Frequency Agility Protocol for nRF24XX

nAN24-07

1. General

This application note describes the low-level Frequency Agility Protocol for nRF24XX. This is a protocol that gives protection against disturbing traffic from frequency stationary systems like WLAN and frequency hopping devices like Bluetooth.

The protocol is generic and can be used in many different systems that require resistance against disturbance from other systems. By using this protocol, a system can operate in close proximity with systems using different WLAN channels, 2,4GHz cordless phones, 2,4GHz remote controls, microwave ovens, Bluetooth devices and other proprietary 2,4GHz systems. Basing its functionality on recovery by retransmission of lost packets, it will be much more reliable than a uni-directional communication protocol.

This document describes the assumptions that the protocol is based on, a description of the functionality, implementation described with state machine diagrams, current consumption calculations and C-code examples.

A 2.4GHz-mouse/keyboard application is a typical 2.4GHz application that will be used in proximity with WLAN and Bluetooth nodes. This document will for that reason use the wireless mouse/keyboard application when describing the Frequency Agility Protocol for nRF24XX.

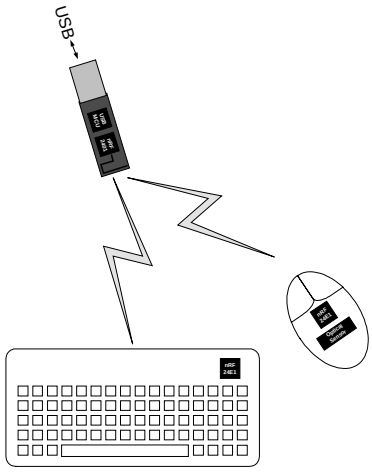

Figure 1: Wireless 2.4GHz mouse/keyboard application

As seen in Figure 1, a wireless 2.4GHz mouse/keyboard application consists of a wireless mouse, a wireless keyboard and a receiver unit for the PC side, referred to as a “dongle” in the rest of this document.

2. Assumptions

The frequency agility protocol is built on a series of assumptions regarding mouse / keyboard applications and the traffic in the 2.4GHz band.

The traffic in the 2.4GHz band is mainly consistent of frequency stationary systems like WLAN and frequency-hopping systems like Bluetooth. While frequency stationary systems operate in a specific part of the band, frequency-hopping systems will generate traffic in the whole band. All traffic generated by systems operating in the 2.4GHz band is packet based.

At a given channel in the 2.4GHz band, if a frequency hopping system is present, the likelihood of a collision with traffic from that system is the same in every channel. It is therefore no use in changing the operating channel if disturbed by a frequency hopping system. If the disturbance comes from a frequency stationary system, it is possible to move in such manner that the likelihood for a collision with the same system on the new channel is minimal.

A mouse will require a much higher update rate than a keyboard. It is assumed that when a mouse is used, it should be updated every 8th millisecond. The mouse will therefore have priority in front of the keyboard regarding updates.

The disturbance from other systems will be strongest close to the PC, and the dongle attached to the PC will suffer the most.

3. Frequency agility protocol for a 2.4GHz mouse/keyboard application

Based on the previous assumptions the definition of the frequency agility protocol emerges:
"A protocol that will move own traffic to another channel in the 2.4GHz band if a stationary disturbance occurs at the currently used frequency."

The main functionality of the frequency agility protocol will be to:

- Detect stationary disturbance.
- Move in such manner that new disturbance from the same source will not occur.
- Do not move if disturbed by a frequency hopping source.
- Give priority to mouse traffic.

It is important to notice that this protocol will only force a change in operating frequency when a stationary disturbance occurs. After it has changed the operating frequency, it will be on the new channel for a relative long time.

The frequency agility protocol functionality is based on the communication between the mouse and the dongle. When the mouse is in use, it will send a packet to the dongle every 8th millisecond and wait for acknowledge. The mouse will re-send a packet up to two times if no acknowledgement has been received. Bluetooth will stay up to 650 microseconds on one channel before hopping. This means that if a Bluetooth system is knocking out the mouse's first attempt to send a packet, the next two should get trough since each packet –

acknowledgement cycle takes about one millisecond. It is therefore not likely that a frequency hopping system will cause a change in frequency.

If all three attempts to send a packet fail, the mouse and dongle will change channel according to a table. The table is built up to take care of the functionality that avoids disturbance from the same source at the new channel. Figure 2 shows a typical table with channels used by the frequency agility protocol. The table is "WLAN weighted," meaning it will find the next channel outside of the assumed WLAN channel that is disturbing the currently used channel.

Index	Channel	Frequency [MHz]
0	2	2402
1	32	2432
2	70	2470
3	5	2405
4	35	2435
5	68	2468
6	8	2408
7	39	2439
8	65	2465
9	11	2411
10	41	2441
11	62	2462

Figure 2: Example of a WLAN weighted channel table

As seen from the Figure 3, the WLAN traffic can be found in three sub-bands in the 2.4GHz band. Looking at Figure 2 and Figure 3 we'll see that the table in Figure 2 will take care of moving the traffic out of a disturbing WLAN channel.

Figure 3: WLAN channels in the 2.4GHz band

If a channel is found to be very noisy, it is desired not to use that channel again. A timing of how long a channel has been used before disturbance occurs will tell how noisy the channel is. Channels that are found to be too noisy can be masked out for a period of time. Masked out channels will not be used for a while.

4. Implementation

This chapter will give a state machine description of the frequency agility protocol for a mouse and a keyboard communicating with a dongle at the PC side of the link.

4.1. The mouse implementation

Figure 4 shows the complete state machine of the frequency agility protocol running in the mouse. The following text will explain the different states.

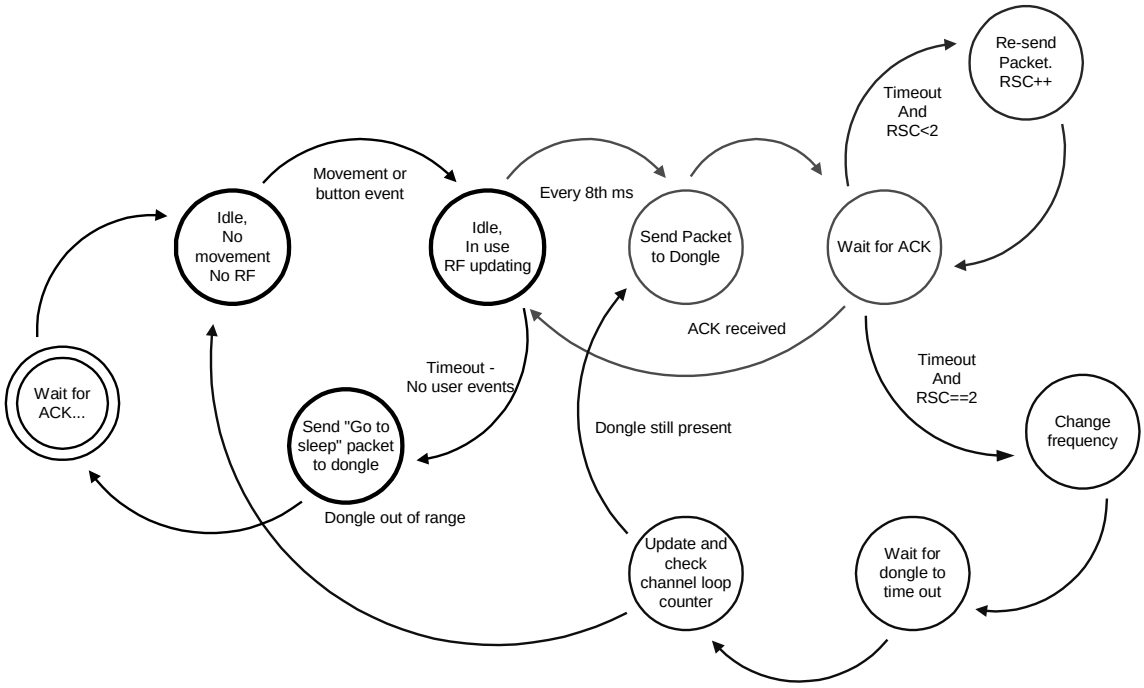

Figure 4: State machine diagram of the protocol on the mouse side

Idle, No movement, No RF is the state where the user does not use the mouse. This is the state where the mouse spends most of its time. In this state, the mouse will only poll for movement. There is no RF communication going on here.

As soon as a user event occurs, the mouse will go into the **Idle, In Use, RF updating** state. In this state, the mouse will send a packet to the dongle by entering the **Send Packet to Dongle** state every 8th millisecond. The packets are sent with a fixed interval even if no new user data is present.

After finishing sending the packet, the mouse moves to the state **Wait for ACK** (acknowledgement.) Normally an acknowledgement will be received from the dongle and the mouse returns to the **Idle, In Use, RF updating** state. This loop is the normal communication loop.

Sometimes, if a packet from the mouse to the dongle gets lost, or an acknowledgement from the dongle to the mouse fails to be received, a timeout must occur in the **Wait for ACK** state. Which state to enter next will then depend on the "Re-Send Counter" (RSC.) If it is less than two for the current packet, the packet is re-sent by entering the **Re-send Packet** state. If the RSC equals two, the mouse enters the **Change Frequency** state.

In the **Change Frequency** state it will perform a table look up to find the next channel to use. It will also mask out a channel that is very noisy, preventing use of this channel in close future. A background timer telling how long the current channel has been used will give input to the masking process. If a channel has been used for less than 20ms before a new frequency change is initiated, it should be masked out. The masked out channels should be reset after a given time period.

After changing frequency, the mouse enters the **Wait for dongle to time out** state. Since the dongle might has received the packets from the mouse, and it is the acknowledgements that have been lost that is the reason for the frequency change, the mouse must wait for the dongle to time out.

Then the mouse will enter the **Update and check channel loop counter** state where it updates and check its channel loop counter to determine if the dongle is still present or not. This decision is made from checking how many times the mouse has looped trough the channel table without contact with the dongle. If the decision is that the dongle is still present, the mouse returns to the **Send Packet to Dongle** state, if not it returns to the **Idle, No movement, No RF** state.

In the **Idle, In Use, RF updating** state the mouse will time out if no user evens has not occurred in a while. The mouse will then send a "I'm going to sleep" packet to the dongle by entering the **Send "Go to sleep" packet to dongle** state. After receiving acknowledgement from the dongle, the mouse enters the low power state; **Idle, No movement, No RF.**

4.2. The keyboard implementation

Figure 5 shows the complete state machine of the frequency agility protocol running in the keyboard. The following text will explain the different states.

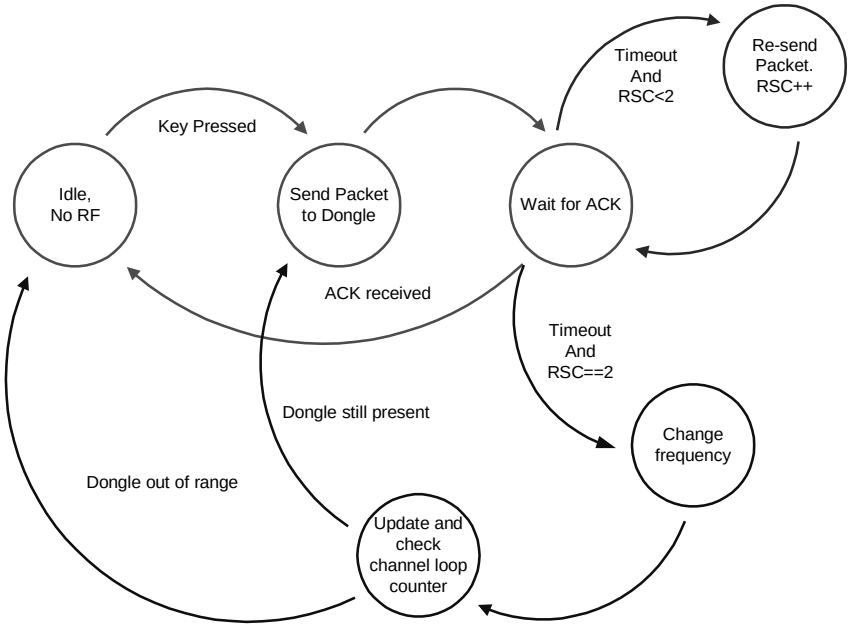

Figure 5: State machine diagram of the protocol on the keyboard side

The **Idle, No RF** state is the state where the keyboard is most of its time. In this state it does not perform any RF communications. Only when a key press occurs, the keyboard enters the **Send Packet to dongle** state.

As for the mouse, it will right after sending the packet, enter the state **Wait for ACK**, where it waits for an acknowledgement to arrive. Normally it will be received, and the keyboard will return to the **Idle, No RF** state.

If no acknowledgement is received, the keyboard will perform the same re-send behavior as the mouse. If acknowledgement still doesn't arrive, the keyboard will change frequency in the **Change Frequency** state.

The keyboard will use a channel table like the mouse, but it will not perform any masking of channels. After changing frequency, the keyboard will also determine if the dongle is present or not in the **Update and check channel loop counter** state. If the dongle is still present, it will return to the **Send Packet to dongle** state where it will try to send the packet to the dongle at the new channel. If the dongle is not present, it will go to the **Idle, No RF** state.

As seen here, the keyboard will follow the mouse and the dongle in frequency. The dongle or the mouse will not tell the keyboard that a channel change has taken place. The keyboard will therefore have to run trough its channel table until it finds the dongle, if the mouse and dongle have moved. Since this is a relative rare event, it will not cause any problems for the keyboard user. The benefit is that the keyboard does not need to receive messages from the dongle when it rests in the idle state, and will therefore use minimum of current.

4.3. The dongle implementation

Figure 6 shows the complete state machine of the frequency agility protocol running in the keyboard. The following text will explain the different states.

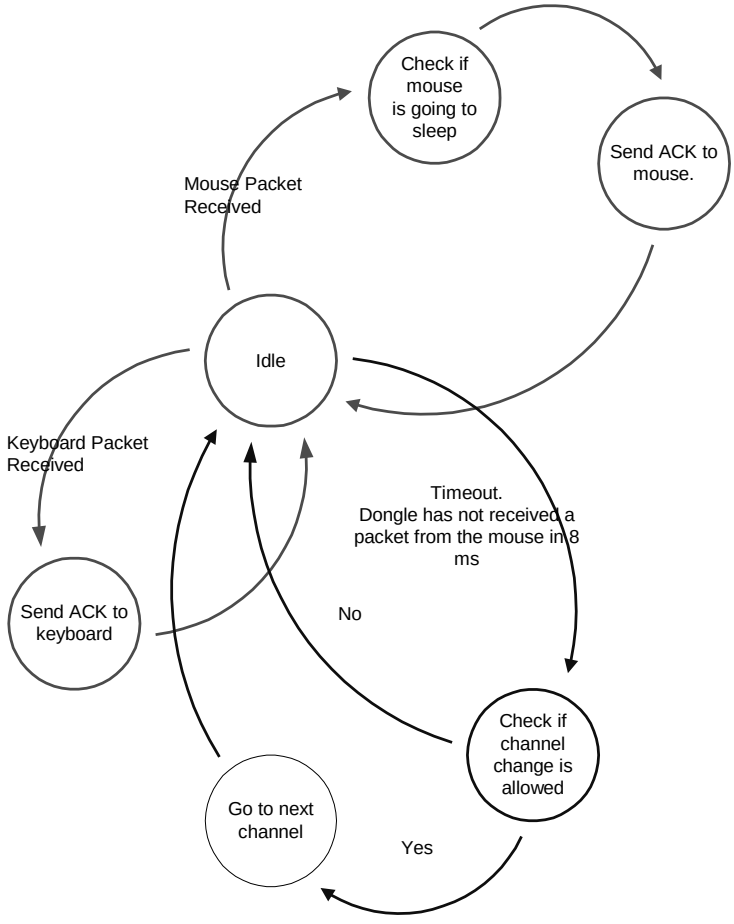

Figure 6: State machine diagram of the protocol on the dongle side

The dongle uses the DuoCeiver functionality and is able to receive from two different channels simultaneously, one for the mouse and one for the keyboard.

In the **Idle** state it checks for received packets from the mouse and the keyboard. If two arrive at the same time, it will give priority to the mouse since the mouse requires the highest update rate.

If a packet from the mouse is received, it will enter the **Check if mouse is going to sleep** state and set a flag if the mouse is going to sleep and clear the flag if it is an ordinary packet from the mouse. It will then acknowledge the received packet by entering the **Send ACK to mouse** state and finally return to the **Idle** state.

If a packet is received from the keyboard, the dongle will acknowledge it by entering the **Send ACK to keyboard** state and then return to the **Idle** state.

If eight milliseconds elapse without receiving any packets from the mouse, the dongle will enter the **Check if channel change is allowed** state. If the mouse sleeps, the dongle is not allowed to change channel. Also, if the dongle has not had communication with the mouse on the current channel, it is not allowed to change channel. This is done to avoid a "runaway dongle."

If the dongle is allowed to change channel it will enter the **Go to next channel** state, if not it will return to the **Idle** state. In the **Go to next channel** state, the dongle will do the same exercise as the mouse regarding masking of channels. After an eventual masking the dongle will change its operating channel and return to the **Idle** state.

5. Proof of concept

A demonstration of this frequency agility protocol exists. Nordic Semiconductor has developed the "nRF24XX Bi-directional mouse/keyboard demo" that runs the frequency agility protocol. This demo can be used to evaluate the protocol running on the nRF24E1 in environment where WLAN and other 2.4GHz systems are operating.

The dongle that comes with the "nRF24XX Bi-directional mouse/keyboard demo" can be plugged directly into a USB port. The dongle has a form factor that is very close to a real mouse/keyboard application dongle.

The two PS/2 reader boards that comes with the "nRF24XX Bi-directional mouse/keyboard demo" can be connected to a PS/2 mouse or a PS/2 keyboard and then emulate the functionality of a wireless mouse and a wireless keyboard. Both PS/2 readers can be used simultaneously, emulating a real wireless desktop solution.

Since the PS/2 reader board is not a real mouse or a keyboard, it has some limitations. The user should beware that the "nRF24XX Bi-directional mouse/keyboard demo" does not include the sleep mode for the mouse. Some of the timing constants used by the demo might also differ from optimal timing constants in a real mouse/keyboard application.

6. Current consumption

The mouse and the keyboard will have to turn on their receiver while waiting for an acknowledgement to arrive from the dongle. That is the reason why the mouse and the keyboard will use more current in a bi-directional solution compared to a uni-directional solution. In this chapter we'll have a closer look at the current consumption for both the mouse and the keyboard.

6.1. The mouse current consumption

The calculations shown in Figure 7 give an average current consumption for the **RF communication** when the mouse is used.

Packet Length from Mouse to PC	Pl	80	bits	8bit preamble, 32 bit address, 24bit X,Y and Z movement, 8 bit buttons, 8 bit CRC
Packet length from PC to Mouse	AckL	56	bits	8bit preamble, 32 bit address, 8 bit ACK info, 8 bit CRC
Data rate	DR	1	Mbit/s	
nRF24E1 TX current consumption	Itx	13	mA	
nRF24E1 RX currnet consumption	Irx	19	mA	
Mouse Update Rate	Tur	8	ms	
Time in TX mode	Ttxm	282	us	
Time in RX mode	Trxm	300	us	Long enough to catch a packet, with margins An ACK packet is 56 bits long (56us.) RX startup time is 202us. This gives 42us spare time to margins
Average current consumption in TX only	IavgTX	0,46	mA	This is the average current consumption when the mouse is actually used
Average current consumption in RX only	IavgRX	0,71	mA	This is the average current consumption when the mouse is actually used
Average current consumption, Two way solution	IavgTW	1,17	mA	This is the average current consumption when the mouse is actually used

Figure 7: Calculation of the average current consumption when the mouse is used

As seen in Figure 7 the average current consumption for the mouse in use is 1.17mA. The calculation assumes a update rate of one packet every 8th millisecond. In a real mouse application, this will be the current consumption used by the nRF part when the mouse is updating the dongle. The actual average current consumption will depend on how often the mouse is used.

6.2. The keyboard current consumption

Since the keyboard does not use the nRF device unless a key is pressed, the average current consumption will depend on how often a key is pressed. The calculation in Figure 8 shows the total energy used by the nRF device to send a keyboard packet and receive acknowledgement from the dongle.

Packet Length from Keyboard to PC	Pl	112	bits	8bit preamble, 32 bit address, 64 bit payload, 8 bit CRC
Packet length from PC to Keyboard	AckL	56	bits	8bit preamble, 32 bit address, 8 bit ACK info, 8 bit CRC
Data rate	DR	1	Mbit/s	
nRF24E1 TX current consumption	Itx	13	mA	
nRF24E1 RX currnet consumption	Irx	19	mA	
Time in TX mode	Ttxm	314	us	
Time in RX mode	Trxm	300	us	Long enough to catch a packet, with margins An ACK packet is 56 bits long (56us.) RX startup time is 202us. This gives 42us spare time to margins
Energy consumption in TX only	IavgTX	1,13E-06	mAh	Energy used to send the keyboard packet
Energy consumption in RX only	IavgRX	1,58E-06	mAh	Energy used to receive ACK
Energy consumption, Two way solution	IavgTW	2,72E-06	mAh	Total energy used pr key press

Figure 8: Calculation of the energy consumption pr key press

The nRF device uses a total of 2.72 nAh of energy every time a key is pressed.

7. Code

This chapter shows code examples of the frequency agility protocol. It can be used as reference for implementing similar functionality in a real mouse/keyboard application. The code will only show the parts needed for the frequency agility protocol. Higher level protocol layers must be added by the individual application engineer.

The code is written to run on the nRF24E1 8051 core.

7.1. C-code for the mouse

```
unsigned char SpiReadWrite(unsigned char b)
{
    EXIF &= ~0x20;                  // Clear SPI interrupt
    SPI_DATA = b;                   // Move byte to send to SPI data register
    while((EXIF & 0x20) == 0x00)    // Wait until SPI hs finished transmitting
        ;
    return SPI_DATA;
}


void TXPacket(void)
{
       unsigned char b;
       CE=1;                        // Set CE high
    for(b=0;b<BufPacket.length;b++) // Load packet into ShockBurst TX register
    {
        SpiReadWrite(BufPacket.buf[b]);
    }
    CE = 0;                         // Clear CE, transmission starts
       Delay100us(3);               // Wait 300us
}

void ChangeChannel(unsigned char channel,RXtx)
{
       CE=0;                        // CE=0 before configuration
       Delay100us(0);               // Delay min 5us before...
       CS = 1;                      // ..CS=1
       LCH=channel;                 // Remember LastCHannel
    SpiReadWrite((channel<<1)|RXtx); // Write LSB to RF config
    CS = 0;                         // CS=0 to end configuration
       if(RXtx)                     // If in RX mode...
       {
              Delay100us(0);        // ..delay min 5us before..
              CE = 1;               // ..CE=1
       }
}


unsigned char CheckACK(void)
{
       if(DR1)                      // Check if data has been received
       {
              PID=SpiReadWrite(0);  // Save received info
          NoDodge=SpiReadWrite(0);  // Save received info
              SpiReadWrite(0);
              SpiReadWrite(0);
              return 1;             // Return 1 to indicate ACK received
    }
       else
              return 0;             // Return 0 to indicate no ACK received
}

unsigned char WaitForACK(unsigned char x)
{

       while((!CheckACK())&(x>0))   // Check if ACK is received
       {
              x--;                  // Decrement counter
              ChangeChannel(LCH,0); // Go TX at same channel
              TXPacket();           // Retransmit
              ChangeChannel(LCH,1); // Go RX at same channel
              Delay100us(ACKTime);  // Delay, enable ACK to arrive
       }
       if(x>0)                      //
              return 1;             // Return 1 if ACK has been received
       else
              return 0;             // Return 0 if ACK has not been received
}
```

```
void ConfigRF(void)
{
       unsigned char b;
       CE=0;
       CS = 1;
    Delay100us(0);
    for(b=0;b<rconf.n;b++)
    {
        SpiReadWrite(rconf.buf[b]);
    }
    CS = 0;
}

void TXMousePacket(unsigned char b,x,y,z)
{
       unsigned char i;
       if(!notsendt)
       {
              PWMDUTY=220;
              y=~y+1;        // Adjust Y coordinate
              z=~z+1;        // Adjust Z coordinate
              PID++;         // Increment PacketID counter
              if((PID)>200)  // reset PID at 200
                     PID=0;
              for(i=0;i<AddressLength;i++)  // Load address
                     BufPacket.buf[i]=RXAddress[i];
              BufPacket.buf[5]=PID;  // Send PID first in payload
              BufPacket.buf[6]=b;    // Send button bits
              BufPacket.buf[7]=x;    // Send X movement
              BufPacket.buf[8]=y;    // Send Y movement
              BufPacket.buf[9]=z;    // Send Z movement
              BufPacket.buf[10]=0x00;// Reserved
              BufPacket.length=11;
              PWMDUTY=50;
       }
}



void TickLimitReached(unsigned char TL)
{
       unsigned char i;
       CLKOUT=1;                     // Bring the clock low to inhibit PS/2
       ChangeChannel(LCH,0);         // Go TX at same channel
       TXPacket();                   // Send the packet
       ChangeChannel(LCH,1);         // Go RX at same channel
       Delay100us(ACKTime);          // Delay
       if(!WaitForACK(2))            // Check if ACK is received
       {                             // No ACK received in 2 attempts
              if(MaskTick<20)        // Check if channel should be masked
                     MaskTable[DTi]=1; // Mask it.
              MaskTick=0;            // Clear MaskTick
              i=DTi;                 // Remember last DTi
              DTi++;                 // Increment DTi
              if(DTi>(DodgeTableSize-1))    // Check for Wrap
                     DTi=0;
              while(MaskTable[DTi]) // Find the next channel not masked
              {
                     DTi++;         // Incrementing DTi
                     if(DTi>(DodgeTableSize-1)) // Check if wrap
                            DTi=0;
                     if(DTi==i)     // Check if checked before
                     {
                            for(i=0;i<DodgeTableSize;i++) // Reset mask table
                                   MaskTable[i]=0;
                            break;
                     }
              }
              if(!NoDodge)          //
                     ChangeChannel(DodgeTable[DTi],1);    // Go RX at new channel
              if(!speeddodge)       // Relaod LittleTick..
              {
                     LittleTick=0;  // with 0 if first time changed
```

```
                        speeddodge=1;
                }
                else
                {
                        LittleTick=TL-2; // with TL-2 if not first time changed
                }
        }
        else
        {
                speeddodge=0;           // Clear flag if communication with dongle
                LittleTick=TL-5;        // Adjust LittleTick to avoid dongle timeout
                notsendt=0;             // Clear the notsend flag
        }
        CLKOUT=0;
}


void Timer2ISR (void) interrupt 5  using 1
{
        LittleTick++;                   // Increment LittleTick
        if(MaskTick<250)                // Increment MaskTick if less than 250
                MaskTick++;
        TF2 = 0;                        // Clear timer2 interrupt
}
```

7.2. C-code for the keyboard

```
unsigned char SpiReadWrite(unsigned char b)
{
    EXIF &= ~0x20;                  // Clear SPI interrupt
    SPI_DATA = b;                   // Move byte to send to SPI data register
    while((EXIF & 0x20) == 0x00)    // Wait until SPI hs finished transmitting
        ;
    return SPI_DATA;
}


void TXPacket(void)
{
        unsigned char b;
        CE=1;                       // Set CE high
    for(b=0;b<BufPacket.length;b++)  // Load packet into ShockBurst TX register
    {
         SpiReadWrite(BufPacket.buf[b]);
    }
    CE = 0;                         // Clear CE, transmission starts
        Delay100us(3);              // Wait 300us
}

void ChangeChannel(unsigned char channel,RXtx)
{
        CE=0;                       // CE=0 before configuration
        Delay100us(0);              // Delay min 5us before...
        CS = 1;                     // ..CS=1
        LCH=channel;                // Remember LastCHannel
    SpiReadWrite((channel<<1)|RXtx); // Write LSB to RF config
    CS = 0;                         // CS=0 to end configuration
        if(RXtx)                    // If in RX mode...
        {
                Delay100us(0);      // ..delay min 5us before..
                CE = 1;             // ..CE=1
        }
}


unsigned char CheckACK(void)
{
        if(DR1)                     // Check if data has been received
        {
                PID=SpiReadWrite(0);  // Save received info
            NoDodge=SpiReadWrite(0);  // Save received info
                SpiReadWrite(0);
                SpiReadWrite(0);
                return 1;           // Return 1 to indicate ACK received
    }
        else
                return 0;           // Return 0 to indicate no ACK received
}

unsigned char WaitForACK(unsigned char x)
{

        while((!CheckACK())&(x>0))  // Check if ACK is received
        {
                x--;                // Decrement counter
                ChangeChannel(LCH,0); // Go TX at same channel
                TXPacket();         // Retransmit
                ChangeChannel(LCH,1); // Go RX at same channel
                Delay100us(ACKTime);  // Delay, enable ACK to arrive
        }
        if(x>0)                     //
                return 1;           // Return 1 if ACK has been received
        else
                return 0;           // Return 0 if ACK has not been received
}
```

```
void ConfigRF(void)
{
       unsigned char b;
       CE=0;
       CS = 1;
    Delay100us(0);
    for(b=0;b<rconf.n;b++)
    {
        SpiReadWrite(rconf.buf[b]);
    }
    CS = 0;
}


void TXKeyPacket(unsigned char b,x)
{
       unsigned char i;
       CLKOUT=1;              // Bring the clock low to inhibit PS/2
       PWMDUTY=220;           // Turn on the LED
       PID++;                 // Increment PacketID counter
       if((PID)>200)          // reset PID at 200
              PID=0;
       for(i=0;i<AddressLength;i++)
              BufPacket.buf[i]=RXAddress[i]+1;
                             // Address bytes on keyboard is incremented with 1.
       BufPacket.buf[5]=PID;  // First byte in payload is PID
       BufPacket.buf[6]=b;    // Flags for special keys
       BufPacket.buf[7]=x;    // Key scan code
       BufPacket.buf[8]=0x00; // Reserved
       BufPacket.buf[9]=0x00; // Reserved
       BufPacket.buf[10]=0x00;// Reserved
       BufPacket.length=11;   // Set packet length

       ChangeChannel(LCH,0);  // Go TX at same channel
       TXPacket();            // Send packet
       ChangeChannel(LCH,1);  // Go RX at same channel
       Delay100us(ACKTime);   // Delay
       while(!WaitForACK(2))  // Check if ACK is received
       {                      // Change channel if not
              DTi++;
              if(DTi>(DodgeTableSize-1))
                     DTi=0;
              ChangeChannel(DodgeTable[DTi]+8,1);   // Go RX at new channel
       }
       PWMDUTY=50;            // Turn off the LED
       CLKOUT=0;              // Release clock to enable PS/2
}
```

7.3. C-code for the dongle

```
#include <Nordic VLSI\reg24e1.h>

struct RFConfig
{
    unsigned char n;
    unsigned char buf[15];
};

struct Packet
{
       unsigned char length;
       unsigned char buf[20];
};

typedef struct RFConfig RFConfig;
typedef struct Packet Packet;

#define ADDR_INDEX 7                        // Index to address bytes in RFConfig.buf
#define ADDR_COUNT 5                        // Number of address bytes
#define DodgeTableSize 9                    // Dodge table size
#define AddressLength 5
#define tickperiod 1                        // in ms
#define timervalue (65536-16000000/(4*(1000/tickperiod))) // Calculate timer value


const unsigned char RXAddress[]      = {0xED, 0xBA, 0x7E, 0xDF, 0xDD};
const unsigned char DodgeTable[]     = {2,27,52,8,33,58,14,39,64};
unsigned char MaskTable[]            = {0,0 ,0 ,0,0 ,0 ,0 ,0 ,0};

const RFConfig rconf =                      // nRF configuration word
{
    15,
    0x30, 0x30, 0xEE, 0xBB, 0x7F, 0xE0, 0xDE, 0xED,
    0xBA, 0x7E, 0xDF, 0xDD, 0xA1, 0xEF, 0x46
};


unsigned char LCH,PID=0,NoDodge=0,DTi=0,LittleTick=0,MaskTick=0;
volatile unsigned char flag=0;

Packet BufPacket;
Packet OutPack;

void convoy(unsigned char type);

void Delay100us(volatile unsigned char n)
{
    unsigned char i;
    while(n--)
        for(i=0;i<32;i++)
            ;
}


unsigned char SpiReadWrite(unsigned char b)
{
    EXIF &= ~0x20;                          // Clear SPI interrupt
    SPI_DATA = b;                           // Move byte to send to SPI data register
    while((EXIF & 0x20) == 0x00)            // Wait until SPI hs finished transmitting
        ;
    return SPI_DATA;
}

void ChangeChannel(unsigned char channel,RXtx)
{
       CE=0;                                // CE=0 before configuration
       Delay100us(0);                       // Delay min 5us before...
       CS = 1;                              // ..CS=1
       LCH=channel;                         // Remember LastCHannel
```

```
        SPI_CTRL = 0x02;                        // Connect SPI to RADIO CH1
    SpiReadWrite((channel<<1)|RXtx);            // Write LSB to RF config
    CS = 0;                                     // CS=0 to end configuration
        if(RXtx)                                // If in RX mode...
        {
                Delay100us(0);                  // ..delay min 5us before..
                CE = 1;                         // ..CE=1
        }
}
void ConfigRF(void)
{
        unsigned char b;
        CE=0;                                   // Set CE low
        Delay100us(0);                          // Short delay
        CS = 1;                                 // Set CS high
        SPI_CTRL = 0x02;                        // Connect SPI to RADIO CH1
    for(b=0;b<rconf.n;b++)                      // Clock out all configuration data
    {
         SpiReadWrite(rconf.buf[b]);
    }
    CS = 0;                                     // Clear CS -> End configuration
}

unsigned char ReceiveMouse(void)
{
        if(DR1)                                 // Check if DR1 is high
        {
                SPI_CTRL = 0x02;                // Connect SPI to RADIO CH1
                BufPacket.length=0;             // Clear Buffer packet length
                while(DR1)                      // Clock out all received data
                {
              BufPacket.buf[BufPacket.length++]=SpiReadWrite(0);
                }
                return 1;                       // Return 1 to indicate reception
    }
        else
                return 0;                       // Return 1 to indicate no reception
}

unsigned char ReceiveKeyboard(void)
{
        if(DR2)                                 // Check if DR2 is high
        {
            SPI_CTRL = 0x03;                    // Connect SPI to RADIO CH2
                BufPacket.length=0;             // Clear Buffer packet length
                while(DR2)                      // Clock out all received data
                {
              BufPacket.buf[BufPacket.length++]=SpiReadWrite(0);
                }
                return 1;                       // Return 1 to indicate reception
    }
        else
                return 0;                       // Return 1 to indicate no reception

}

void TXPacket(void)
{
        unsigned char b;
        CE=1;                                   // Set CE high
        SPI_CTRL = 0x02;                        // Connect SPI to RADIO CH1
    for(b=0;b<BufPacket.length;b++)             // Clock out the packet
    {
         SpiReadWrite(BufPacket.buf[b]);
    }
    CE = 0;                                     // Set CE low (TX starts)
        Delay100us(3);                          // Wait 300us
}

void Timer2ISR (void) interrupt 5  using 1
{
        LittleTick++;                           // Increment LittleTick every ms
        if(MaskTick<250)
                MaskTick++;                     // Increment MaskTick every ms until 250
```

```
    TF2 = 0;                                // Clear timer2 interrupt
}

void Init(void)
{
    // I/O
    P0_DIR = 0x02;          // P0.1 is input, rest are output

        P1_DIR = 0x00;
        P1 = 0x00;

    // SPI:
    SPICLK = 0x00;          // Max SPICLK (=CLK/8)
    SPI_CTRL = 0x02;        // Connect SPI to RADIO CH1

    // Radio
    PWR_UP = 1;             // Turn on Radio
    Delay100us(30);         // Wait > 3ms
        ConfigRF();

    // TIMER2:
    TR2 = 0;                // Stop timer2 if running
    CKCON |= 0x20;          // T2M=1 (/4 timer clock)

    RCAP2L = timervalue;    //
    RCAP2H = timervalue>>8; //

    TF2 = 0;                // Clear any pending timer2 interrupts
    TR2 = 1;                  // Start timer2
    ET2 = 1;                // Enable timer2 interrupts

    // System
    EA = 1;                   // Enable global interrupts
}

void BuildACK(unsigned char CCC)
{
        unsigned char i;
        BufPacket.buf[5]=BufPacket.buf[0];    // Return the PID to the device
        for(i=0;i<AddressLength;i++)          // Build address
                BufPacket.buf[i]=RXAddress[i];
        BufPacket.buf[6]=CCC;                 // Add user byte
        BufPacket.buf[7]=0;                   // Future use
        BufPacket.buf[8]=0;                   // Future use
        BufPacket.length=9;                   // Set the packet length
}



void main(void)
{
        unsigned char OldKeyboardPacketID,OldMousePacketID;
        unsigned char MouseSleeps=0,hasmoved=1,i,n;
        Init();
        ChangeChannel(DodgeTable[DTi],1);     // Go RX at first channel in the DodgeTable
        while(1)
        {
                if(ReceiveMouse())            // Check if data from the mouse has arrived
            {
                        if(OldMousePacketID != BufPacket.buf[0])// Check if it is a new packet
(not a re-sent one.)
                        {
                                OldMousePacketID=BufPacket.buf[0];// Save the new packet ID
                                for(i=0;i<BufPacket.length;i++)
                                // Transfer received data to the USB out buffer
                                        OutPack.buf[i]=BufPacket.buf[i];
                                flag=0x01;     // Flag that new mouse data are present

                                // A check of if the mouse is going to sleep
                                // must be done here to set the MouseSleep flag
                                // The MouseSleep flag is cleard if a new packet
                                // is received from the mouse.
                        }
                        LittleTick=0;         // Clear the tick counter
```

```
                        hasmoved=0;// Clear the "hasmoved" flag -> allow dongle to move in freq
                        ChangeChannel(LCH,0);  // Go TX at same channel
                        BuildACK(NoDodge);     // Build the acknowledgement
                        TXPacket();            // Send the acknowledgement to the mouse
                        ChangeChannel(LCH,1);  // Go RX at same channel
                }
                else
                        if(ReceiveKeyboard())  // Check if data from the keyboard has arrived
                    {
                                if(OldKeyboardPacketID != (BufPacket.buf[0]))// Check if it is a
new packet (not a re-sent one.)
                                {
                                        OldKeyboardPacketID=(BufPacket.buf[0]);// Save the new
packet ID
                                        for(i=0;i<BufPacket.length;i++)// Transfer received data
to the USB out buffer
                                                OutPack.buf[i]=BufPacket.buf[i];
                                        flag=0x05;// Flag that new keyboard data are present
                                }
                                ChangeChannel(LCH+8,0);// Go TX at keyboard channel (8MHz up)
                                BuildACK(NoDodge);     // Build the acknowledgement
                                TXPacket();            // Send the acknowledgement to the
keyboard
                                ChangeChannel(LCH-8,1);// Go RX at operating channel (8MHz down)
                        }

                if((LittleTick>8)&(!hasmoved)&(!MouseSleeps))// Timeout?
                {
                        // If it hasn't moved since last time a mouse packet arrived

                        // and the mouse doesn't sleep...
                        LittleTick=0;  // Clear the tick counter
                        hasmoved=1;    // Set the "hasmoved" flag
                        if(MaskTick<50)// Check if it is reason to mask the old channel
                                MaskTable[DTi]=1;
                        MaskTick=0;    // Reset the MaskTick counter
                        n=DTi;         // Remember old DTi
                        DTi++;         // Increment DodgeTable index
                        while(MaskTable[DTi]) // Find an unmasked channel
                        {
                                DTi++;
                                if(DTi>(DodgeTableSize-1))
                                        DTi=0;
                                if(n==DTi)
                                {
                                        NoDodge=1;// If all channels are masked, stop dodging
                                        break;
                                }
                        }
                        if(DTi>(DodgeTableSize-1))// Check for wrap
                                DTi=0;
                        ChangeChannel(DodgeTable[DTi],1);// Go RX at new channel
                }
                convoy(flag);  // Send flags to USB convoy routine (the next layer.)
                flag=0;        // Clear flags
        }
}
```

1. LIABILITY DISCLAIMER

Nordic Semiconductor ASA reserves the right to make changes without further notice to the product to improve reliability, function or design. Nordic Semiconductor does not assume any liability arising out of the application or use of any product or circuits described herein.

LIFE SUPPORT APPLICATIONS

These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Nordic Semiconductor ASA customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Nordic Semiconductor ASA for any damages resulting from such improper use or sale.

Application Note, Revision: 1.0, Date: 12.10.2004.

Application Note order code: nAN24-07

YOUR NOTES

Nordic Semiconductor - World Wide Distributors

For Your nearest dealer, please see http://www.nordicsemi.no

Main Office:
Vestre Rosten 81, N-7075 Tiller, Norway
Phone: +47 72 89 89 00, Fax: +47 72 89 89 89

Visit the Nordic Semiconductor ASA website at http://www.nordicsemi.no