WiDom API

Nuno Pereira

February 13, 2007

Contents

Chapter 1

A dominance Protocol for Wireless Networks

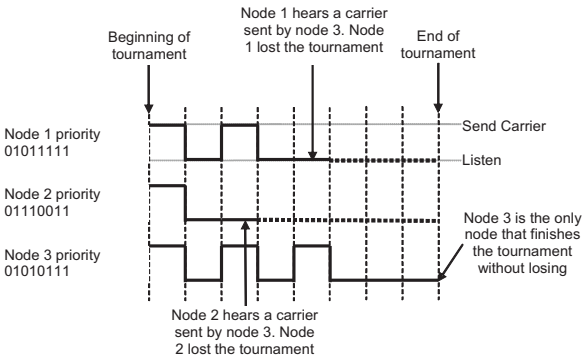

Figure 1.1: The MAC protocol tournament.

This MAC protocol is inspired in wired Dominance/Binary Countdown protocols [1, 2]. In this MAC protocol when messages contend for the channel, a conflict resolution phase is performed. In our protocol, this conflict resolution phase is named tournament. During the tournament, nodes transmit the priority of the message contending for the medium bit-by-bit. But, wireless transceivers can hardly be transmitting and receiving at the same time. Thus, when the transmitted bit is dominant there no need to sense the medium, whereas, when the bit to transmit is recessive, nothing has to be effectively sent, instead only the medium state has to be sensed.

The MAC protocol performs a tournament as depicted in Figure 1.1. The nodes start by agreeing on an instant when the tournament starts. Then nodes transmit the priority bits starting with the most significant bit. A bit is assigned a time interval. If a node contends with a dominant bit ("0"), then a carrier wave is transmitted in this time interval; if the node contends with a recessive bit ("1"), it transmits nothing but listens. At the beginning of the tournament, all nodes have the potential to win, but if a node contends with a recessive bit and perceives a dominant bit then it withdraws from the tournament and cannot win. If a node has lost the tournament then it continues to listen in order to know the priority of the winner. When a node finishes sending all priority bits without hearing a dominant bit, then it has won the tournament and clearly knows the priority of the winner. Hence, lower numbers represent higher priorities. This is similar to the CAN bus [1], but nodes in a CAN network are usually assigned unique priorities; that assumption is not made here.

In this protocol, a bit of the tournament is different from a data bit. Each bit in the tournament has a fixed duration of time which is considerably longer than a data bit. But, when a node wins the access to the medium, it may transmit at the full bit rate allowed by the specific radio transceiver.

Several versions of widom where implemented. Please see `http://www.nanork.org:8000/nano-RK/wiki/WiDom` for more information.

1.1 Using WiDom

You must start by including widom and define transmit and receive buffers.

```
#include <widom.h>
#include <widom_rf.h>


RF_TX_INFO rfTxInfo;
RF_RX_INFO rfRxInfo;
uint8_t tx_buf[RF_MAX_PAYLOAD_SIZE];
uint8_t rx_buf[RF_MAX_PAYLOAD_SIZE];
```

1.1.1 Initialization

Function `wd_init()` initializes the radio and protocol. Initializes CC2420 for radio communication via the basic RF library functions. Turns on the voltage regulator, resets the CC2420, turns on the crystal oscillator, writes all necessary registers and protocol addresses (for automatic address recognition). Note that the crystal oscillator will remain on (forever). It also sets up a periodic timer to poll for the synchronization packet and execute the protocol.

The pointer to `RF_RX_INFO` data structure is to be used during the first packet reception. The structure can be switched upon packet reception. The `channel` defines the RF channel to be used (11 = 2405 MHz to 26 = 2480 MHz), `panId` is 802.15.4 related and is just for initialization purposes of the radio, `myAddr` is the 16-bit short address which is used by this node.

```
rfRxInfo.pPayload = rx_buf;
rfRxInfo.max_length = RF_MAX_PAYLOAD_SIZE;
wd_init (&rfRxInfo, WD_CHANNEL, 0x2420, 0x1215);
```

1.1.2 Protocol Parameters

The protocol parameters can be set in `widom.h`. The number of priority bits used is defined by `NPRIOBITS`. The time out parameters (E, G, ETG) can be set in micro-seconds units. We also can define the maximum packet duration allowed.

The default parameters are based on the the following parameters: $\epsilon = 10^{-5}$, $CLK = 1/8 \times 10^{-6}$, $L = 5us$, $\alpha = 1us$, $TFCS = 256us$ and $SYNC\_ERROR = 50us$. This gives the timeouts $F = 320us$, $G = 58us$ and $ETG = 59us$.

```
// num of priorities
#define NPRIOBITS                       3 // number of priority bits

// timeouts interval constants (in us)
#define H_us                           320 // duration of a pulse of a carrier
#define G_us                           58 // "guarding" time between bits
```

```
#define ETG_us                          59 // "guarding" time after the tournament

#define CMSG_us WD_MAX_MSG_LEN_us         // max time to tx our messages
```

The parameters related to the synchronization are the the following. The amount of time before receiving the synchronization packet the node must start polling for it is defined by three parameters (WD_MAX_BLOCKING_TIME, WD_SYNC_PULSE_RX_SWX and WD_SYNC_WITH_MASTER_MAX_ERROR).

The synchronization period is automatically defined by WD_SYNC_PERIOD as a function of the duration of the tournament and message transmission. Due to the implementation, this period cannot be longer than 8 ms.

```
// these three parameters define how long we wake up before the synchronization
#define WD_MAX_BLOCKING_TIME_us 150 // maximum blocking widom may experience
#define WD_SYNC_PULSE_RX_SWX_us 256 // time to switch to rx and lock on to the preamble
#define WD_SYNC_WITH_MASTER_MAX_ERROR 400 // maximum difference with synch master
```

When the node is searching for the synchronization packet, it will do so every WD_SYNC_SEARCH_INT clock ticks and for a duration of WD_SYNC_MAX_WAIT_TIME us.

```
// interval between sync attempts (clock tks)
#define WD_SYNC_SEARCH_INT_clk_tks 16000

// timeout when waiting for sync packet
#define WD_SYNC_MAX_WAIT_TIME_us 1800
```

1.1.3 Sending a Packet

Sending a packet is done by either either function wd_tx_packet() or wd_tx_wait_packet(). While function wd_tx_packet() just signals to the protocol that a new packet was enqued, wd_wait_tx_packet() signals the new packet and blocks until it is sent. These two functions accept as arguments a RF_TX_INFO data structure and the priority of the message.

```
// fill packet buffer with data ...
for (i=0; i<DATA_LENGHT; i++) tx_buf[i]=data[i];
rfTxInfo.pPayload=tx_buf;
wd_wait_tx_packet(&rfTxInfo, MSG_PRIO);
```

or

```
// fill packet buffer with data ...
for (i=0; i<DATA_LENGHT; i++) tx_buf[i]=data[i];
rfTxInfo.pPayload=tx_buf;
wd_tx_packet(&rfTxInfo, MY_PRIO);
wd_wait_tx_packet(&rfTxInfo, MSG_PRIO);
```

1.1.4 Receiving a Packet

To receive a packet, just call wd_wait_until_rx_packet().

```
wd_wait_until_rx_packet();
// get data from packet ...
for (i=0; i<rfRxInfo.length; i++) putchar (rfRxInfo.pPayload[i]);
```

1.2 Example Test Application

The following is the source code for a simple demo application using widom. It has two periodic tasks. One sends packets, the other checks for the reception of packets.

```
#include <include.h>
#include <ulib.h>
#include <stdio.h>
#include <nrk.h>
#include <hal.h>
#include <basic_rf.h>
#include <avr/sleep.h>
#include <avr/wdt.h>
#include <avr/eeprom.h>
#include <nrk_error.h>
#include <nrk_timer.h>
#include <widom_rf.h>
#include <widom.h>

#define SND_TASK_PRIO 1
#define RCV_TASK_PRIO 2

#define SND_TASK_PERIOD_ms 1000
#define SND_TASK_RESERVE_ms SND_TASK_PERIOD_ms

#define RCV_TASK_PERIOD_ms 100
#define RCV_TASK_RESERVE_ms RCV_TASK_PERIOD_ms

NRK_STK StackTaskSnd[NRK_APP_STACKSIZE];
nrk_task_type TTaskSnd;
void TaskSnd(void);

NRK_STK StackTaskRcv[NRK_APP_STACKSIZE];
nrk_task_type TTaskRcv;
void TaskRcv(void);

void nrk_create_taskset();

RF_TX_INFO rfTxInfo;
RF_RX_INFO rfRxInfo;
uint8_t tx_buf[RF_MAX_PAYLOAD_SIZE];
uint8_t rx_buf[RF_MAX_PAYLOAD_SIZE];

//-------------------------------------------------------------------------------
//      void main (void)
//
//      DESCRIPTION:
//              Startup routine
//-------------------------------------------------------------------------------
int main (void)
{
        nrk_setup_ports();
```

```
        nrk_setup_uart (UART_BAUDRATE_115K2);

        printf( "WiDom Test Starting up... (ADDR=%d)\r\n", NODE_ADDR );

        nrk_init();

        nrk_led_set(0);
        nrk_led_clr(1);
        nrk_led_clr(2);
        nrk_led_clr(3);

        nrk_time_set(0,0);
        nrk_create_taskset ();
        nrk_start();

        return 0;
}

//------------------------------------------------------------------------------
//      void TaskSnd (void)
//
//      DESCRIPTION:
//              Task that periodically sends a packet
//------------------------------------------------------------------------------
void TaskSnd()
{

        rfRxInfo.pPayload = rx_buf;
        rfRxInfo.max_length = RF_MAX_PAYLOAD_SIZE;

        // widom init must be in a task
        wd_init (&rfRxInfo, WD_CHANNEL);

        while(1) {
                nrk_gpio_toggle(NRK_BLED);
                nrk_int_disable();
                rfTxInfo.pPayload=tx_buf;
                // put just two bytes of payload in the packet...
                tx_buf[0]=0xCB;
                tx_buf[1]=NODE_ADDR;
                rfTxInfo.length=2;
                nrk_int_enable();
                // NODE_ADDR is used for the priority of messages
                wd_tx_packet(&rfTxInfo, NODE_ADDR);
                //wd_wait_tx_packet(&rfTxInfo, MY_PRIO);
                nrk_wait_until_next_period();
        }
}

//------------------------------------------------------------------------------
//      void TaskSnd (void)
```

```
//
//      DESCRIPTION:
//              Task that periodically checks to receive a packet
//--------------------------------------------------------------------------------
void TaskRcv()
{
        while(1) {
                nrk_gpio_toggle(NRK_GLED);
                wd_wait_until_rx_packet();

                //
                // do something with packet here ...
                //
                printf ("Rx Packet prio:%u\r\n", rfRxInfo.pPayload[1]);

                wd_release_rx_packet();
                nrk_wait_until_next_period();
        }
}

void nrk_create_taskset()
{
        TTaskSnd.task = TaskSnd;
        TTaskSnd.Ptos = (void *) &StackTaskSnd[NRK_APP_STACKSIZE];
        TTaskSnd.Pbos = (void *) &StackTaskSnd[0];
        TTaskSnd.prio = SND_TASK_PRIO;
        TTaskSnd.FirstActivation = TRUE;
        TTaskSnd.Type = BASIC_TASK;
        TTaskSnd.SchType = PREEMPTIVE;
        TTaskSnd.period.secs = 0;
        TTaskSnd.period.nano_secs = SND_TASK_PERIOD_ms*NANOS_PER_MS;
        TTaskSnd.cpu_reserve.secs = 1;
        TTaskSnd.cpu_reserve.nano_secs =  SND_TASK_RESERVE_ms*NANOS_PER_MS;
        TTaskSnd.offset.secs = 0;
        TTaskSnd.offset.nano_secs= 0;
        nrk_activate_task (&TTaskSnd);

        TTaskRcv.task = TaskRcv;
        TTaskRcv.Ptos = (void *) &StackTaskRcv[NRK_APP_STACKSIZE];
        TTaskRcv.Pbos = (void *) &StackTaskRcv[0];
        TTaskRcv.prio = RCV_TASK_PRIO;
        TTaskRcv.FirstActivation = TRUE;
        TTaskRcv.Type = BASIC_TASK;
        TTaskRcv.SchType = PREEMPTIVE;
        TTaskRcv.period.secs = 0;
        TTaskRcv.period.nano_secs = RCV_TASK_PERIOD_ms*NANOS_PER_MS;
        TTaskRcv.cpu_reserve.secs = 10;
        TTaskRcv.cpu_reserve.nano_secs =  RCV_TASK_RESERVE_ms*NANOS_PER_MS;
        TTaskRcv.offset.secs = 0;
        TTaskRcv.offset.nano_secs= 0;
        nrk_activate_task (&TTaskRcv);
```

```
}
```

Bibliography

[1] Bosch. *CAN Specification, ver. 2.0, Robert Bosch GmbH, Stuttgart*, 1991.

[2] A. K. Mok and S. Ward. Distributed broadcast channel access. *Computer Networks*, 3:327–335, 1979.