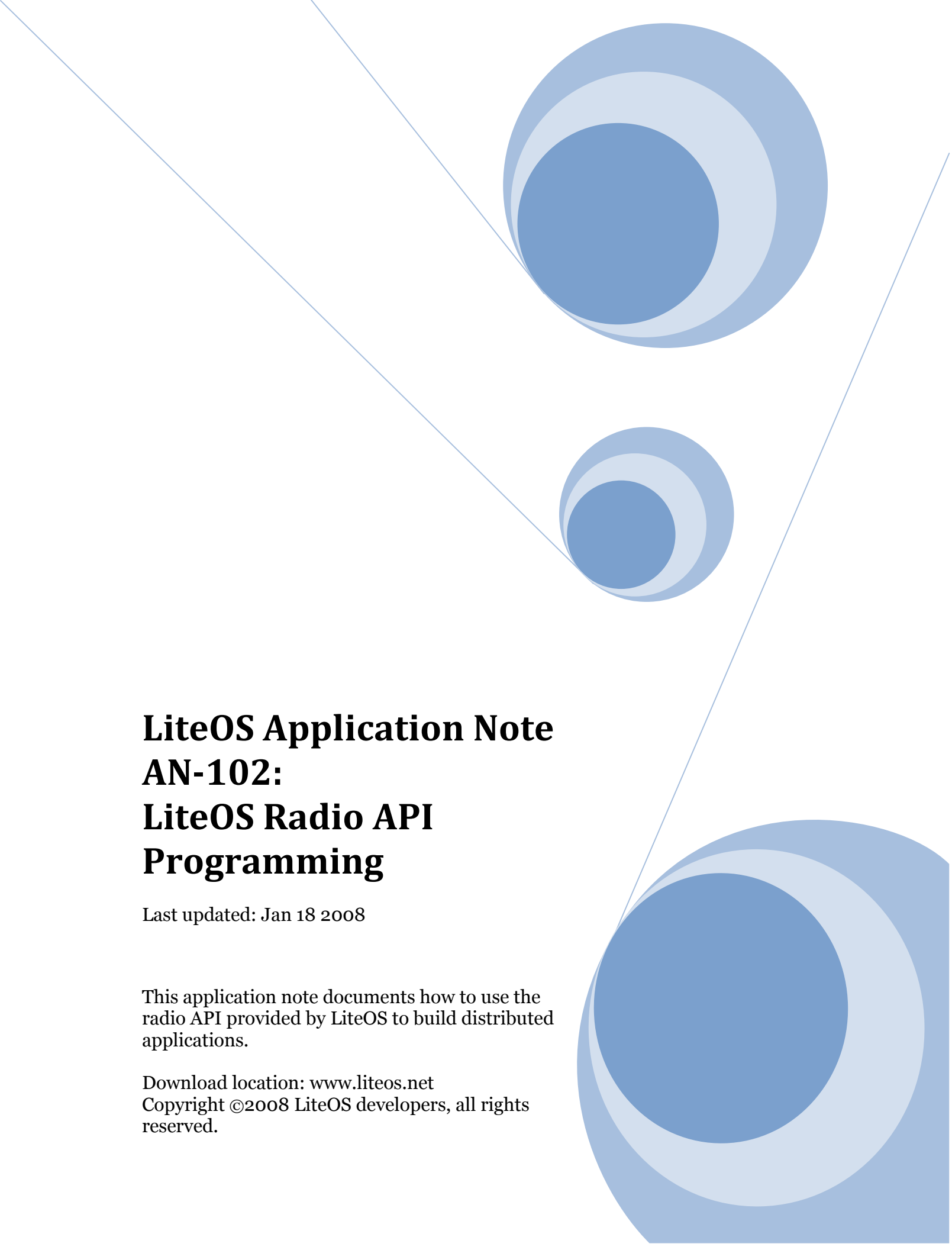


LiteOS Application Note AN-102: LiteOS Radio API Programming

Last updated: Jan 18 2008

This application note documents how to use the radio API provided by LiteOS to build distributed applications.

Download location: www.liteos.net

Purpose

This application note documents how to use to the radio API in LiteOS to write multi-hop applications, as well as how to write PC applications to communicate with sensor nodes.

Introduction to LiteOS Radio API

LiteOS 0.2 provides a comprehensive set of radio functions to send and receive messages. The detailed function definitions are provided in the *radio.h* file in the LiteOS library. We illustrate how to uses these API through two examples: one simple *HelloWorld* program where one node repeatedly sends out radio messages, and one more complicated multi-hop geographic forwarding program that implements distributed geographic routing.

The "Hello, World" Program

The "Hello, World" program is located in the HelloWorld directory of the sample applications. The complete program is very short, with a main() body as follows:

```
int main()
{
  while (1)
        {
        radioSend_string("Hello, world!\n");
        greenToggle();
        sleepThread(100);
        }
        return 0;
}
```

In this example, we send out the string "Hello, world!" repeatedly, using the radioSend_string function. This function parses the string, finds out its length, and broadcasts it over the radio with default port (port 1) and

destination (broadcast address 0xffff). The green LED will toggle when the messages are broadcast.

More generally, the radio API provides several send functions, including radioSend_string() for sending out strings terminating with '\0', radioSend_uint16() for sending out integers, and a generic radioSend() function to send out a chunk of data with a certain length. Both the radioSend_string() and radioSend_uint16() are implemented through the radioSend() function, which further interacts with the kernel through several system calls. Once the radioSend() function is called, it yields the current thread, until the message has been sent out by the kernel. To coordinate the sending operations of multiple threads, the sending function uses mutex to control thread behavior: if a thread cannot obtain the mutex, it has to wait by yielding the CPU, until another thread finishes using the mutex and wakes it up.

Introduction to the Geographic Forwarding Algorithm

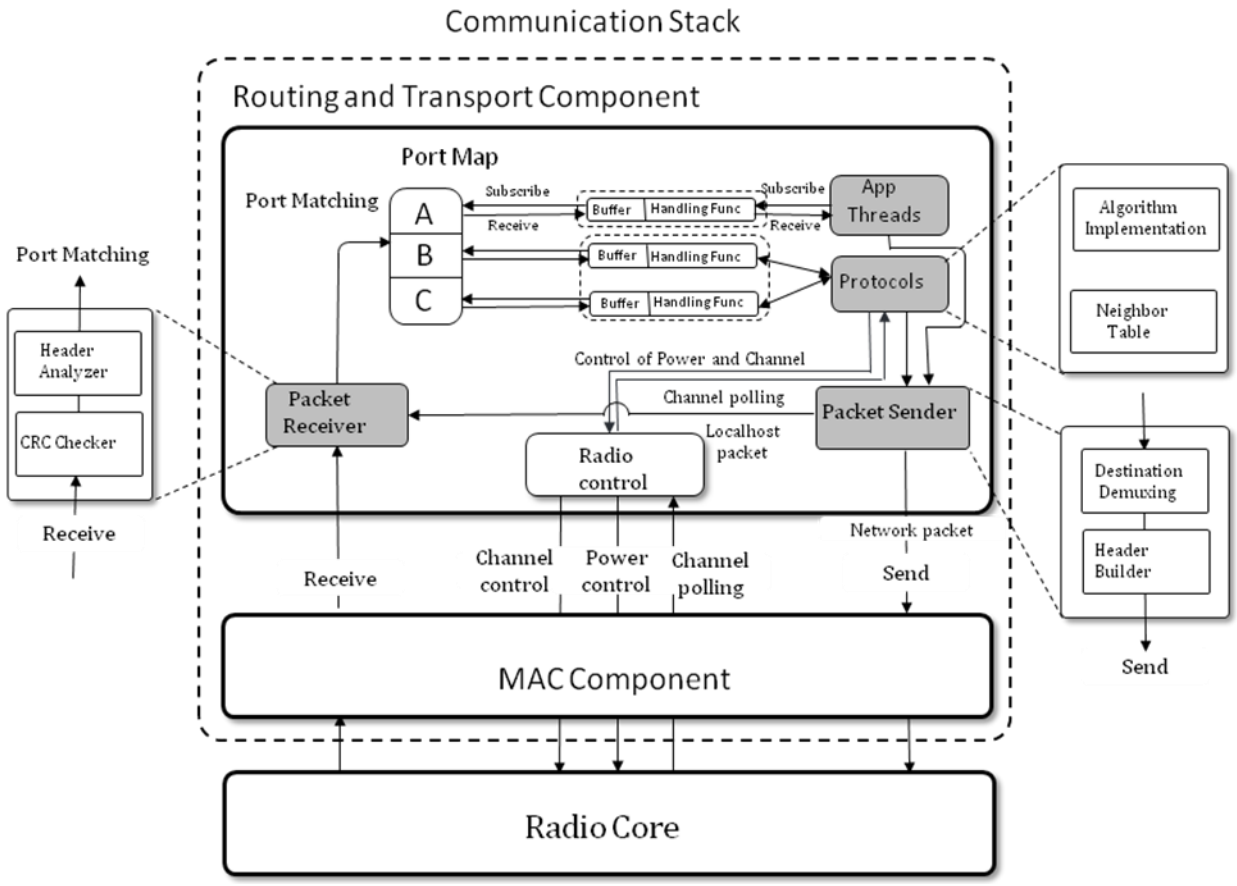
We now move to a more complicated example, where we implement the geographic forwarding routing algorithm. As an address-based algorithm, geographic routing is particularly attractive in the context of sensor networks for the following reasons. Most early address-based routing protocols maintain per-node routing state that grows as a function of either the network size or the number of active destinations. This is the case with DSR and AODV, both of which were initially envisioned for ad-hoc networks. This state growth causes problems for sensor networks, where the storage space for individual nodes is severely constrained. Geographic routing solves this problem by only requiring a constant amount of per-node state: namely, the node's immediate neighborhood. Therefore, geographic routing protocols can be easily implemented under extreme resource constraints such as those of sensor nodes.

The basic mechanism of geographic forwarding works as follows. Each node obtains a geographic location either through GPS or through some localization algorithm. There have been more than 100 different localization algorithms proposed in the sensor network literature for different scenarios. Once nodes obtain such location information, they deliver packets to the destination node via multiple hops as follows. At each hop, the node delivers the packet to the next node in its neighbor table that is the nearest to the destination. If nodes deployed are relatively dense, such a simple solution will usually succeed to deliver packets to the destination node.

Implementing the geographic routing in LiteOS

The implementation approach we use is a direct reflection of the design choices of LiteOS, namely, every application serves as a thread that is scheduled by the kernel. Hence, the geographic routing service is also implemented as a thread. Normally, this thread is sleeping, until a packet needs to be delivered. When such a packet arrives, either from another thread locally or from another node, this routing thread wakes up, looks up the neighbor table for the best next node, and then transmits this packet. The advantage of this implementation structure is that it is extremely flexible: different routing protocols could be switched in and out at application runtime, and provide a highly modular architecture for communication.

More formally, the logic diagram showing the generic communication layer in LiteOS is shown in the following figure. Note that our implementation does not have a MAC layer in this example. A MAC protocol could be implemented in a similar way to the routing protocol, which is also implemented as a flexible thread.



Implementing details

In the implementation of the protocol, the *protocol.c* file, you will find the main() function as the entry address for the whole protocol. The protocol then calls two functions, registerDataPacket() and registerNeighborhoodPacket(), to register with the kernel packet dispatcher that this thread is waken up every time a new data packet or a neighborhood maintenance packet is received. The main protocol then enters a loop, where it repeatedly calls the sleepThread() function to put the current thread into sleep mode, and waits for incoming packets. If incoming packets are received while this thread is sleeping, it will wake up, search through the neighbor table for the best next-hop node based on the packet header information, and deliver the packet to the next hop. It is also possible that, however, that the current node is

the destination node itself. Under such circumstances, the routing thread will deliver the packet locally to the waiting application thread.

To maintain neighbor table and keep topology information, the routing thread also wakes up periodically to broadcast neighborhood maintenance packets (heartbeat messages), so that each node in its neighborhood will be informed with topology changes.

Using the routing protocol

In this section, we explain how an application thread uses the routing protocol. In the implementation of the routing protocol, it listens on two ports, Port 10 and Port 11. The port 10 is used to receive data packets, while the port 11 is used to receive neighborhood broadcast packets. Therefore, to use the routing protocol, a user application should send packets to port 10, with a destination address set to be the local node, 0, meaning that the packet should be delivered first to the thread that listens on port 10 on the local node.

Once the packet is delivered to the routing protocol thread, the first several bytes of the packet are parsed to get the final destination node ID and its geographic address. In our implementation, the first several bytes are organized as follows.

Dest Node Location	Multi-hop source node	Multi-hop dest node	Dest node receiving port

As an example, suppose we want to send a packet from the node 1 located at (1,1) to node 5 located at (5,5) via multiple hops, we could write the following piece of code to achieve this purpose:

```
uint8_t msg[12];
msg[0] = 5;                          //destx
msg[1] =  5;                         //desty;
msg[2] = 1;                          //source node;
msg[3] = 5;                          //dest node
msg[4] = 80;
strcnpy(&msg[5], content, 5);//copy the data content
radioSend(10, 0, 16, msg);      //deliver the packet
```

After the packet is delivered, the routing thread will deliver the packet hop by hop until it arrives at the destination.

Conclusions

In this application note, we have briefly outlined the steps to talk to the radio system API, and how to implement a routing protocol based on this API. The detailed definitions of the API, as well as examples, are available in the library source code in the LiteOS distribution.