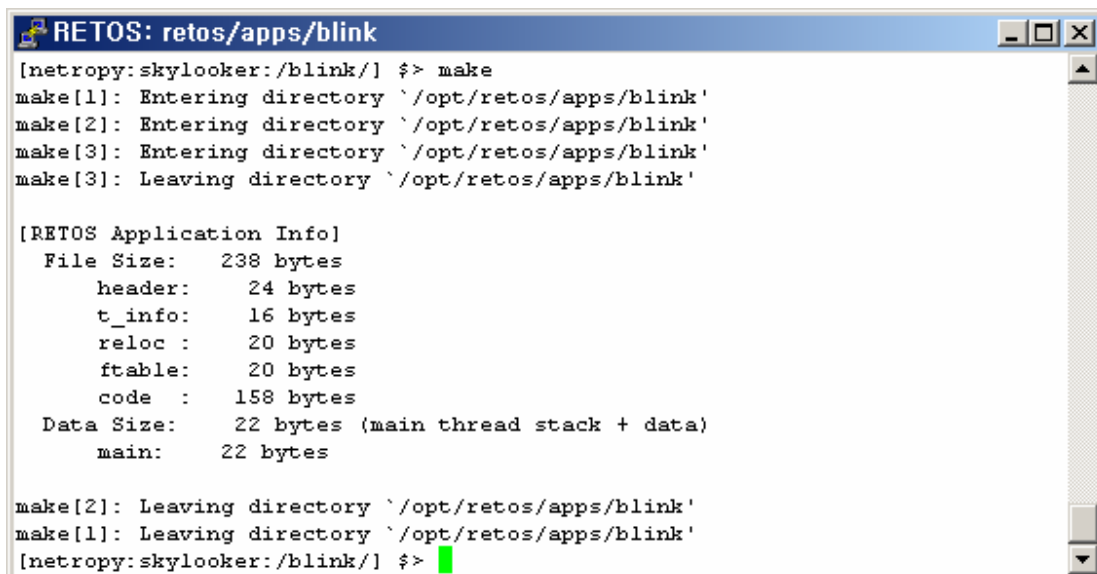
Blink program

This example of compiling and installing “blink” program which toggles LED periodically is intended for explanation of application program compilation and injection.

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define SLEEP_TIME 1000

int main()
{
    while(1)
    {
        led3_toggle();
        sleep(SLEEP_TIME); // 1000ms
    }
    return 0;
}
```



The screenshot shows a terminal window titled "RETOS: retos/apps/blink". The terminal output displays the execution of the 'make' command, which enters and leaves the directory '/opt/retos/apps/blink'. It then provides application information for the compiled program:

[RETOS Application Info]	
File Size:	238 bytes
header:	24 bytes
t_info:	16 bytes
reloc :	20 bytes
ftable:	20 bytes
code :	158 bytes
Data Size:	22 bytes (main thread stack + data)
main:	22 bytes

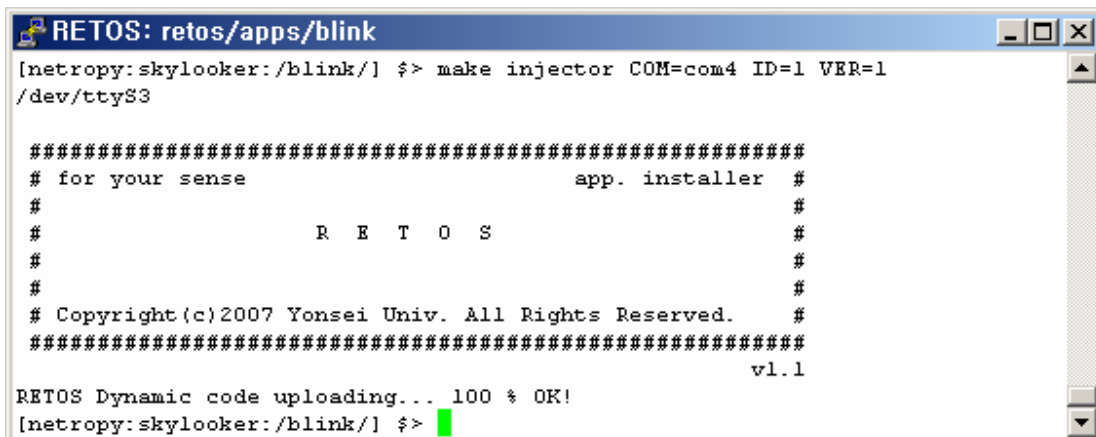
After the information, the terminal shows the program leaving the directory and returning to the prompt. The prompt is a green cursor on a green background.

1. move to “retos/apps/blink” directory
2. compile an application program for TmoteSky (make tmote)
3. inject compiled application program (make injector)

If you enter “make tmote”, this will compile your application program. As you can see in the above figure, several information of your application is displayed when it is compiled successfully. That information consists of File Size and Data Size. File Size is the size of bytes which will be written in Flash ROM of the mote. In this figure, the file size is 234 bytes and code size without meta-data is 154 bytes. Data size consists of stack usage of main thread and data section requirement of the application program. In this figure, it is 22 bytes. If you write a multi-threaded program using “pthread”, RAM usage of each thread will be displayed below Data Size. (Blink is a single-threaded program)

You have to consider the File Size and Data Size of the application program due to the limited resources of sensor motes. For instance, RETOS provides the application program with 18Kbyte of Flash ROM and 8Kbyte of RAM on TmoteSky. Therefore application program which requires more File and Data size can’t be loaded. And if you want to run more than one application program, The total ROM/RAM requirements of the program should not exceed 18KB/8KB limitation.

If you run “make injector” command, you can see that mote’s LEDs are toggled periodically. This means that your application program has been injected and is running on the RETOS kernel. Once the application program is injected, it will remain there although you reset the mote and starts automatically. Following figure shows the process of injecting your application after setting the application version.



```

RETOS: retos/apps/blink
[netropy:skylooker:/blink/] $> make injector COM=com4 ID=1 VER=1
/dev/ttyS3

#####
# for your sense                app. installer #
#                                #
#          R E T O S            #
#                                #
#                                #
# Copyright(c)2007 Yonsei Univ. All Rights Reserved. #
#####
                                v1.1
RETOS Dynamic code uploading... 100 % OK!
[netropy:skylooker:/blink/] $>

```


2.4 Kernel Module Installation

RETOS provides dynamically loadable kernel modules for expending its functionalities. Kernel module is a kernel code which modularizes kernel supporting functionalities so that they can be loaded and used whenever they are necessary. Kernel modules can be drivers, routing protocols or a code dissemination module, etc. They are either compiled with kernel codes or loaded after kernel is already installed.

Kernel image and linking

You can define how a RETOS module will be linked by modifying kernel Makefile.

- **MODULE_LIST**: Add a module type [mod|app], module ID, module name in module list. Module type has two types; module and application. Module ID can be 1 to 255, and this should be distinguishable with other modules since this value is used for a module file name. It's good to set module names as directory names of modules or applications. Kernel can include at most 10 modules.

Example:

```
[kernel/Makefile]
MODULE_LIST = mod 128 recode_mod
MODULE_LIST += mod 140 pns_mod
MODULE_LIST += app 1 blink
```

retos/modules/recode_mod,
retos/modules/pns_mod,
retos/apps/blink are included.

After modifying Makefile as shown above, prepare your kernel image with modules by entering compile command. Now you can install your kernel on your mote as explained before.

```
[retos/kernel/] $> make tmote
```

Dynamic loading on already installed kernel

Following list shows the required program for RETOS module feature. These programs should be already installed if you established your development environment properly as described in Chapter 1.

- RETOS Application Injector (Included in RETOS installation release)

These are the general steps for injecting the RETOS modules

1. move to corresponding directory in retos/modules (ex: blink_mod)
2. connect your mote to your PC (check your COM port number)
3. install your module (make injector)

These are required options for module installation.

- **ID:** Write a file ID (File ID, FID) of an application program. Module ID acts like a file name on RETOS. If currently installed program and your program to be installed have an identical ID, new program will be injected deleting the old one. Module ID can be 1 to 255.
- **COM:** This is a serial port number which will be used for program injection. If you want to use COM3, write “COM=2” or “COM=com3”.

e.g.) `make injector COM=com3 ID=3`

3

Writing RETOS Application Programs

This chapter describes how you use necessary functionalities of RETOS when you write your application programs. You should be familiar with system calls and multi-thread libraries to write RETOS application programs. System calls are usually used to use hardware resources such as sensors or radio communication. You can also use them to write program which consists of several threads by using the pthread library. Here are meanings of several variable types.

- `uint8_t` / `int8_t` means 8bit unsigned / signed int,
 - `uint16_t` / `int16_t` means 16bit unsigned / signed int
 - `uint32_t` / `int32_t` means 32bit unsigned / signed int
 - `float` / `double` means 32bit variable
- int** and **int16_t** are the same types and **char** and **int8_t** are the same types.

Following chapters describe the system calls and macros for writing a RETOS application. In chapter 4, system calls needed to perform serial communication are explained. In chapter 5 and 6, system calls related to radio communication are described. Threading and thread scheduling related system calls are explained in chapter 7 and 8, respectively. System calls related to sensor reading are described in chapter 9 and miscellaneous system calls are explained in chapter 10.

4

Serial Communications

RETOS supports serial communication for communication between motes and host PC. The serial communication is also used for loading modules and applications. RETOS provides serial communication stack to prevent collisions among application programs and system loadings. Each application program can use different port number on their data communication. Hence, system and applications can communicate with host PC independently by setting different port numbers.

The serial port should not be equivalent with other ports. The serial ports used or reserved by the RETOS kernel and system modules are listed below. (port_list.txt) Users need to use different port number from following ports.

- 177(0xB1): Kernel shell
- 178(0xB2): RMTTool shell
- 190(0xBE): Injector/ISP
- 254(0xFE): Application error reporting (Reserved)
- 255(0xFF): Reserved

Serial communication mode settings for RETOS are as follows:

Parameter	Value
Speed	57600 (bits/sec)
Data bit	8
Parity	N
Stop bit	1
Flw control	N

4.1 Mote Side

In RETOS, the following system calls are provided for mote-PC serial communication.

int serial_send(uint8_t port, uint16_t length, uint8_t *data);

Transmit data to host PC through serial communication. Non-blocking function. Return transmitted data length when it success. Return 0 when it fails to transmit.

port Port number of a destination application. Support 0 to 255 (*the 'port' does not mean address of serial port number such as COM1 of MS Windows or /dev/ttyS0 of Linux)

length Length of the transmitting data.

data Pointer to the transmitting data.

serial_send can transmit up to 30bytes of data at once. If a user tries to send data more than 30bytes, serial_send would transmit 30bytes of data only and return 30.

int serial_sendto(uint8_t port, uint16_t length, uint8_t *data);

Does not wait for the ACK frame of host PC after transmitting data. Appropriate for high data rate communication. Usage and return values are same as those of serial_send().

int serial_recv(uint8_t port, uint16_t length, uint8_t *data);

Receive data from a host PC through serial communication. Blocking function. Return received data length when it success. Return 0 when it fails to receive.

port port number used for communication

length size of the buffer for receiving data

data pointer to the buffer

4.2 Host PC Side

In this section, we introduce how to control the serial packets on host PC side. The RETOS development tool provides several libraries to use the RETOS serial communication protocol.

```
int serial_open(const char *dev, void* discnct_handler);
```

Open a serial port and connect to a mote. If it success to open a serial port, it returns file descriptor number (greater than 0). If it fails, it returns -1.

dev connecting COM Port in the form of null-terminated string. e.g.)
 “com3”, “ttyS2” or “2”

discnct_handler pointer to the callback function which would be executed when the
 serial connection has been disconnected. Enter NULL if there isn't any.

```
int serial_close();
```

Finish the serial communication with a mote and close the serial port of PC.

```
int serial_recv(uint8_t port, uint16_t length, uint8_t *data);
```

```
int serial_send(uint8_t port, uint16_t length, uint8_t *data);
```

Functions for data transmission and reception of serial communication. As application program in motes, host PC needs to set up the communication port number. Data from other port s will be discarded. Usage of the functions is same as the system calls on mote side.

4.3 Examples

Sample code for testing serial communication - Mote (apps/serial_test/serial_test.c)

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define SERIAL_PORT 4

int main()
{
    uint16_t count = 0;

    while(1) {
        count = (count + 1) & 0x7;
        leds_set(count, LEDS_NORMAL);

        serial_send(SERIAL_PORT, sizeof(uint16_t), (uint8_t*)&count);

        sleep(1000); // 1000ms
    }
    return 0;
}
```

**Sample code for testing serial communication - Host PC
(utility/serial_test/serial_test.c)**

```
#include <stdio.h>
#include <serial.h>

#define PORT 4

int main (int argc, char* argv[])
{
    uint16_t buf;

    // argument check
    if (argc < 2)
        return 1;

    // initialize serial driver
    if (serial_open(argv[1], NULL) < 0)
        return 1;

    while (1) {
        // receive thread info
        while (serial_recv(PORT, sizeof(uint16_t), (uint8_t*)&buf) == 0);

        printf("%u\n", buf);
    }
    // close serial port
    serial_close();
    return 0;
}
```


5

Radio Communications

RETOS supports radio communication (RF) for motes. The RETOS application programs can communicate with each other using RF packets. Applications can use different port number on their radio communication for independent radio communication. The radio port should not be equivalent with other radio ports. The radio ports used or reserved by RETOS system are listed below. Users need to use different port number from following ports.

- 170(0xAA): FTSP daemon
- 220(0xDC): NSL Query
- 221(0xDD): NSL Info.
- 244(0xF4) ~ 248(0xF8), 250(0xFA): RMTTool
- 252(0xFC): Ultrasound (us_radio_send/recv)
- 254(0xFE): Application error reporting (Reserved)
- 255(0xFF): Reserved

5.1 Interface

The RETOS radio data packet (rf_pkt)

Length	fcfhi	fcflo	seq
dstpan		Dstaddr	
srcaddr		port	group
Time			
data (32bytes)			
RSSI	LQI		

Important Fields

- **length:** Size of transmitted data
- **dstaddr:** destination address of receiver mote
- **srcaddr:** source address (local address of the transmitting mote)
- **port:** port number of a packet
- **data:** payload area, cannot exceed 32 bytes
- **time:** this field is reserved by ETA and future purpose.
- **RSSI:** data from the radio's Received Signal Strength Indicator
- **LQI:** data from the radio's Link Quality Indicator

System Calls

Following system calls are provided for the RETOS application programs to use radio communication.

```
int radio_send(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);
int radio_sendto(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);
int radio_recv(uint8_t port, rf_pkt * msg);
```

These functions are used for data communications within one hop range. If the mote specified with the argument, 'addr', is not in the one hop range, the transmission request will be failed.

`radio_send()`, `radio_sendto()` and `radio_recv()` are blocking functions. `radio_send` is used for reliable transmission as in TCP of common network stack. The function blocks until it receives acknowledgment from destination. `radio_sendto` blocks until the radio transceiver finishes the transmission.

When the communication succeeds, the functions return the length of payload that have transmitted or received. Otherwise they return -1.

`addr` unique address of destination mote. Use `BROADCAST_ADDR` to broadcast a message.

`port` port number specified by each application. the sender and receiver communicate when their port numbers are identical

`length` length of communicating data. The data length is less than or equal to 32bytes

`msg` address of received `rf_pkt` structure. Transmitted data should be stored in data field of the structure. While receiving a radio packet, various fields refer to important information including actual data.

e.g.) `msg->length` is the length of received data

```
int set_rfpower (uint8_t value);
```

Setting the RF transmission power. The value should be between 1 and 31.

```
uint8_t get_rfpower ();
```

Returns the current value of RF transmission power.

```
uint8_t set_channel(uint8_t ch);
```

Setting the current radio channel used by RF transmission. The argument 'ch' should be between 11 and 26.

Following functions require NSL (Network Support Layer). The NSL is a network stack that manages information of neighbor nodes in one-hop range. Please refer to Chapter 5.2 for further details.

```
void set_sink_loc(uint16_t addr, uint16_t pos_x, uint16_t pos_y, uint16_t pos_z);
```

Set up the mote address and position of sink node. These information are used for send_sink. Requires NSL (The Network Support Layer)

addr address of the sink
pos_x x-coordination of the sink
pos_y y-coordination of the sink
pos_z z-coordination of the sink

```
int send_sink(uint16_t addr, uint8_t port, uint8_t length, rf_pkt* msg,  
             uint8_t module, uint8_t param);
```

Send data from node to sink through routing. To use send_sink(), routing modules must be installed. Currently, RETOS supports PNS (Parametric Neighbor Selector) module for routing. Requires NSL (The Network Support Layer)

addr address of a sink node. The sink has to be registered through 'set_sink_loc'.
port port number specified by each application.
length length of the data.
msg address of received rf_pkt structure.
module routing module identifier. Choose routing module for transmission.
 e.g.) PNS_MODULE
param pass parameters required for routing modules.

```
int send_nbr(uint8_t port, uint8_t length, rf_pkt* msg);
```

Transmit radio to motes recorded in Neighbor Table. (1-to-neighbor)

Usage and parameters are same as radio_send. Requires NSL (The Network Support Layer)

```
int send_net(uint8_t port, uint8_t length, rf_pkt* msg);
```

Transmit radio to all motes in the network. (1-to-all, flooding) Usage and parameters are same as radio_send(). Requires NSL (The Network Support Layer) (Not implemented yet)

5.2 NSL: The Network Support Layer

RETOS supports Network Support Layer which is a network stack that manages information of neighbor nodes in one-hop range. Basically, NSL maintains a neighbor table that stores real-time neighbor information and provides the information upon user's requests. Each node periodically broadcasts the messages containing its information and the neighbor table is generated with them. The broadcast messages include a node identifier, a position, and battery status. A node in one-hop range which receives the broadcast messages only stores the information of other nodes with a good link quality.

NSL API functions

uint16_t get_update_interval()

Returns the updating interval of the neighbor table

void set_update_interval(uint16_t attr)

Sets the neighbor table update interval with a unit of seconds. The default interval is 10 seconds

uint16_t get_nb_cnt()

Returns the number of nodes in the neighbor table.

uint16_t get_nb_id(uint8_t node_seq)

Returns the identifier of the node specified in the argument, 'node_seq'. 'node_seq' is an index of the neighbor table.

uint16_t get_nb_lqi(uint8_t node_seq)

Returns the LQI(Link Quality Indicator) of the specified node.

int8_t get_nb_rssi(uint8_t node_seq)

Returns the RSSI(Received Signal Strength Indicator) of the specified node.

uint16_t get_nb_battery(uint8_t node_seq)

Returns the amount of the battery left on the specified node.

uint16_t get_nb_pos_x(uint8_t node_seq)

Returns the x-coordinate of the specified node.

uint16_t get_nb_pos_y(uint8_t node_seq)

Returns the y-coordinate of the specified node.

uint16_t get_local_pos_x()

Returns the x-coordinate of the node itself.

uint16_t get_local_pos_y()

Returns the y-coordinate of the node itself.

void set_local_pos_x(uint16_t attr)

Sets the x-coordinate of the node itself.

void set_local_pos_y(uint16_t attr)

Sets the y-coordinate of the node itself.

An example of printing the NSL neighbor table

```
#include <syscall.h>

// nsl test serial port
#define SERIAL_PORT 4

int main()
{
    uint8_t nb_count,i;
    int16_t buf[3];
    while(1) {
        nb_count = get_nb_cnt();
        for(i=0;i<nb_count;i++) {
            led1_on();
            sleep(50);
            led1_off();
            sleep(100);
            buf[0] = get_nb_id(i); // identifier
            buf[1] = (int16_t)get_nb_rssi(i); // rssi value
            buf[2] = get_nb_battery(i); // battery
            serial_send(SERIAL_PORT, sizeof(int16_t)*3, (uint8_t*)&buf);
        }
        sleep(1000);
    }
    return 0;
}
```

5.3 Examples

Sample code for sending RF packet (apps/radio_send/radio_send.c)

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define SLEEP_TIME 250
#define RADIO_PORT 10

typedef struct int_msg {
    int count;
} int_msg;

rf_pkt msg;

int main ()
{
    uint16_t count = 0;

    while (1) {
        int_msg *data = (int_msg*)msg.data;

        count = (count + 1) & 0x7;
        data->count = count;

        if (radio_send(BROADCAST_ADDR, RADIO_PORT, sizeof(int_msg), &msg)
        < 0)
            led3_toggle();
        else
            led2_toggle();

        sleep(SLEEP_TIME);
    }
    return 0;
}
```

This example code increases the value *count* at each iteration and broadcast the value. It declares an *rf_pkt* structure, copies *count* value into *data* field of the structure and calls *radio_send* system call. The first parameter of the system call, *BROADCAST_ADDR*, indicates

broadcasting address which is 0xFFFF. To transmit the packet to a specific mote, input the destination mote's ID(address) instead of BRAODCAST_ADDR.

Sample code for receiving RF packet (apps/radio_recv/radio_recv.c)

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define SLEEP_TIME 250
#define RADIO_PORT 10

typedef struct int_msg {
    int count;
} int_msg;

int main()
{
    rf_pkt recvd_msg;

    while(1) {
        // radio_recv: blocking I/O
        radio_recv(RADIO_PORT, &recvd_msg);

        // type casting
        int_msg *data = (int_msg*)recvd_msg.data;

        // print
        leds_set((uint8_t)(data->count), LEDS_NORMAL);
    }

    return 0;
}
```

This example shows how to use the *radio_recv()* system call. Once a packet is received, the system call stores the packet into the structure which is passed as the second parameter. The system call blocks until the reception process finishes. To perform other works during the blocking time, user can generates some threads. Port number of receiver should correspond to that of sender.

Sample code for sending RF packet to a sink (apps/sendsink_test/sendsink_test.c)

```
#include <stdio.h>
#include <syscall.h>

#define SLEEP_TIME 500
#define RADIO_PORT 10
#define SINK_ADDR 1

typedef struct int_msg {
    int count;
} int_msg;

rf_pkt msg;
uint16_t count = 0;

void init()
{
    // setup sink location (sink_addr, pos_x, pos_y, pos_z)
    set_sink_loc(SINK_ADDR, 0, 0, 0);

    // setup my location
    if(LOCAL_ADDR == SINK_ADDR) {
        set_local_pos_x(0);
        set_local_pos_y(0);
    } else if(LOCAL_ADDR == 2) {
        set_local_pos_x(10);
        set_local_pos_y(0);
    }
}

int main()
{
    init();
    while(1) {
        int_msg *data = (int_msg*)msg.data;
        count = (count + 1) & 0x7;
        data->count = count;
        // send to sink (sink_addr, port, length, msg,
module_name,
module_parameter)
        if (send_sink(SINK_ADDR, RADIO_PORT, sizeof(int_msg),
&msg,
```

```
PNS_MODULE, RELIABILITY) < 0)
    led3_toggle(); // send fail
else
    led2_toggle(); // send success

    sleep(SLEEP_TIME);
}
return 0;
}
```

The usage of `send_sink` is similar to that of `radio_send`. Instead, `send_sink` uses routing modules and transmits radio data to a sink through multi-hop routing. Hence, routing modules must be installed to use `send_sink`. Currently, RETOS supports PNS (Parametric Neighbor Selection) module. (See Appendix D. The PNS Module)

Sample code of RF transmission to neighbors and entire network

```
#include <stdio.h>
#include <syscall.h>

#define SLEEP_TIME 250
#define RADIO_PORT 10

typedef struct int_msg {
    int count;
} int_msg;

int main ()
{
    rf_pkt msg;
    uint16_t count = 0;
    while (1) {
        int_msg *data = (int_msg*)msg.data;
        count = (count + 1) & 0x7;
        data->count = count;

        // send to neighbor (port, length, msg)
        if (send_nbr(RADIO_PORT, sizeof(int_msg), &msg) < 0)
            led3_toggle();

        // send to network - flooding (port, length, msg)
        if (send_net(RADIO_PORT, sizeof(int_msg), &msg) < 0)
            led2_toggle();

        sleep(SLEEP_TIME);
    }
    return 0;
}
```

send_nbr() sends RF packets to its neighbors. The usage of send_nbr is similar to that of radio_send. The system call transmits a packet to all neighbors instead of particular destination node. send_net broadcasts a packet to the entire network. (Not implemented yet.)

6

Threading

RETOS provides some of the POSIX thread library to support multi-threading. The RETOS thread libraries include mutex and conditional variables for thread creation, termination, or synchronization. The usage and functionality of the RETOS pthread library are similar to the general UNIX/Linux's. The followings describes the RETOS threading functions.

6.1 Interface

Thread creation and cancellation

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg);
```

Creates a thread. This function returns 0 in case of SUCCESS, -1 in case of FAIL.

thread used for identification when a thread is created.

att ignored in RETOS Pthread library. (set to NULL)

start_routine is a thread function. Do not use pointer assignment, assign the function name.

arg used for argument passing.

```
int pthread_join (pthread_t th, void **thread_return);
```

A blocking function which waits until the thread is terminated. Returns if the thread is terminated or the waiting thread does not exist.

th waiting thread identifier.

thread_return used for the return value of a waiting thread. The thread return value can

be passed with a pointer. Set to NULL if not used.

```
int pthread_cancel(pthread_t thread);
```

Send termination request to the thread which is pointed by the thread identifier. The thread that received termination request is always terminated because a switching function which changes the status of thread cancellation function (`pthread_setcancelstate`) does not exist.

Thread synchronization

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

Initialize a mutex object. RETOS does not use “attr” field. (set to NULL)

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Request locking a mutex variable. If the mutex is already locked, the calling thread will be blocked until the mutex becomes available

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Equivalent to `pthread_mutex_lock()`, except that if the mutex object is already locked the function will return -1 immediately

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Unlock a mutex.

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_cond_attr *attr);
```

Initialize the condition variable. RETOS does not use “attr” field. (set to NULL)

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Unblock at least one of the threads that are blocked on the specified condition variable, cond (if there is any). If more than one thread is blocked on a condition variable, send a signal to only one of them. A signal is ignored when no thread is waiting.

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Unblock all threads that are currently blocked on the specified condition variable, cond.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Block current thread on the condition variable, `cond`. Before blocking, it will unlock the mutex and acquire the mutex lock again when this function is about to finish due to the reception of condition variable signal.

6.2 Examples

Example for using pthread library (apps/pthread_exam/pthread_exam.c)

```
#include <io.h>
#include <stdio.h>
#include <pthread.h>
#include <syscall.h>

#define SLEEP_TIME 1000

pthread_mutex_t mutex_lock;
int count = 1;

void *thread_func(void* arg)
{
    while(1) {
        pthread_mutex_lock(&mutex_lock);
        if(count == 0) led3_off();
        else led3_on();
        pthread_mutex_unlock(&mutex_lock);
        sleep(SLEEP_TIME);
    }
}

int main()
{
    pthread_t thread_t;
    pthread_mutex_init(&mutex_lock, NULL);
    pthread_create(&thread_t, NULL, (void*)thread_func, NULL);
    sleep(SLEEP_TIME/2);
    while(1) {
        pthread_mutex_lock(&mutex_lock);
        count = (count + 1) % 2;
        if(count == 0) led2_off();
        else led2_on();
        pthread_mutex_unlock(&mutex_lock);
        sleep(SLEEP_TIME);
    }
    return 0;
}
```

7

Thread Scheduling

RETOS provides a POSIX 1003.1b real-time scheduler interface. Execution of each thread is based on the three types of scheduling policies, and user can select the thread scheduling policy and priority by calling certain system calls. RETOS provides `SCHED_RR`, `SCHED_FIFO`, and `SCHED_OTHER` scheduling policies. `SCHED_RR` and `SCHED_FIFO`, which support soft real-time, are run with higher priority than the general thread (`SCHED_OTHER`).

`SCHED_FIFO` / `SCHED_RR`: Static priority

`SCHED_FIFO` and `SCHED_RR` have static priority. When a *`SCHED_FIFO`* process becomes runnable, it will always immediately preempt any currently running normal *`SCHED_OTHER`* processes. A *`SCHED_FIFO`* process runs until it is blocked by an I/O request, preempted by a higher priority process, or it blocks itself. *`SCHED_RR`* is a simple enhancement of *`SCHED_FIFO`*. Everything described for *`SCHED_FIFO`* before can also be applied to *`SCHED_RR`*, except that each process is only allowed to run for maximum time quantum. If a *`SCHED_RR`* process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A *`SCHED_RR`* process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum.

`SCHED_OTHER`: Dynamic priority

`SCHED_OTHER` has dynamic priority, and the priority is lower than `SCHED_FIFO` and `SCHED_RR`. A thread whose scheduling policy or priority is not explicitly defined is set to `SCHED_OTHER`. RETOS has “event-boosting” feature which is similar to the mechanism that

boosts the priority of I/O bound thread in Linux. The event-boosting elevates the priority of Timer/Radio/Sensing threads, so these threads can run before other threads.

A user can define the thread policy as `SCHED_FIFO`, `SCHED_RR`, or `SCHED_OTHER`, and can define the priority in case of `SCHED_FIFO` or `SCHED_RR`. The followings state system calls such as setting the policy and priority of scheduler, which follows POSIX interfaces.

Getting scheduling policy and priority

```
int sched_getscheduler (uint8_t pid);
```

A function which gets current scheduling policy.

pid indicates process ID(thread ID), which can be obtained by `getpid()`.

Returns one of `SCHED_RR`, `SCHED_FIFO`, and `SCHED_OTHER` when SUCCESS, otherwise returns -1. FAIL occurs when no corresponding pid exist.

```
int sched_getparam(uint8_t pid, struct sched_param *param);
struct sched_param {
    int sched_priority;
};
```

A function which gets the priority of current thread.

pid indicates process ID(thread ID), which can be obtained by `getpid()`.

param indicates the address where the priority of current thread will be stored.

Returns 0 when SUCCESS, otherwise returns -1.

Changing scheduling policy and priority

```
int sched_setscheduler (uint8_t pid, int policy, const struct sched_param *param);
struct sched_param {
    int sched_priority;
};
```

A function which sets the scheduling policy and priority. Returns 0 when SUCCESS, otherwise returns -1.

pid indicates process ID(thread ID), which can be obtained by `getpid()`. policy can be `SCHED_RR`, `SCHED_FIFO`, or `SCHED_OTHER`. Other arguments will cause return of -1. param specifies the priority in `sched_param` structure.

The priority has range of 0~9. The priority of SCHED_FIFO and SCHED_RR can be between 1 and 9. The priority of SCHED_OTHER is always 0. In here, the priority means static priority, different from the dynamic priority in SCHED_OTHER. (The dynamic priority of SCHED_OTHER is defined by the kernel.)

```
int sched_setparam(uint8_t pid, struct sched_param *param);
```

A function which is similar to sched_setscheduler. The difference is that this function sets only the priority of a thread.

8

Sensing

This chapter describes how to obtain sensor values in RETOS.

8.1 Interface

RETOS provides the following system calls to read sensor values.

uint16_t sensor_read (uint16_t type);

A system call which reads a sensor value. Returns the sensor value. If the sensor reading fails, returns 0.

When `sensor_read()` is called repeatedly, some delay time (a few or several tens of milliseconds) is required. It varies based on the types or the numbers of sensors that are being used in your application.

`type` the type of sensor to be read.

Supported Sensors:

1 - LIGHT (TmoteSky, Telos)

2 - HUMIDITY (TmoteSky, Telos)

3 - TEMPERATURE (TmoteSky, Telos)

8.2 Examples

Sensor reading application program (apps/sensor_test/sensor_test.c)

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define FREQ    100
#define SLEEP_TIME  (1000 / FREQ)
#define PORT     1
#define LENGTH  (30 / sizeof(uint16_t))

#define SENSOR LIGHT                // TmoteSky
// #define SENSOR TEMPERATURE        // TmoteSky
// #define SENSOR HUMIDITY           // TmoteSky

int count = 0;
uint16_t buf[LENGTH];

int main()
{
    while(1) {
        // micro-second sleep
        // usleep(SLEEP_TIME);

        // milli-second sleep
        sleep(SLEEP_TIME);

        buf[count] = sensor_read(SENSOR);

        // sensor_read() returns 0, if the device is not available yet.
        // if (buf[count] == 0) continue;

        if (++count == LENGTH) {
            serial_send(PORT, LENGTH * sizeof(uint16_t), buf);
            count = 0;
        }
        led1_toggle();
    }
    return 0;
}
```

The sensor reading frequency can be determined by modifying `FREQ`. One of the three types of sensors (`LIGHT`, `TEMPERATURE`, and `HUMIDITY`) can be defined. The sensing values will be sent to PC through serial communication.

Receiving program on PC (utility/sensor_test/sensor_test.c)

```
#include <stdio.h>
#include <serial.h>

// serial port
#define PORT 1
#define LENGTH 15

uint16_t buf[LENGTH];

void test()
{
    int i;

    while (1) {
        if (serial_recv(PORT, LENGTH * sizeof(uint16_t), (uint8_t*)buf)
            == 0) continue;
        for (i = 0; i < LENGTH; i++) {
            printf("%04u ", buf[i]);
        }
        printf("\n-----\n");
    }
}

int main (int argc, char* argv[])
{
    int ret = -1;

    // argument check
    if (argc < 2)
        return -1;

    // initialize serial driver
    if (serial_open(argv[1], NULL) < 0)
        return -1;

    test();
}
```

```
// close serial port
serial_close();
return 0;
}
```

9

Miscellaneous System Calls

Obtaining local address

```
uint16_t LOCAL_ADDR;
```

The address value of current mote. This value is decided when kernel is loaded. If you want to get the address of current mote, you can use this identifier

```
uint16_t id = LOCAL_ADDR;
leds_set(id, LEDS_NORMAL);
```

An example of showing your mote's address through LEDs when it is 0 to 7.

Sleeps

```
void sleep(uint32_t count);
```

Make current thread sleep as long as the count value. Unit of the count value is millisecond (=1/1,000 sec)

```
void usleep(uint32_t count);
```

Make current thread sleep as long as the count value. Unit of the count value is microsecond (=1/1,000,000 sec)

Getting current local time

```
void get_localtime(uint32_t* time);
```

Obtains current mote's local time. Local time means the time expired since the mote has been started or reset. Unit of local time is millisecond.

Controlling LEDs

```
void led1_on();
void led1_off();
void led2_on();
void led2_off();
void led3_on();
void led3_off();
```

Turn on or turn off LED1 ~ LED3

```
void leds_set(uint8_t var, uint16_t mode);
```

Sets or unsets the 8bit 'var' of the LED. The lower 3 bits represent each LED. There are 4 types of modes, which are:

LEDS_NORMAL	Turns on/off LED according to 'var'.
LEDS_ON	Turns on LED according to 'var'.
LEDS_OFF	Turns off LED according to 'var'.
LEDS_TOGGLE	Toggles LED according to 'var'.

```
uint8_t leds_get();
```

Returns the LED's which are currently turned on. The lower 3 bits represent each LED.

Obtaining battery information

```
uint16_t get_battery();
```

Returns the current voltage of batteries. The return value is in unit of mV and between 2000mV and 3300mV.

10

Detecting Application Errors

Since hardware resources for sensor mote is limited and does not provide MMU and privileged mode, developers should consider the fact that applications could access the unauthorized kernel memory area. RETOS supports the dynamic error checking service that prevents applications from accessing the unauthorized area during its operations. RETOS also analyzes the source code and detects some errors during compile time.

10.1 Static Error Detection

Some errors can be detected during compile time. RETOS provides a static checker for this kind of errors.

```
#include <syscall.h>

int main()
{
    void (*func)(void) = (void*)0x1000;
    func();
    return 0;
}
```

The sample code makes a function pointer and accesses somewhere in the memory of the system. RETOS compiler analyzes the application's memory area and detects whether the sample application access incorrect memory area. Then compiler reports error and its location.

```
[retos/apps/safety_pointer/] $> make
scheck-msp430: 8: cannot find the function (maybe, immediate
address:4096)
    4004:      b0 12 00 10      call    #4096          ;#0x1000

[retos/apps/safety_pointer/] $>
```

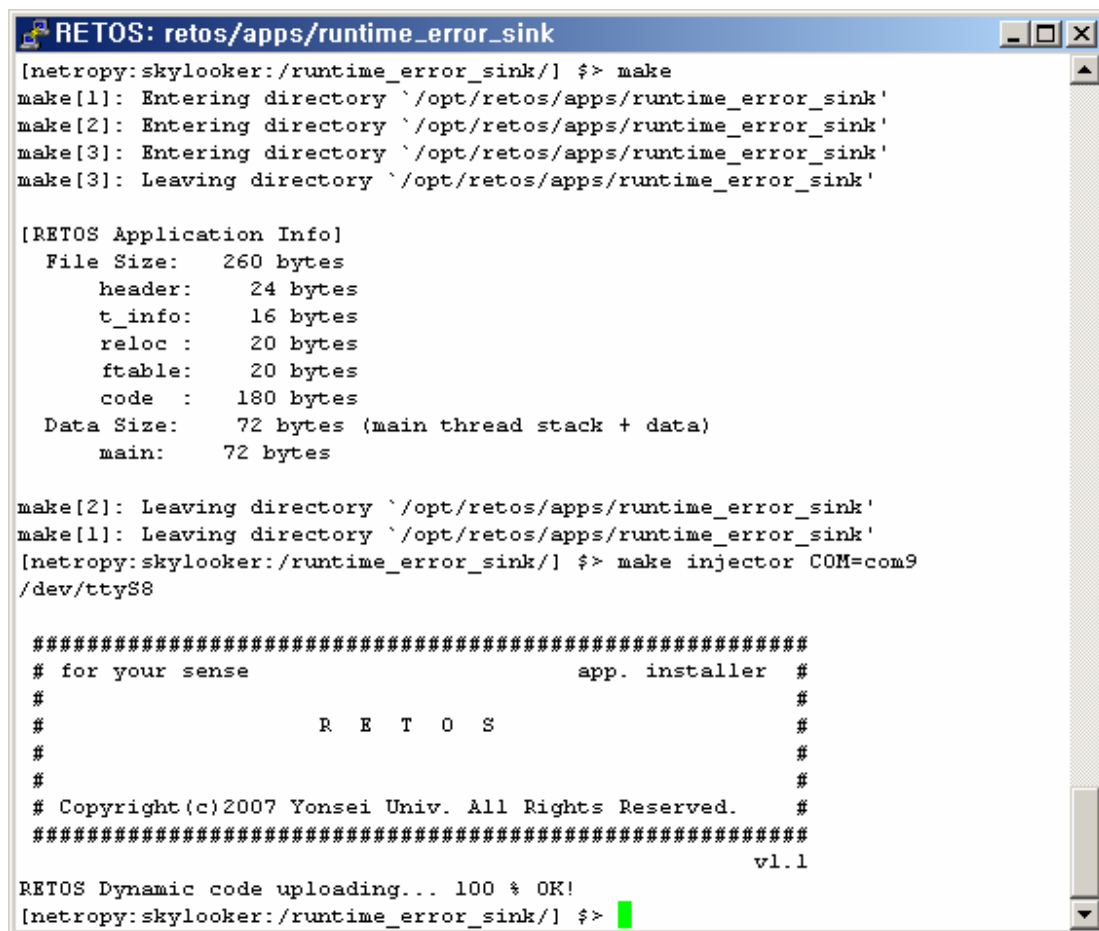
This information can be used for code debugging.

RETOS can detect and report following errors.

- calling recursive calls
- accessing unauthorized memory space
- running code on a memory space which doesn't belong to the current application
- accessing hardware area
- using limited functions

10.2 Dynamic Error Detection

An application may commit a malfunction during its running time. When application's malfunction is detected by kernel, the system shuts down the application and report the error through the network. An example is shown below.



```

[netropy:skylooker:/runtime_error_sink/] $> make
make[1]: Entering directory `/opt/retos/apps/runtime_error_sink'
make[2]: Entering directory `/opt/retos/apps/runtime_error_sink'
make[3]: Entering directory `/opt/retos/apps/runtime_error_sink'
make[3]: Leaving directory `/opt/retos/apps/runtime_error_sink'

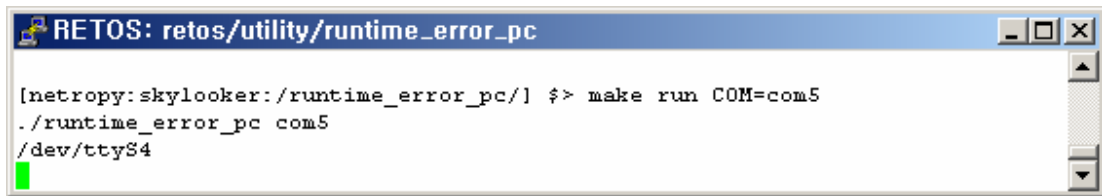
[RETOS Application Info]
  File Size:  260 bytes
    header:   24 bytes
    t_info:   16 bytes
    reloc :   20 bytes
    ftable:   20 bytes
    code  :   180 bytes
  Data Size:  72 bytes (main thread stack + data)
    main:     72 bytes

make[2]: Leaving directory `/opt/retos/apps/runtime_error_sink'
make[1]: Leaving directory `/opt/retos/apps/runtime_error_sink'
[netropy:skylooker:/runtime_error_sink/] $> make injector COM=com9
/dev/ttyS8

#####
# for your sense                               app. installer #
#                                                                 #
#               R E T O S                               #
#                                                                 #
#                                                                 #
# Copyright(c)2007 Yonsei Univ. All Rights Reserved.        #
#####
                                                                    vl.1
RETOS Dynamic code uploading... 100 % OK!
[netropy:skylooker:/runtime_error_sink/] $>

```

To detect run-time errors of the application, a host PC with a connected sink mote is required. (Install **app/runtime_error_sink** application into this mote.) The sink mote receives and reports the error reported from the remote sensor mote.



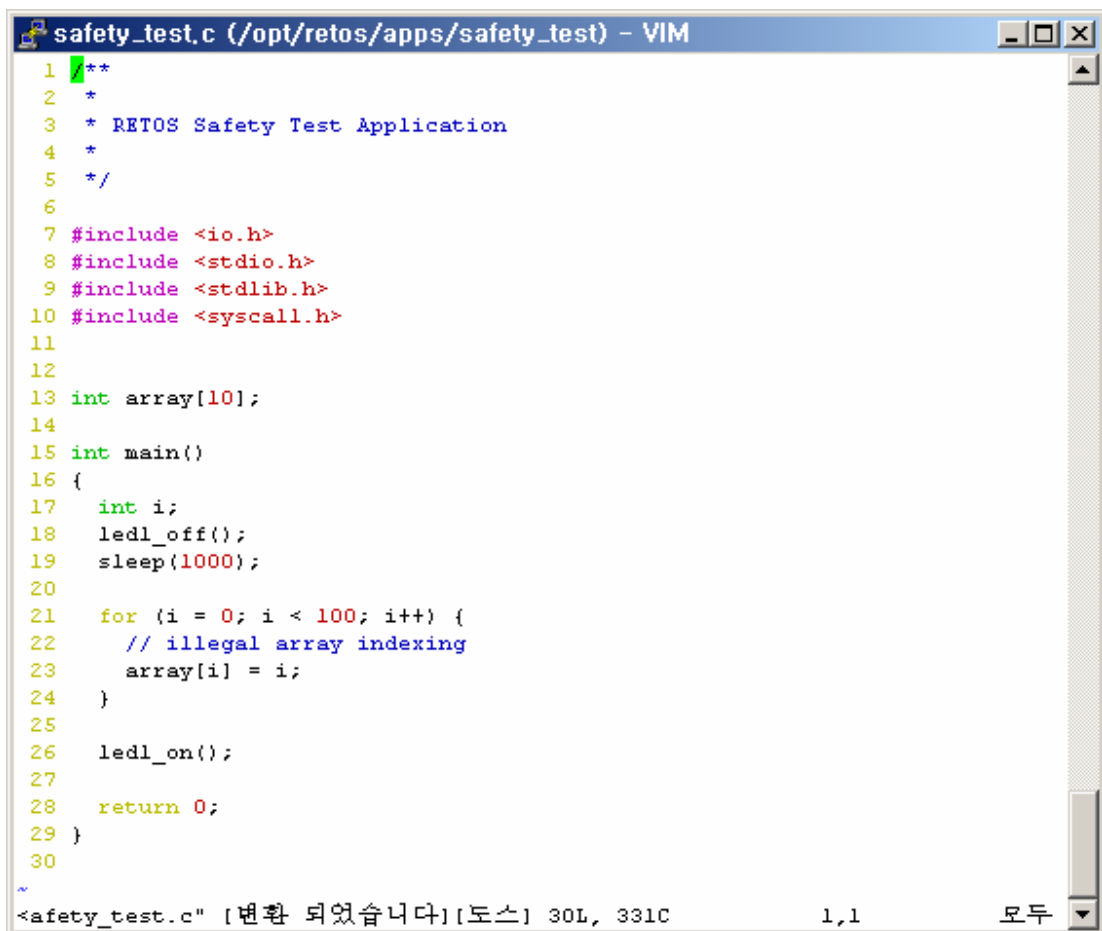
```

RETOS: retos/utility/runtime_error_pc

[netropy:skylooker:/runtime_error_pc/] $> make run COM=com5
./runtime_error_pc com5
/dev/ttyS4

```

Run the report application (**utility/runtime_error_pc**) to receive and display the error reported on the host PC.

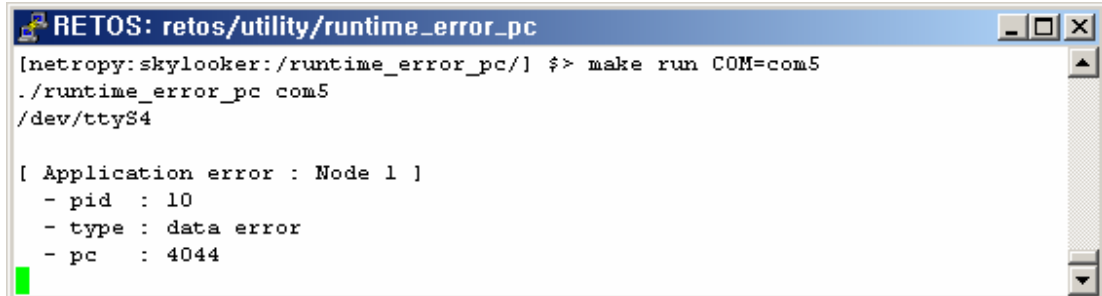


```

safety_test.c (/opt/retos/apps/safety_test) - VIM
1  /**
2   *
3   * RETOS Safety Test Application
4   *
5   */
6
7  #include <io.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <syscall.h>
11
12
13 int array[10];
14
15 int main()
16 {
17     int i;
18     ledl_off();
19     sleep(1000);
20
21     for (i = 0; i < 100; i++) {
22         // illegal array indexing
23         array[i] = i;
24     }
25
26     ledl_on();
27
28     return 0;
29 }
30
~
<safety_test.c" [변환 되었습니다][도스] 30L, 331C      1,1      모두

```

This code (**apps/safety_test**) is an example that tries to access the memory space which is not allocated to this application. The application has 20 bytes of an array of integers in bss/data area. The application tries to write 100 integers into the array and that area is out of boundary. Install this application into another sensor mote.



```

RETOS: retos/utility/runtime_error_pc
[netropy:skylooker:/runtime_error_pc/] $> make run COM=com5
./runtime_error_pc com5
/dev/ttyS4

[ Application error : Node 1 ]
- pid : 10
- type : data error
- pc : 4044

```

After the installation, this application (**apps/safety_test**) tries to access the unallocated area and it is detected by the kernel. The system reports error through the network and this error report is received by sink mote and the host PC (**runtime_error_sink** and **runtime_error_pc**). The example shows that the error occurred from the application with ID number 10. The program counter indicates specific location of assembly code which is generated by a code dump tool.

10.3 Restrictions

The RETOS application error detection has following restrictions.

Limitation	Description
Dynamic Memory Allocation	RETOS does not provide dynamic memory allocation for application. Using malloc() and free() functions are detected by static error checker of RETOS in compile time.
Recursive Call	RETOS does not provide recursive calls.
Function Pointer	Function pointer is not allowed to use, for stack analysis of the application.
ETC	Using sprintf() is not allowed.

11

Module Programming

11.1 Overview

RETOS provides kernel module feature to make system functionalities and application independent, and modify the kernel dynamically. Module is part of the kernel and it has two characteristics, event-driven and run-to-completion. Kernel module is efficient for implementing system functionality used by application. Kernel module can be linked to the kernel and installed, or dynamically loaded into already installed kernel. Please refer to chapter 2, Running RETOS Kernel and Application Programs, for installing and loading kernel module.

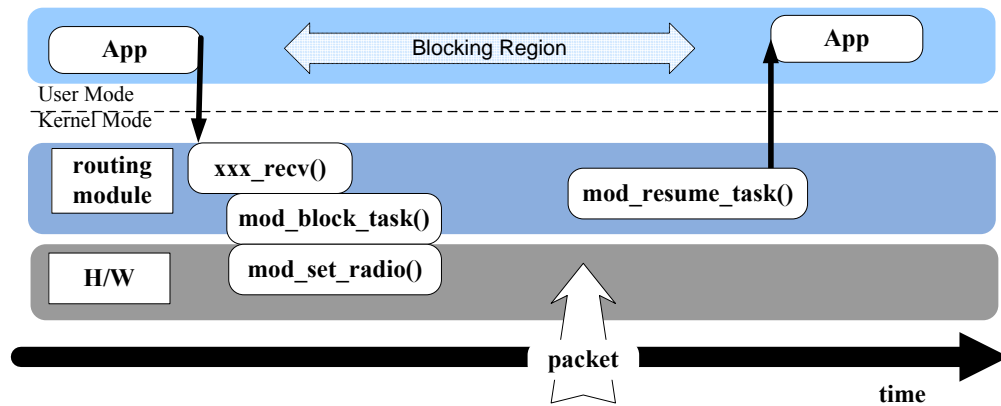
11.2 Controlling Application Context

Kernel module can control the context of an application program. That is, it can stop or resume application thread execution. This is especially useful for implementing blocking I/O functions.

- `uint16_t mod_block_task();`
- `uint16_t mod_resume_task(uint16_t task_id, uint16_t ret);`

`mod_block_task()` is a function for blocking user thread. It will put user thread that is currently running into the ready queue of the scheduler and return the ID of the thread.

`mod_resume_task()` puts the specified thread into run-queue so the thread can resume its execution. The parameter 'ret' is provided for a return value of the system call function when the kernel re-enters into user-mode.



The above figure is the example of blocking I/O using module functions. This example implements kernel module such as routing module and provides `xxx_rcv()`. An application might execute `xxx_rcv()` and will be blocked. On an arrival of certain packet, the application will continue its execution. For this process, `xxx_rcv()` of the kernel module calls `mod_block_task()` to block the user thread and stores the ID, then receives radio packets by calling `mod_set_radio()`. After receiving of certain packet, it will wake up the sleeping thread by passing the ID of it to `mod_resume_task()`.

11.3 Managing Function Table

A function table which enables communication between kernel and kernel module is provided. Functions that is provided by a kernel module should be registered in the function table in order for other kernel module or application to use the functions.

- `int mod_register_fn(uint8_t fnid, void* fn);`
- `int mod_unregister_fn(uint8_t fnid, void* fn);`
- `uint16_t call_fn(uint16_t fnid, void* arg);`

You can use `mod_register_fn()` to register your kernel module functions. You should provide unique function ID and the address of the function. `mod_unregister_fn()` removes already registered function. `call_fn()` executes registered functions. You can pass the unique ID of the function that you want to execute. A function that is to be registered on function table has following form of function prototype.

■ `uint16_t function (void * arg);`

`call_fn()` returns -1 if it can not find the function or returns the return value of the callee function. Therefore, it is recommended not to use -1 as a return value of the callee function.

11.4 Managing System Events

Kernel module follows the event-driven execution model as the RETOS kernel does. Kernel is not executed in multi-threaded manner so it provides various events to support programming concurrency. Kernel module can set a timer to make certain period of delay in the code or support periodic execution of the code.

■ `int mod_set_timer(void* fn, uint32_t count);`
 ■ `int mod_clr_timer(uint16_t id, void* fn);`

`mod_set_timer()` executes function *fn* after expiration of a certain amount of time, *count*. This timer is not repeated so you should call this function recursively at the end of the function *fn*. `mod_set_timer()` returns the ID of the timer on successful timer initialization, or returns -1 on failure. The unit of *count* is milliseconds.

`mod_clr_timer()` cancels a timer. You should pass the timer ID obtained by `mod_set_timer()` function and also the address of the function.

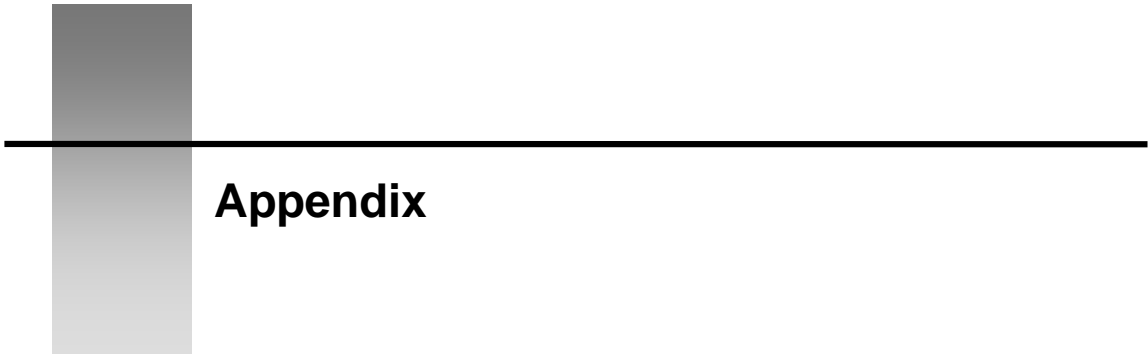
Kernel module should use following functions for RF and serial communications.

- `int mod_set_radio (void* fn, uint8_t port, uint8_t length, uint8_t* buf);`
- `int mod_set_serial (void* fn, uint8_t port, uint8_t length, uint8_t* buf);`
- `int mod_set_sensor (void* fn, uint8_t n_dev);`

- `int mod_clr_radio(uint16_t id, void* fn);`
- `int mod_clr_serial(uint16_t id, void* fn);`
- `int mod_clr_sensor(uint16_t id, void* fn);`

- `void fn_radio_callback();`
- `void fn_serial_callback (uint16_t length);`
- `void fn_sensor_callback(uint16_t data);`

`mod_set_xxx()` functions make a request for reading from radio, serial, and sensor, respectively. On the completion of data receiving or reading, call-back function *fn* will be executed. On the failure of such request, they will return -1, otherwise return the ID of the request. This ID can be used to cancel the request. `mod_clr_xxx()` functions cancel the request using request ID obtained by `mod_set_xxx()` functions and the address of the call-back function. Only call-back functions of serial and sensor have parameters, which is the length of receiving data and sensor value. Sending data through RF or serial uses the same `xxx_send()` and `xxx_sendto()`, which are also used by application programs.



Appendix

Appendix	65
Appendix A System Calls	66
Appendix B Module Calls	76
Appendix C The PNS Module	84
Appendix D RMTTool: The RETOS Monitoring Tool	86
Appendix E ETA (Elapsed Time Arrival) Field	96
Appendix F Code Dissemination	98
Appendix G Hardware Supports	100
Appendix H RETOS Paper List	101

A**Appendix****System Calls**

The RETOS kernel provides system level functionalities for correct systematic operations. Hence, the applications are able to use various system calls, such as radio and serial communications. The system calls provided by RETOS are as follows.

System functions

uint16_t LOCAL_ADDR;

Represents the ID of the current Mote. The ID is given at the time of kernel injection, and can be acquired from applications by referring to this identifier.

void sleep(uint32_t count);

Puts a thread into sleep for given period, 'count'. milliseconds. (=1/1,000 sec)

void usleep(uint32_t count);

Puts a thread to sleep for given period, 'count'. microseconds. (=1/1,000,000 sec)

void get_localtime(uint32_t* time);

Returns the time elapsed since the startup time of the mote, which is represented in milliseconds.

void led1_on(); void led1_off();

void led2_on(); void led2_off();

void led3_on(); void led3_off();

Turns on/off the LEDs of the mote (LED1, LED2 LED3)

void leds_set(uint8_t var, uint16_t mode);

Sets or unsets the 8bit ‘var’ of the LED. The lower 3 bits represent each LED. There are 4 types of modes, which are:

LEDS_NORMAL	Turns on/off LED according to ‘var’.
LEDS_ON	Turns on LED according to ‘var’.
LEDS_OFF	Turns off LED according to ‘var’.
LEDS_TOGGLE	Toggles LED according to ‘var’.

uint8_t leds_get();

Returns the LED’s which are currently turned on. The lower 3 bits represent each LED.

uint16_t get_battery();

Returns the current voltage of batteries. The return value is in unit of mV and between 2000mV and 3300mV.

Serial communications

int serial_send(uint8_t port, uint16_t length, uint8_t *data);

Transmits data through the serial in non-blocking fashion. If transmission successful, it returns the length of the transmitted data packet, otherwise returns 0.

port the port number through which the application sends data, which is within a range of 0~255. (*here it is not the PC serial port, such as MS Windows COM1 or Linux’s /dev/ttyS0)

length the size of the data packet to be sent.

data the pointer to the data packet.

The maximum size of data is up to 30bytes. If the data is larger than 30bytes, the first 30bytes are sent and it returns the length of the remaining data. Hence, large data needs to be segmented and transmitted in chunks of 30bytes.

int serial_sendto(uint8_t port, uint16_t length, uint8_t *data);

It is a non-ACK based serial_send function, which provides faster sequential transmission of serial communication. (the transmission speed depends on the Host PC)

The usage and return value is equivalent to serial_send.

int serial_recv(uint8_t port, uint16_t length, uint8_t *data);

Receiving data through serial. Blocking fashion. If data is successfully received, it returns the size of the received data, otherwise 0 is returned. The data is received through the ‘port’ into the pointer to buffer, ‘data’, which has a size of ‘length’

Radio communications

```
int radio_send(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);  
int radio_sendto(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);
```

Transmits data through RF. radio_send is a TCP like reliable transmission function which operates in blocking-fashion. radio_sendto is also blocking function which blocks the current thread until the RF chip finishes transmitting data.

If data is successfully transmitted, the size of the payload is returned, otherwise returns -1.

addr every mote has its unique address. Here ‘addr’ is the node’s address that will receive the data. Because addr is 2bytes variable, it has a range of 0~65534. If the data is to be broadcasted the addr should be BROADCAST_ADDR, which is 0xFFFF.

port this is the port number through which the sender transmits. Hence the receiver needs to listen to this port to correctly receive the data.

length the size of the payload. The maximum payload size is 32 bytes.

msg the pointer to the rf_pkt structure. The actual data will be stored in the data field. Additional information can be retrieved from the rf_pkt structure.
(eg. Msg->length, which is the length of the payload etc.)

```
int radio_recv(uint8_t port, rf_pkt * msg);
```

Receives an RF packet sent from another mote. The thread is blocked until the data is received. If the receive request is successful 0 is returned, otherwise -1 is returned (e.g. Incorrect port number, etc.)

port the port number through which the data is to be received.

msg the rf_pkt buffer which the data is to be stored. The payload is stored in the msg’s data field, according to the size of msg->length.

```
void set_sink_loc(uint16_t addr, uint16_t pos_x, uint16_t pos_y, uint16_t pos_z);
```

Set the address and the position of the sink, which is needed to use the send_sink function.

Requires NSL (The Network Support Layer)

addr the address of the sink
 pos_x the x-coordination of the sink
 pos_y the y-coordination of the sink
 pos_z the z-coordination of the sink

```
int send_sink(uint16_t addr, uint8_t port, uint8_t length, rf_pkt* msg,  

              uint8_t module, uint8_t param);
```

Uses a given routing algorithm to send data to the sink. (1-to sink) Before using send_sink, the routing module is needed. Currently, PNS (Parametric Neighbor Selector) is provided as the routing algorithm. Requires NSL (The Network Support Layer)

addr the address of the sink. The sink information must be registered using the 'set_sink_loc' system call.
 port the transmission port used by the application.
 length the size of the payload.
 msg the radio_pkt struct. The payload will be stored in this packet's data field.
 module this is the name of the routing module used to send the packet to the sink (eg. PNS_MODULE).
 param the parameter of the module

```
int send_nbr(uint8_t port, uint8_t length, rf_pkt* msg);
```

Sends data to motes which are currently registered as a neighbor. (1-to-neighbor) The parameters are equivalent to radio_send(). Requires NSL (The Network Support Layer)

```
int send_net(uint8_t port, uint8_t length, rf_pkt* msg);
```

Sends data to the entire network. (1-to-all, flooding) The parameters are equivalent to radio_send(). Requires NSL (The Network Support Layer) (Not implemented yet)

```
int set_rfpower(uint8_t value);
```

Setting the RF transmission power. The value should be between 1 and 31.

```
uint8_t get_rfpower();
```

Returns the current value of RF transmission power.

```
uint8_t set_channel(uint8_t ch);
```

Setting the current radio channel used by RF transmission. The argument ‘ch’ should be between 11 and 26.

NSL API functions

uint16_t get_update_interval()

Returns the updating interval of the neighbor table

void set_update_interval(uint16_t attr)

Sets the neighbor table update interval with a unit of seconds. The default interval is 10 seconds

uint16_t get_nb_cnt()

Returns the number of nodes in the neighbor table.

uint16_t get_nb_id(uint8_t node_seq)

Returns the identifier of the node specified in the argument, ‘node_seq’. ‘node_seq’ is an index of the neighbor table.

uint16_t get_nb_lqi(uint8_t node_seq)

Returns the LQI(Link Quality Indicator) of the specified node.

int8_t get_nb_rssi(uint8_t node_seq)

Returns the RSSI(Received Signal Strength Indicator) of the specified node.

uint16_t get_nb_battery(uint8_t node_seq)

Returns the amount of the battery left on the specified node.

uint16_t get_nb_pos_x(uint8_t node_seq)

Returns the x-coordinate of the specified node.

uint16_t get_nb_pos_y(uint8_t node_seq)

Returns the y-coordinate of the specified node.

uint16_t get_local_pos_x()

Returns the x-coordinate of the node itself.

```
uint16_t get_local_pos_y()
```

Returns the y-coordinate of the node itself.

```
void set_local_pos_x(uint16_t attr)
```

Sets the x-coordinate of the node itself.

```
void set_local_pos_y(uint16_t attr)
```

Sets the y-coordinate of the node itself.

Pthread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine)(void *), void *arg);
```

Creates a thread. Returns 0 if successful, otherwise -1 is returned.

`thread` the identifier of the thread.

`attr` may be used to set specific Pthread characteristics, however, RETOS pthread library does not adopt this functionality and simply passes NULL.

`start_routine` the pointer to the thread function which is to be executed at startup. However, do not use the pointer to the function, but simply pass the function name.

`arg` the parameter which the `start_routine` accepts.

```
int pthread_join (pthread_t th, void **thread_return);
```

This function blocks the current thread until the 'th' thread terminates. If the target thread terminates, or the target thread does not exist, it returns.

`th` the target thread's identifier which is to be waited for.

`thread_return` used to store the return value of the waiting thread. If no return value is needed, simply pass NULL as a parameter.

```
int pthread_cancel(pthread_t thread)
```

Sends a kill signal to a specific thread. Since, there is no function to configure the cancel state of a thread (`pthread_setcancelstate`), the thread which receives a `pthread_cancel` signal always terminates.

Synchronization

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr *attr);
```

Initializes a mutex object. attr is used to set specific modes of the mutex provided by the Pthread library. However, RETOS does not support such options, hence simply pass NULL as parameter.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Requests a mutex lock. If the mutex is already locked by another thread, it waits (blocked) until the lock is released.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Requests a mutex lock. If the mutex is not locked it returns 0, otherwise -1 is returned.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Releases the mutex if locked.

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_cond_attr *attr);
```

Initializes the conditional variable. attr in Pthread is used to set specific options on cond. However RETOS does not provide such option, hence simply pass NULL as parameter.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Sends signal to the conditional variable. If a thread is waiting for the condition signal, it wakes up on the reception of the conditional signal. If multiple threads are waiting, only one thread wakes up. If there is no thread waiting, the cond signal is ignored.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Broadcasts the cond signal to all threads that are currently waiting for it.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Block current thread on the condition variable, cond. Before blocking, it will unlock the mutex and acquire the mutex lock again when this function is about to finish due to the reception of condition variable signal.

Scheduling

int sched_getscheduler(uint8_t pid);

This acquires the current scheduler policy of a specific thread. pid is the process's identifier. The pid can be retrieved through the getpid() system call.

If successfully retrieved, the function returns one of the following, SCHED_RR, SCHED_FIFO, SCHED_OTHER. Returns -1 if the thread with the given the pid cannot be found.

int sched_getparam(uint8_t pid, struct sched_param *param);

A function which gets the priority of current thread. The pid is the process's identifier. The pid can be retrieved through the getpid() system call.

'param' is the buffer where the priority information is to be stored. If successful 0 is returned, otherwise it returns -1.

int sched_setscheduler (uint8_t pid, int policy, const struct sched_param *param);

This function sets the policy and the priority of the specific thread. Returns 0 if successfully set, otherwise returns -1.

'pid' is the process's identifier. The pid can be retrieved through the getpid() system call. 'policy' can be set as one of the followings, SCHED_RR, SCHED_FIFO, SCHED_OTHER. Any other parameter is not allowed and would return -1.

The 'param' is set to a struct sched_param type. The structure of sched_param is as follows.

Priority is within the range of 0~9. SCHED_FIFO and SCHED_RR can have a priority within 1~9, SCHED_OTHER is always 0. Here, the SCHED_OTHER priority value does not reflect the same with other static priority values. (The SCHED_OTHER priority is assigned by the kernel)

After creating a struct sched_param variable (e.g. struct sched_param a), the priority of it can be set (eg. a.sched_priority = 5;).

int sched_setparam(uint8_t pid, struct sched_param *param);

This function configures the priority of a specific thread. The parameters are equivalent to sched_setscheduler.

Sensing

uint16_t sensor_read (uint16_t type);

This function acquires the readings of a sensor. ‘type’ is the type of sensor to read the data from. It returns the sensor data. If it was unsuccessful to read data from the sensor, 0 is returned. According to which sensor is accessed, the delay to read the data from the sensor may take a few or several tens of milliseconds.

Available Sensor Types:

- 1 - LIGHT (TmoteSky, Telos)
- 2 - HUMIDITY (TmoteSky, Telos)
- 3 - TEMPERATURE (TmoteSky, Telos)

B**Appendix****Module Calls****System functions**

```
uint16_t LOCAL_ADDR;
```

The address value of current mote. This value is decided when kernel is loaded. If you want to get the address of current mote, you can use this identifier.

```
uint16_t mid      ID of the installed kernel module
```

```
void get_localtime(uint32_t* time);
```

Obtains current mote's local time. Local time means the time expired since the mote has been started or reset. Unit of local time is millisecond.

```
void led1_on(); void led1_off();
```

```
void led2_on(); void led2_off();
```

```
void led3_on(); void led3_off();
```

Turn on or turn off LED1 ~ LED3

```
void leds_set(uint8_t var,uint16_t mode);
```

Sets or unsets the 8bit 'var' of the LED. The lower 3 bits represent each LED. There are 4 types of modes, which are:

```
LEDS_NORMAL      Turns on/off LED according to 'var'.
```

```
LEDS_ON           Turns on LED according to 'var'.
```

```
LEDS_OFF          Turns off LED according to 'var'.
```

```
LEDS_TOGGLE      Toggles LED according to 'var'.
```

```
uint8_t leds_get();
```

Returns the LED's which are currently turned on. The lower 3 bits represent each LED.

```
int mod_set_timer(void* fn, uint32_t count);
```

Setting a timer. After the amount of time specified in count, call-back function fn will be executed. The unit of count is milliseconds.

```
int mod_clr_timer(uint16_t id, void* fn);
```

Cancel the timer. The ID returned from mod_set_timer() and the address of the call-back function should be passed as a parameters.

```
int mod_register_fn(uint8_t fnid, void* fn);  
uint16_t fn(void* arg);
```

Register a function on the shared function table. A function ID and pointer to the function should be passed as parameters. It returns positive integer on success, otherwise returns negative integer. Shared function should have a void * type argument and return uint16_t.

```
int mod_unregister_fn(uint8_t fnid, void* fn);
```

Remove a function from the shared function table. The function ID and pointer should be passed as parameters. It returns 0 on success, or returns negative integer on failure if it can not find the function in the table.

```
uint16_t call_fn(uint16_t fnid, void* arg);
```

Execute a function on the shared function table. The function ID and an argument to the callee function should be passed as parameters. It returns 0 if it can not find the function, otherwise returns the return value of the callee function.

Serial communications

```
int serial_send(uint8_t port, uint16_t length, uint8_t *data);
```

Transmit data to host PC through serial communication. Non-blocking function. Return transmitted data length when it success. Return 0 when it fails to transmit.

port Port number of a destination application. Support 0 to 255 (*the ‘port’ does not mean address of serial port number such as COM1 of MS Windows or /dev/ttyS0 of Linux)

length Length of the transmitting data.

data Pointer to the transmitting data.

`serial_send` can transmit up to 30bytes of data at once. If a user tries to send data more than 30bytes, `serial_send` would transmit 30bytes of data only and return 30.

```
int serial_sendto(uint8_t port,uint16_t length, uint8_t *data);
```

Does not wait for the ACK frame of host PC after transmitting data. Appropriate for high data rate communication

Usage and return values are same as those of `serial_send`.

```
int mod_set_serial(void* fn, uint8_t port, uint8_t length, uint8_t* data);  
void fn (uint16_t length);
```

Receive data from serial. Call-back function `fn` will be executed when the reception of data is completed. It returns -1 if it can not receive data, otherwise returns the request ID which is used to cancel the request.

The parameter `port` is the port number to be used, and `length` is the size of the buffer for the receiving data. The pointer to the buffer should be passed in `data`.

The parameter for the call-back function is the length of received data.

```
int mod_clr_serial(uint16_t id, void* fn);
```

Cancel reading sensor. The request ID and the pointer to call-back function should be passed as parameters.

Radio communications

```
int radio_send(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);  
int radio_sendto(uint16_t addr, uint8_t port, uint8_t length, rf_pkt * msg);
```

These functions are used for data communication with radio frequency.

`radio_send`, `radio_sendto` are blocking functions. `radio_send` is used for reliable transmission as in TCP of common network stack. The function blocks until it receives acknowledgment from destination. `radio_sendto` blocks until the radio transceiver finishes the transmission.

When the communication succeeds, the functions return the length of payload that have transmitted or received. Otherwise they return -1.

`addr` unique address of destination mote. Use `BROADCAST_ADDR` to broadcast a message.

`port` port number specified by each application. the sender and receiver communicate

when their port numbers are identical

length length of communicating data. The data length is less than or equal to 32bytes

msg address of rf_pkt structure to transmit. Transmitted data should be stored in data field of the structure. While receiving a radio packet, various fields refer to important information including actual data.

e.g.) msg->length is the length of received data

```
int mod_set_radio(void* fn, uint8_t port, uint8_t length, uint8_t* buf);
```

```
void fn ();
```

Receive data from other nodes using RF communication. The call-back function fn will be called on the completion of receiving data. If the request has been failed, it will return -1, otherwise returns request ID. This ID might be used for canceling the request.

Port number to be used should be passed in port. The address of radio packet structure which will store the received radio packet should be passed in buf. The size of received packet is the length field of the rf_pkt structure, and actual data will be stored in data field.

```
int mod_clr_radio(uint16_t id, void* fn);
```

Cancel request to receive radio packets. The request ID and call-back function should be passed as parameters.

```
void set_sink_loc(uint16_t addr, uint16_t pos_x, uint16_t pos_y, uint16_t pos_z);
```

Set up the node address and position of sink node. These information are used for send_sink. Requires NSL (The Network Support Layer)

addr address of the sink

pos_x x-coordination of the sink

pos_y y-coordination of the sink

pos_z z-coordination of the sink

```
int send_sink(uint16_t addr, uint8_t port, uint8_t length, rf_pkt* msg,  
              uint8_t module, uint8_t param);
```

Send data from node to sink through routing To use send_sink, routing modules must be installed. Currently, RETOS supports PNS (Parametric Neighbor Selector) module for routing. Requires NSL (The Network Support Layer)

addr address of a sink node. The sink has to be registered through 'set_sink_loc'.

port port number specified by each application.
length length of the data.
msg address of received rf_pkt structure.
module routing module identifier. Choose routing module for transmission. e.g.)
 PNS_MODULE
param pass parameters required for routing modules.

```
int send_nbr(uint8_t port, uint8_t length, rf_pkt* msg);
```

Transmit radio to motes recorded in Neighbor Table (1-to-neighbor) Usage and parameters are same as radio_send. Requires NSL (The Network Support Layer)

```
int send_net(uint8_t port, uint8_t length, rf_pkt* msg );
```

Transmit radio to all motes in the network (1-to-all, flooding) Usage and parameters are same as radio_send(). Requires NSL (The Network Support Layer) (Not Implemented Yet)

```
int set_rfpower (uint8_t value);
```

Setting the RF transmission power. The value should be between 1 and 31.

```
uint8_t get_rfpower ();
```

Returns the current value of RF transmission power.

```
uint8_t set_channel(uint8_t ch);
```

Setting the current radio channel used by RF transmission. The argument 'ch' should be between 11 and 26.

NSL API functions

```
uint16_t get_update_interval()
```

Returns the updating interval of the neighbor table

```
void set_update_interval(uint16_t attr)
```

Sets the neighbor table update interval with a unit of seconds. The default interval is 10 seconds

```
uint16_t get_nb_cnt()
```


Returns the number of nodes in the neighbor table.

uint16_t get_nb_id(uint8_t node_seq)

Returns the identifier of the node specified in the argument, 'node_seq'. 'node_seq' is an index of the neighbor table.

uint16_t get_nb_lqi(uint8_t node_seq)

Returns the LQI(Link Quality Indicator) of the specified node.

int8_t get_nb_rssi(uint8_t node_seq)

Returns the RSSI(Received Signal Strength Indicator) of the specified node.

uint16_t get_nb_battery(uint8_t node_seq)

Returns the amount of the battery left on the specified node.

uint16_t get_nb_pos_x(uint8_t node_seq)

Returns the x-coordinate of the specified node.

uint16_t get_nb_pos_y(uint8_t node_seq)

Returns the y-coordinate of the specified node.

uint16_t get_local_pos_x()

Returns the x-coordinate of the node itself.

uint16_t get_local_pos_y()

Returns the y-coordinate of the node itself.

void set_local_pos_x(uint16_t attr)

Sets the x-coordinate of the node itself.

void set_local_pos_y(uint16_t attr)

Sets the y-coordinate of the node itself.

Scheduling

int sched_getscheduler(uint8_t pid)

A function which gets current scheduling policy. pid indicates process ID(thread ID), which can be obtained by getpid(). Returns one of SCHED_RR, SCHED_FIFO, and SCHED_OTHER when SUCCESS, otherwise returns -1. FAIL occurs when no corresponding pid exist.

int sched_getparam(uint8_t pid, struct sched_param *param);

A function which gets the priority of current thread. pid indicates process ID(thread ID), which can be obtained by getpid(). param indicates the address where the priority of current thread will be stored. Returns 0 when SUCCESS, otherwise returns -1.

int sched_setscheduler (uint8_t pid, int policy, const struct sched_param *param);

A function which sets the scheduling policy and priority. Returns 0 when SUCCESS, otherwise returns -1. pid indicates process ID(thread ID), which can be obtained by getpid(). policy can be SCHED_RR, SCHED_FIFO, or SCHED_OTHER. Other arguments will cause return of -1. param specifies the priority in sched_param structure.

The priority has range of 0~9. The priority of SCHED_FIFO and SCHED_RR can be between 1 and 9. The priority of SCHED_OTHER is always 0. In here, the priority means static priority, different from the dynamic priority in SCHED_OTHER. (The dynamic priority of SCHED_OTHER is defined by the kernel.)

int sched_setparam(uint8_t pid, struct sched_param *param);

A function which is similar to sched_setcheduler. The difference is that this function sets only the priority of a thread.

uint16_t mod_block_task();

Block user thread that is currently running. Return the ID of blocked thread.

uint16_t mod_resume_task(uint16_t task_id, uint16_t ret);

Wake up the blocked user thread. The thread ID and a return value for the waking up thread should be passed as parameters.

Sensing

```
int mod_set_sensor(void* fn, uint8_t n_dev);  
void fn (uint16_t data);
```

Read sensor data. Call-back function fn will be called on the completion of reading sensor. type is the type of sensor that will be read. Returns sensor data and it will have different values depending on the type of sensor and environment.

It returns -1 on the failure of sensor reading, otherwise the request ID will be returned. This ID might be used to cancel the request. Sensor data value will be passed in the argument, data.

Supported sensor types:

- 1 - LIGHT (TmoteSky, Telos)
- 2 - HUMIDITY (TmoteSky, Telos)
- 3 - TEMPERATURE (TmoteSky, Telos)

```
int mod_clr_sensor(uint16_t id, void* fn);
```

Cancel sensor reading. The request ID obtained by calling mod_set_sensor() and the call-back function pointer should be passed as parameters.

C**Appendix****The PNS Module**

PNS (Parametric Neighbor Selector)¹ module is the policy engine for node-to-sink routing protocol. The PNS module is a geometric routing algorithm. PNS supports run-time policy adjustment that arranges a routing path for each packet. PNS supports data-centric routing.

PNS Parameters

PNS sets up the network parameters which helps deciding the next node while forwarding a packet. RETOS supports the neighbor table which manages neighbor nodes in a single-hop distance. PNS supports 3 domains.

- **SPEED**: selects a node that resides nearest to the sink.
- **RELIABILITY**: selects a node that has the finest link quality
- **ENERGY**: selects a node that has the highest battery power

Users can configure these parameters for each network packets. For example, RELIABILITY domain should be set up for the reliable transmissions. Also, the domains may be used simultaneously. For example, both ENERGY and RELIABILITY could be set up together.

1) Y. Sung, H. Cha, "Parametric Routing for Wireless Sensor Networks," 2006 International Symposium on Ubiquitous Computing Systems (UCS 2006), Seoul, Korea, October 2006

PNS API

RETOS provides the PNS module at ‘modules/pns_mod,’ with an ID ‘PNS_MODULE.’ (For an installation, please refer to section 2.4.)

send_sink(1, RADIO_PORT, sizeof(int_msg), &msg, PNS_MODULE, SPEED|RELIABILITY);

send a packet to the sink node, considering SPEED and RELIABILITY domains (A full source code is in chapter 5.2)

get_parent(1, PNS_MODULE, ENERGY);

get the neighbor node that has the highest battery power for the energy-aware routing

The system should support a NSL for the PNS module.

D

Appendix

RMTool: The RETOS Monitoring Tool

RETOS provides a monitoring tool for analyzing if a deployed sensor network is correctly functioning. With every mote running the monitoring application, the overall network information is received at the base station which can be easily analyzed through a GUI at the HOST PC.

Configuring RMTool

First of all, every mote needs to be assigned with a unique node ID at kernel install time. One node is needed as a sink, the rest function as network nodes that periodically transmits network information to the sink placed at the Host PC. The applications and modules needed for RMTool is located under **apps/**, **modules/** and **utility/**. The detailed configurations is as follows

Mote - Sink Node (Base station)

```
cooper@inukj /opt/retos-v1.1.0/apps/rmtool_sink
$ ls
Makefile  mrouting_sink.c  mrouting_sink.h

cooper@inukj /opt/retos-v1.1.0/apps/rmtool_sink
$ make telosh
make[1]: Entering directory `/opt/retos-v1.1.0/apps/rmtool_sink'
make[2]: Entering directory `/opt/retos-v1.1.0/apps/rmtool_sink'
make[2]: Leaving directory `/opt/retos-v1.1.0/apps/rmtool_sink'

[RETOS Application Info]
  File Size: 788 bytes
    header: 24 bytes
    t_info: 16 bytes
    reloc : 60 bytes
    ftable: 40 bytes
    code : 648 bytes
  Data Size: 100 bytes (main thread stack + data)
    periodic_receiver: 74 bytes
    main: 98 bytes

make[1]: Leaving directory `/opt/retos-v1.1.0/apps/rmtool_sink'

cooper@inukj /opt/retos-v1.1.0/apps/rmtool_sink
$ make injector COM=com4
/dev/ttyS3
RETOS Dynamic code uploading... 100 % OK!
```

The sink node, which is connected with the Host PC for serial communication, should run the **apps/rmtool_sink/** application. The ID of the sink must be 1. The sink node and other nodes rely on the NSL, which are defined in the Makefile when compiling the kernel. After installing kernel to the sink node, compile and inject **apps/rmtool_sink**. Unlike the network nodes, the sink node does not need the routing module (**modules/pns**)

Mote - Network Nodes

The **apps/rmtool_router** application is needed to be injected into the network nodes. Afterwards, the routing module **modules/pns** is needed to be injected. The installation process is the same as the sink mote. The network nodes periodically manage their 1-hop neighbor nodes and try to sustain connectivity. The collected environmental data are sent to the sink node periodically.

Host PC

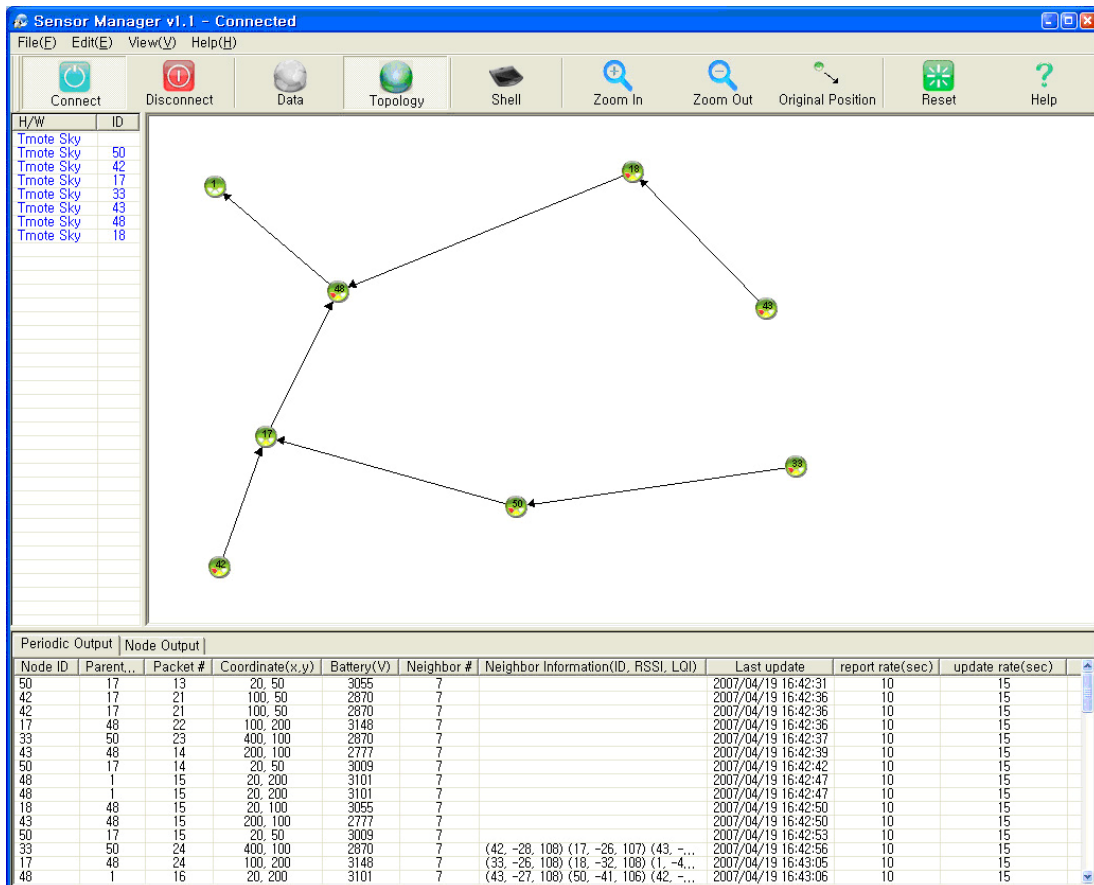
The incoming monitoring data is seen through the GUI at the Host PC by executing **utility/rmtool_win32/SensorManager.exe**. After loading the program, the serial COM port needs to be initiated by clicking on the 'Connect' button. If the connection between RMTTool and the sink mote is successful, the sink sends its sink information, which is displayed in the GUI. Otherwise, a disconnect message will be shown at the title of the program window. When sink is initialized and displayed in the panel, it starts to transmit network packets received from network nodes. All incoming data is shown in the display panel, which provides a monitoring environment of the deployed network. Specific nodes can be monitored and controlled through issuing command messages.

RMTTool usage guide

Sensor Manager Overview

RMTTool provides three display panels. The left panel lists the IDs and the hardware types of all nodes that are currently taking part in the network in sorted order based on ID. The center panel







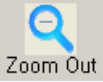

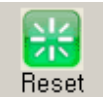
displays the topology of the deployed network. The location of each node is logical representation, which is given at the compile time of **apps/rmtool_router** application. General information of each node is provided in this panel, such as Node ID, LED status, Active status, neighbor table and last update time. The bottom panel displays the incoming data packets in text. It consists of two panels (Periodic Info, Node Info). The Periodic Info tab shows the continuous incoming data packets of all nodes. The Node Info tab shows the continuous incoming data packets of a specific node. The node of interest can be chosen by double clicking the entry at the Periodic Info tab. The diagram below shows a scenario of a currently active sensor network.








Main Menu

Name	Description
File (Ctrl+S)	Save the incoming data. The log file is named with the current time (e.g. SMlog_20070322165423).
Edit->SelectComport	Provide the user with a dialog for choosing the COM port to connect to the sink mote.
View->Show Routing Path	Displays the routing path of each node to the sink.
View->Show Neighbors	Displays the relation between a node and its neighbor nodes.
Help	Shows version history of RMTool

Icons

Icon / Name		Description
 Connect	Connect	Connect RMTool with sink mote
 Disconnect	Disconnect	Disconnect with sink mote
 Data	Data Table	Show overall data table
 Topology	Topology View	Show network topology
 Shell	Shell	Show shell command window for issuing command queries
 Zoom In	Zoom In	Zoom into the network topology
 Zoom Out	Zoom Out	Zoom out of the network topology
 Original Position	Original Position	Place all nodes to their original logical location, if they were manually moved.
 Reset	Reset	Initializes RMTool, the sink and all nodes are removed.

Main View

Icon	Description
	Each node is represented with one of the three different colors, Green, orange and red. Here, the green status implies that the particular node has a packet loss of less than 3 sequential packets. Also, each node has three led status displays, which can show the current status of a mote. Up to three leds are currently available.
	Nodes with more than 3 but less than 6 sequential packet losses are displayed as orange.
	Nodes with more than 6 sequential packet losses are displayed as red, which also implies disconnection from the network for some reason.
	The solid arrow line shows the forwarding routing path of the node.
	The relation between neighbors and a node is shown as dotted arrow lines. Also the RSSI (Radio Signal Strength Indicator) relation is shown as the color of the head of the edge. If a neighbor and the node of interest have an RSSI larger than -60dB, the arrow will be painted with green color. If the RSSI is lower than -60dB and larger than -80dB, the arrow will be painted orange. RSSI lower than -80dB will be painted in red. Hence, a green arrow implies that the RF signals is strong, which suggests a good link.

Data Message Output View

All incoming data packets at the base station are simply shown as text. If there is a node of interest, the data in the past can be seen in the Node Info tab, the Periodic Info tab is shown as default.

Periodic Info Tab

Here, the reported information from all nodes is shown, which are Local ID, parent ID, packet count, location, battery residue, neighbor count, neighbor table (including its relation), and the

timestamp. The information being shown in this tab is obtained by collecting periodic network management messages from all motes.

Node Info Tab

Here, the information of a node of interest can be separately seen. Simply double click on the entry of the node of interest in the periodic info tab.

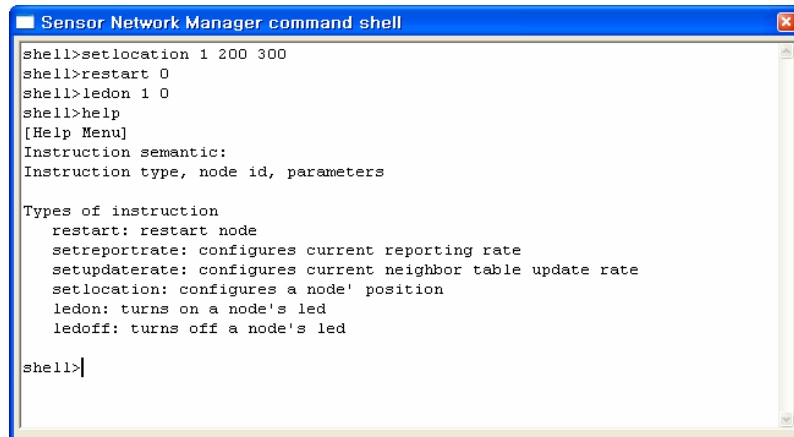
Log File

All the incoming information is continuously logged on a file with a format of the current date, year/hour/minute/second. (eg. SMlog_20070528223758.txt) The information that is being logged is identical of what is shown in the Periodic Info Tab. An example snapshot is shown below. The log file is automatically saved when RMTool is closed or can be saved manually by pressing CTRL + S.

```
<Sensor>
[Src ID] [Parent ID] [Pkt seq] [Coord X, Coord Y] [Battery(V)] [No. of Nbr] [Nbr Table(ID,RSS)] [Update Time] [Report Rate(sec)] [Update Rate(sec)]
2 1 133 150, 150 2.9 5 (5, 13)(1, -57)(3, -17)(7, -12)(11, -30) 2007/05/28 22:38:02 5 12
5 1 80 250, 50 3.1 5 (2, -57)(1, -57)(3, -17)(7, -10)(11, -5) 2007/05/28 22:38:04 5 12
3 5 104 50, 50 2.9 5 (2, -10)(1, -13)(5, -8)(7, -7)(11, -46) 2007/05/28 22:38:05 5 12
7 1 35 10, 150 2.6 5 (2, -16)(1, -26)(3, -17)(5, -10)(11, -26) 2007/05/28 22:38:05 5 12
11 5 15 10, 10 3.0 5 (2, -8)(1, -47)(3, -41)(7, -15)(5, -20) 2007/05/28 22:38:06 5 12
2 1 134 50, 150 2.8 5 (5, -2)(1, -36)(3, -17)(7, -30)(11, -15) 2007/05/28 22:38:06 5 12
5 1 118 20, 50 2.9 5 (2, -49)(1, -50)(3, -1)(7, -13)(11, -11) 2007/05/28 22:38:07 5 12
</Sensor>
```

Shell Usage

The shell provides a command shell like interface where nodes can be controlled. As shown below, the shell is text-based and sends commands to the sink, which then disseminates it to the network.



```

Sensor Network Manager command shell
shell>setlocation 1 200 300
shell>restart 0
shell>ledon 1 0
shell>help
[Help Menu]
Instruction semantic:
Instruction type, node id, parameters

Types of instruction
restart: restart node
setreportrate: configures current reporting rate
setupdaterate: configures current neighbor table update rate
setlocation: configures a node's position
ledon: turns on a node's led
ledoff: turns off a node's led

shell>

```

The command input syntax is shown below:

- Instruction type, node id, parameters

The instruction type is shown in the 'System Command' table below. The 'node id' is the target node to which the command will be issued. If the command is broadcast, pass 0 as the 'node id'. A quick help command is available to see the available commands.

System Command

Command	Opcode	Parameter	Description
restart	0001	int nid	Restarts the RMon application. ‘nid’ is the ID of the node to restart the RMon nid 0 would broadcast the command to the entire network. This applies to the all instructions below.
setreportrate	0111	int nid, int report_rate	Configures the periodic reporting rate of node ‘nid’. ‘report_rate’ is the new periodic reporting rate, which is in seconds. The reporting takes an input of a range of 0~255. 0 implies to stop reporting.
setupdaterate	1000	int nid, int update_rate	This configures the neighbor updating rate. ‘update_rate’ is the new update rate in seconds. The update rate takes an input of the range of 0~255.
setlocation	1001	int nid, int x_pos, int y_pos	Set the location of node ‘nid’. x_pos is the new x-coordination y_pos is the new y-coordination
ledon	1010	int nid, int led_num	Turns on the led of node ‘nid’. ‘led_num’ is the flag value for controlling mote’s leds. 0 turns on all leds, 1 turns on led1, 2 turns on led2, 3 turns on led3.
ledoff	1011	int nid, int led_num	Turns off the led of node ‘nid’. ‘led_num’ is the flag value for controlling mote’s leds. 0 turns off all leds, 1 turns off led1, 2 turns off led2, 3 turns off led3.
help	1100	-	Displays a simple menu of the available command instructions.

Example - Shell Command Issuing

Some instruction examples using shell commands.

```
: Restart RMon application of node 7
> restart 7

: Restart all nodes RMon application
> restart 0

: Turn on led 2 of node 7
> ledon 7 2

: Turn off all leds of all nodes
> ledoff 0 0

: Set new location of node 29, to x=250, y=170
> setlocation 29 250 170

: Set reporting rate to 20 sec of node 30
> setreportrate 30 20

: Set the neighbor table update rate of node 31 to 100 sec
> setupdaterate 31 100
```

E**Appendix****ETA (Elapsed Time Arrival) Field**

ETA (Elapsed Time Arrival)¹ is used by reactive time synchronization technique. The ETA field is time-stamped with the elapsed time since an event occurs until a radio packet is actually sent. Then receiver side can extract the ETA field from the packet header. The receiver can estimate the real occurrence time of the event using the receiver's local time and the ETA field, since it is assumed that packet delivery time is trivial. Sample codes are as follows.

A sample code that sends packets with the ETA information

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define PORT 5

typedef struct st_payload{
    uint32_t data;
}st_payload;

void send_msg()
{
    rf_pkt msg;
    uint32_t event_time;

    get_localtime_tick(&event_time);
    msg.time = event_time; // ETA packet
    radio_sendto(BROADCAST_ADDR, PORT, sizeof(st_payload), &msg);
}
```

1) Kusy, B. et al. Elapsed Time on Arrival: A simple, versatile, and scalable primitive for canonical time synchronization services, accepted for publication in Int. J. of Ad Hoc and Ubiquitous Computing, 2005

Sender sends a packet to the receiver with occurrence time of the event. The MAC layer handles this field to indicate a delay until the actual emission of the packet.

Receiver code

```
#include <io.h>
#include <stdio.h>
#include <syscall.h>

#define PORT 5

typedef struct st_payload{
    uint32_t data;
} st_payload;

void* recv_thread(void *arg)
{
    rf_pkt msg;
    uint32_t event_time;
    while(1){
        radio_rcv(PORT, &rcvd_msg);
        event_time = rcvd_msg.time; // ETA packet
        /* do something */
    }
}
```

The MAC layer of the receiver extracts the ETA field from the packet header and transforms its value into local time. Then the ETA field indicates the actual occurrence time of the sender's event. The receiver may reference this ETA field in a view of local event.

F

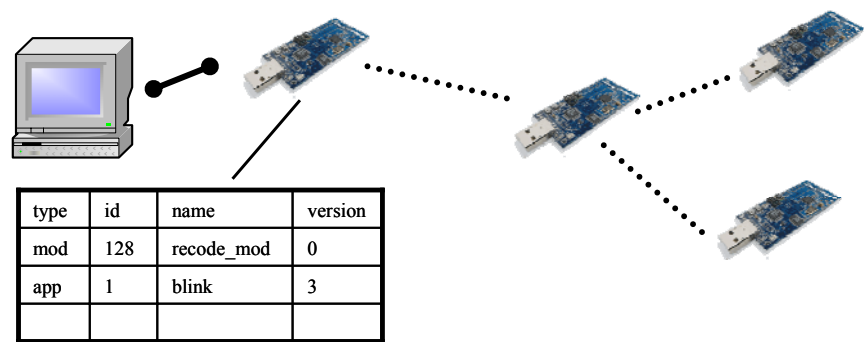
Appendix

Code Dissemination

RETOS provides a code dissemination service for installing kernel modules and applications to the remote sensor motes.

Network Topology for Code Dissemination

RETOS provides a code dissemination module. To test the code dissemination functionality, every sensor motes needs the code dissemination module. The module may be linked to the kernel image or is loaded into the system while the system is running. The sensor network topology is as follows.



The sink mote is connected to a host PC through COM port (USB). The RETOS kernel and the code dissemination module (recode_mod) are installed on every sensor mote including the sink mote. Each mote in the network shares the application list and the version information together. Since the host PC installs an application into the sink mote, every mote in the network communicates with each other and installs the application. In that way, every sensor mote has

the latest version of the application.

Example of Code Dissemination

The code dissemination may be started by installing a newer version of the application on any sensor nodes in the network.

```
[preparing an application that blinks a blue LED]
retos/apps/blink/] $> make injector COM=com1 ID=1 VER=1
[Start] - Every node starts to install the application
[Complete] - Every blue LED starts to blink

[modifying the application to blink green LED]
retos/apps/blink/] $> make injector COM=com1 ID=1 VER=2
[Start] - Every node start to install the application with version
number 2
[Complete] - Every green LED starts to blink
```

Every kernel module and application in the file system is handled by the code dissemination module, except for codes with version number zero and the code dissemination module itself. And the installed applications can not be removed since the algorithm try to reinstall the application from neighbor nodes.

[Tip]

To remove the application from the networks a dummy application can be installed instead. The dummy application does not do anything. The dummy application should have the same ID and a higher version number. Though the file system wastes some spaces for the dummy, still it is useful to remove the application from the network temporally.

G**Appendix****Hardware Supports**

Vendor	Hardware	MCU	RAM	Flash	Sensors
Moteiv	Telos Rev.A	TI MSP 430 (8MHz)	2kB	60kB, 1MB (external)	Humidity, Light and Temperature
	Telos Rev.B	TI MSP 430 (8MHz)	10kB	48kB.	Humidity, Light and Temperature
	Tmote Sky			1MB (external)	Temperature

H**Appendix****RETOS Papers**

- H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, C. Yoon, "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks", The Sixth International Conference on Information Processing in Sensor Networks (IPSN 2007), Cambridge, Massachusetts, USA, April 2007.
- S. Choi, H. Cha, S. Cho, "A SoC-based Sensor Node: Evaluation of RETOS-enabled CC2430," Fourth Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON 2007), San Diego, USA, June 2007.
- H. Kim, H. Cha, "Multithreading Optimization Techniques for Sensor Network Operating Systems", The 4th European conference on Wireless Sensor Networks (EWSN 2007), Delft, Netherlands, January 2007.
- I. Jung, H. Cha, "RMTool: Component-Based Network Management System for Wireless Sensor Networks," 2007 IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, January 2007.
- Y. Sung, H. Cha, "Parametric Routing for Wireless Sensor Networks," 2006 International Symposium on Ubiquitous Computing Systems (UCS 2006), Seoul, Korea, October 2006.
- H. Shin, H. Cha, "Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes," The 2nd International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2006), Hong Kong, China, December 2006.
- S. Choi, H. Cha, "Application-Centric Networking Framework for Wireless Sensor Nodes," The 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services (MOBIQUITOUS) 2006, San Jose, California, July 2006.
- H. Kim, H. Cha, "Towards a Resilient Operating System for Wireless Sensor Networks", The 2006 USENIX Annual Technical Conference (USENIX'06), Boston, Massachusetts, June 2006.