```c
#define F_CPU                   16000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>

#include "UART_Bluetooth.h"
#include "nrf24.h"

void initIO();
char messageTest[] = "UART TESTING COMMANDS! \n";

int main(void)
{
    cli();  // Interrupts off
    initIO();
    initBluetoothUart();
    setupReceiveMode();
    nrf24_initRF_SAFE(POWER_BOARD_RF, RECEIVE); // CONNECTION TO POWER BOARD AND MOTORIZED BOARD : GENERAL RF CHANNEL 11
    sei();  // Interrupts on
    while (1)
    {
        if (commandAvailable) {
            cli();
            processReceivedLine();
            setupReceiveMode();

        }

         // Disable UART

        if(nrf24_dataReady())
        {
            cli();
            nrf24_getData(command_buffer);
            CommandStatus status = DecomposeMessageFromBuffer();
            if (status==SUCCESFUL_DECOMPOSITION) { RetransmissionToPhone();  }
            sei();
        }

        if (nrf24_checkAvailability()==false) { nrf24_initRF_SAFE(POWER_BOARD_RF, RECEIVE); }

    }
}


```

```
51  void initIO(){
52      /*
53          Input/Output pin initialization
54          1 : OUTPUT | 0 : INPUT | 0b76543210 Bit order
55          ATTACHMENTS
56              RED LED     : PD7               |   OUTPUT
57              GREEN LED   : PB0               |   OUTPUT
58          HC-05
59              TX          : PD0 (RX ATMEGA)   |   INPUT
60              RX          : PD1 (TX ATMEGA)   |   OUTPUT
61              KEY/ENABLE  : PD2               |   OUTPUT
62              STATE       : PC5               |   INPUT
63          nRF24L01
64              CE   : PC0                      |   OUTPUT
65              CSN  : PC1                      |   OUTPUT
66              MISO : PD0 (MSPIM MISO ATMEGA)  |   INPUT
67              MOSI : PD1 (MSPIM MOSI ATMEGA)  |   OUTPUT
68              SCK  : PD4 (MSPIM XCK)          |   OUTPUT
69      */
70      DDRD = 0b11111110;
71      DDRB = 0b00101001;
72      DDRC = 0b11011111;
73  }
74
75
76
77
78
79
```

```
 1
 2  #include "Command_Handler.h"
 3  #include "UART_Bluetooth.h"
 4  #include "nrf24.h"
 5  #include "crc.h"
 6
 7
 8
 9  const CommandType commandList[] = {
10      { .handlerFunction = &UPDATE_ALL_DEVICES_VALUE_H},
11      { .handlerFunction = &UPDATE_DEVICE_VALUE_H},
12      { .handlerFunction = &GET_ALL_DEVICES_VALUE_H},
13      { .handlerFunction = &GET_DEVICE_VALUE_H},
14      { .handlerFunction = &MESSAGE_STATUS_H}
15  };
16  #define commandListLength (uint8_t)(sizeof commandList/sizeof commandList[0])
17
18  bool initliazeMemory(){
19      if(memoryInitialized) return false;
20      parameter[0].startingPointer = (void*)calloc(23,1);
21      parameter[1].startingPointer = (void*)calloc(2,1);
22      parameter[2].startingPointer = (void*)calloc(2,1);
23      for (uint8_t x = 3; x<12; x++) parameter[x].startingPointer = (void*)calloc
          (1,1);
24      command_buffer = (uint8_t*)calloc(32,1);
25      if(command_buffer==NULL) return false;
26      for (uint8_t x = 0; x<12; x++) { if(parameter[x].startingPointer==NULL)
          return false; }
27      memoryInitialized = true;
28      return true;
29  }
30
31  CommandStatus DecomposeMessageFromBuffer(){
32      // Search for header
33      uint8_t* headerStart = command_buffer;
34      uint8_t* footerEnd = command_buffer+31;
35
36      for(;headerStart!=(command_buffer+22);headerStart++){
37          if (*headerStart==SOH&&(*(headerStart+4)==STX)){
38              for(;footerEnd!=(command_buffer+6);footerEnd--){
39                  if (*footerEnd==ETB&&(*(footerEnd-2)==ETX)){
40                      uint8_t netMessageLength = ((footerEnd-2)-headerStart);
41                      crc_t crc;
42                      crc = crc_init();
43                      crc = crc_update(crc, headerStart, netMessageLength);
44                      crc = crc_finalize(crc);
45                      if (*(footerEnd-1)!=crc) return WRONG_CHECKSUM_CONSISTENCY;
46                      if (*(headerStart+2)!=currentModuleID&&*(headerStart+2)!
                         =0xFF&&currentModuleID!=0x01) return WRONG_MODULE_ID;
47                      lastTargetModuleID = *(headerStart+2);
48                      lastTransmitterModuleID = *(headerStart+3);
49                      if (*(headerStart+5)>commandListLength-1) return
```

```
                    UNDEFINED_COMMAND_CODE;
50                  lastMessageCommandType = commandList[*(headerStart+5)];
51                  lastMessagePID = *(headerStart+1);
52
53                  uint8_t* parameterStart = headerStart+6;
54
55                  for (uint8_t x = 0; x < 12; x++) {
56                      realloc(parameter[x].startingPointer, *parameterStart);
57                      parameter[x].byteLength = *parameterStart;
58                      memcpy(parameter[x].startingPointer,parameterStart+1,
                     *parameterStart);
59                      parameterStart+=((*parameterStart)+1);
60                      if (parameterStart>=(footerEnd-2)) break;
61                  }
62
63                  return SUCCESFUL_DECOMPOSITION;
64              }
65          }
66      }
67  }
68  return WRONG_HEADER_SEGMENTATION;
69 }
70
71 CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t
    parameterCount, uint8_t targetBoardID){
72
73      memset(command_buffer, 0, 32);
74
75      command_buffer[0] = SOH;
76      if (lastMessagePID==0xFF) { lastMessagePID++; } else { lastMessagePID = 0; }
77      command_buffer[1] = lastMessagePID;
78      command_buffer[2] = targetBoardID;
79      command_buffer[3] = currentModuleID;
80      command_buffer[4] = STX;
81      command_buffer[5] = targetTypeID;
82
83      if (parameterCount>12) return PARAMETER_COUNT_OVERSIZE;
84
85      uint8_t* parameterStart = &command_buffer[6];
86
87      for (uint8_t x = 0; x < parameterCount; x++){
88          *parameterStart = parameter[x].byteLength;
89          memcpy(parameterStart+1, parameter[x].startingPointer, parameter
              [x].byteLength);
90          parameterStart+=(parameter[x].byteLength)+1;
91      }
92
93      crc_t crc;
94      crc = crc_init();
95      uint8_t crc_length = ((parameterStart)-(&command_buffer[0]));
96      crc = crc_update(crc, &command_buffer[0], crc_length);
97      crc = crc_finalize(crc);
```

```
 98
 99      *parameterStart = ETX;
100      *(parameterStart+1) = crc;
101      *(parameterStart+2) = ETB;
102
103      return SUCCESFUL_COMPOSITION;
104  }
105
106  void HandleAvailableCommand(){
107      lastMessageCommandType.handlerFunction();
108  }
109
110  RF_TransmissionStatus RetransmissionToModule(){
111      nrf24_initRF_SAFE((lastTargetModuleID-1), TRANSMIT);    // CONNECTION TO
           MODULE: GENERAL RF CHANNEL 112, (lastTargetModuleID-1) offset 1
112      nrf24_send(command_buffer);
113      while(nrf24_isSending());
114
115      uint8_t messageStatus = nrf24_lastMessageStatus();
116      if(messageStatus == NRF24_TRANSMISSON_OK) { return
           RF_SUCCESFUL_TRANSMISSION; }
117      else if(messageStatus == NRF24_MESSAGE_LOST) { return
           RF_UNREACHEABLE_MODULE;}
118      return RF_UNREACHEABLE_MODULE;
119  }
120
121  void RetransmissionToPhone(){
122      transmitMessageSync(command_buffer, 32);
123  }
124
125
126
127  void writeParameterValue(uint8_t parameterIndex, uint8_t* parameterData, uint8_t
       parameterByteLength){
128      parameter[parameterIndex].startingPointer = (uint8_t*) realloc(parameter
           [parameterIndex].startingPointer, parameterByteLength);
129      memcpy(parameter[parameterIndex].startingPointer, parameterData,
           parameterByteLength);
130      parameter[parameterIndex].byteLength = parameterByteLength;
131  }
132
133  void UPDATE_ALL_DEVICES_VALUE_H() {}
134  void UPDATE_DEVICE_VALUE_H() {}
135  void GET_ALL_DEVICES_VALUE_H() {
136      _delay_ms(100);
137
138      uint8_t boardState[2];
139
140      ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, POWER_MODULE);
141      nrf24_initRF_SAFE(POWER_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE:
           GENERAL RF CHANNEL 112
142      nrf24_send(command_buffer);
```

```
143     while(nrf24_isSending());
144
145     uint8_t messageStatus = nrf24_lastMessageStatus();
146     if(messageStatus == NRF24_TRANSMISSON_OK) { boardState[0] = 0xFF; }
147     else if(messageStatus == NRF24_MESSAGE_LOST) { boardState[0]= 0x00; }
148
149     _delay_ms(50);
150
151     ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, MOTOR_MODULE);
152     nrf24_initRF_SAFE(MOTORIZED_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE:
          GENERAL RF CHANNEL 112
153     nrf24_send(command_buffer);
154     while(nrf24_isSending());
155
156     uint8_t messageStatusSecond = nrf24_lastMessageStatus();
157     if(messageStatusSecond == NRF24_TRANSMISSON_OK) { boardState[1] = 0xFF; }
158     else if(messageStatusSecond == NRF24_MESSAGE_LOST) { boardState[1]= 0x00; }
159
160
161     writeParameterValue(0, &boardState[0], 1);
162     writeParameterValue(1, &boardState[1], 1);
163     ComposeMessageToBuffer(UPDATE_ALL_DEVICES_VALUE_ID, 2, PHONE_MODULE);  //
          PHONE_MODULE should be lastTransmitterModuleID
164     transmitMessageSync(command_buffer, 32);
165 }
166
167 void GET_DEVICE_VALUE_H() {
168     _delay_ms(100);
169     uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
170     uint8_t deviceValue;
171
172     switch(deviceIndex){
173         case 0:
174             ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, POWER_MODULE);
175             nrf24_initRF_SAFE(POWER_BOARD_RF, TRANSMIT);    // CONNECTION TO
                  MODULE: GENERAL RF CHANNEL 112
176             nrf24_send(command_buffer);
177             while(nrf24_isSending());
178
179             uint8_t messageStatus = nrf24_lastMessageStatus();
180             if(messageStatus == NRF24_TRANSMISSON_OK) { deviceValue = 0xFF; }
181             else if(messageStatus == NRF24_MESSAGE_LOST) { deviceValue= 0x00; }
182             break;
183         case 1:
184             ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, MOTOR_MODULE);
185             nrf24_initRF_SAFE(MOTORIZED_BOARD_RF, TRANSMIT);    // CONNECTION TO
                  MODULE: GENERAL RF CHANNEL 112
186             nrf24_send(command_buffer);
187             while(nrf24_isSending());
188
189             uint8_t messageStatusSecond = nrf24_lastMessageStatus();
190             if(messageStatusSecond == NRF24_TRANSMISSON_OK) { deviceValue =
```

```
                0xFF; }
191             else if(messageStatusSecond == NRF24_MESSAGE_LOST) { deviceValue=
                0x00; }
192             break;
193     }
194
195     writeParameterValue(0, &deviceIndex, 1);
196     writeParameterValue(1, &deviceValue, 2);
197
198     ComposeMessageToBuffer(UPDATE_DEVICE_VALUE_ID, 2, PHONE_MODULE);  //
        PHONE_MODULE should be lastTransmitterModuleID
199
200     transmitMessageSync(command_buffer, 32);
201 }
202 void MESSAGE_STATUS_H() {}
```

```
 1
 2
 3  #ifndef COMMAND_HANDLER_H_
 4  #define COMMAND_HANDLER_H_
 5
 6  #ifndef nullptr
 7  #define nullptr ((void *)0)
 8  #endif
 9
10  #ifndef F_CPU
11  #define F_CPU               16000000UL
12  #endif
13
14  #include <stdbool.h>
15  #include <stdint.h>
16  #include <stdio.h>
17  #include <string.h>
18  #include <stdlib.h>
19  #include <avr/io.h>
20  #include <util/delay.h>
21
22  #ifndef BIT_MANIPULATION_MACRO
23  #define BIT_MANIPULATION_MACRO 1
24  #define bit_get(p,m) ((p) & (m))
25  #define bit_set(p,m) ((p) |= (m))
26  #define bit_clear(p,m) ((p) &= ~(m))
27  #define bit_flip(p,m) ((p) ^= (m))
28  #define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
29  #define BIT(x) (0x01 << (x))
30  #define LONGBIT(x) ((unsigned long)0x00000001 << (x))
31  #endif
32
33  typedef struct CommandType {
34      void (*handlerFunction)();
35  } CommandType;
36
37  typedef enum   {
38      SUCCESFUL_DECOMPOSITION,
39      WRONG_HEADER_SEGMENTATION,
40      WRONG_FOOTER_SEGMENTATION,
41      WRONG_CHECKSUM_CONSISTENCY,
42      WRONG_MODULE_ID,
43      UNDEFINED_COMMAND_CODE,
44      PARAMETER_DATA_OVERFLOW,
45      PARAMETER_COUNT_OVERSIZE,
46      RETRANSMISSION_FAILED,
47      SUCCESFUL_RETRANSMISSION,
48      SUCCESFUL_COMPOSITION
49  } CommandStatus;
50
51
52  typedef enum {
```

```
53      RF_SUCCESFUL_TRANSMISSION,
54      RF_UNREACHEABLE_MODULE,
55      RF_ACKNOWLEDGE_FAILED
56  } RF_TransmissionStatus;
57
58  typedef enum   {
59      UPDATE_ALL_DEVICES_VALUE_ID,
60      UPDATE_DEVICE_VALUE_ID,
61      GET_ALL_DEVICES_VALUE_ID,
62      GET_DEVICE_VALUE_ID,
63      MESSAGE_STATUS_ID
64  } CommandTypeID;
65
66  typedef struct {
67      void *startingPointer;
68      uint8_t byteLength;
69  } Parameter;
70
71  typedef enum   {
72      PHONE_MODULE = 0x00,
73      MAIN_MODULE = 0x01,
74      POWER_MODULE = 0x02,
75      MOTOR_MODULE = 0x03,
76  } ModuleInternalCode;
77
78
79  #define currentModuleID MAIN_MODULE
80
81  #define SOH 0x01
82  #define STX 0x02
83  #define ETX 0x03
84  #define ETB 0x17
85  #define ON_STATE    0xFF
86  #define OFF_STATE   0x00
87
88  #define AVAILABLE_DEVICES 4
89  uint16_t device_value[AVAILABLE_DEVICES];
90
91  uint8_t *command_buffer;
92  Parameter parameter[12];
93  bool memoryInitialized;
94
95  uint8_t lastMessagePID;
96  uint8_t lastTargetModuleID;
97  uint8_t lastTransmitterModuleID;
98  CommandType lastMessageCommandType;
99
100 extern bool initliazeMemory();
101 extern void UPDATE_ALL_DEVICES_VALUE_H(), UPDATE_DEVICE_VALUE_H(),
      GET_ALL_DEVICES_VALUE_H(), GET_DEVICE_VALUE_H(), MESSAGE_STATUS_H();
102 extern CommandStatus DecomposeMessageFromBuffer();
103 extern void HandleAvailableCommand();
```

```
104  extern RF_TransmissionStatus RetransmissionToModule();
105  extern CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t
       parameterCount, uint8_t targetBoardID);
106  void writeParameterValue(uint8_t parameterIndex, uint8_t* parameterData, uint8_t
       parameterByteLength);
107  void RetransmissionToPhone();
108  #endif /* COMMAND_HANDLER_H_ */
```

```
 1
 2
 3  #include "UART_Bluetooth.h"
 4  #include <avr/io.h>
 5  #include <avr/interrupt.h>
 6  #include "Command_Handler.h"
 7  #include "nrf24.h"
 8  #include <stdlib.h>
 9  #include <string.h>
10
11  uint8_t* uartBufferPos;
12  uint8_t* uartTxMessageEnd;
13  bool commandAvailable;
14
15  void initBluetoothUart(){
16      // UART Initialization : 8-bit : No parity bit : 1 stop bit
17      UBRR0H = (BRC >> 8); UBRR0L =  BRC;             // UART BAUDRATE
18      UCSR0A |= (1 << U2X0);                          // DOUBLE UART SPEED
19      UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);        // 8-BIT CHARACTER SIZE
20
21      // Setup UART buffer
22      initliazeMemory();
23      uartBufferPos = command_buffer;
24  }
25
26  void transmitMessage(uint8_t* message, uint8_t length){
27      while (!(UCSR0A & (1<<UDRE0)));
28      uartBufferPos = command_buffer;
29      uartTxMessageEnd = (command_buffer+length);
30      memcpy(command_buffer, message, length);
31      UCSR0A |= (1<<TXC0) | (1<<RXC0);
32      UCSR0B |= (1<<TXEN0) | (1<<TXCIE0);
33      UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);
34
35      uartBufferPos++;
36      UDR0 = *(command_buffer);
37  }
38
39  void transmitMessageSync(uint8_t* message, uint8_t length){
40      while (!(UCSR0A & (1<<UDRE0)));
41      uartBufferPos = command_buffer;
42      uartTxMessageEnd = (command_buffer+length);
43      memcpy(command_buffer, message, length);
44      UCSR0A |= (1<<TXC0) | (1<<RXC0);
45      UCSR0B |= (1<<TXEN0) | (1<<TXCIE0);
46      UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);
47      sei();
48
49      uartBufferPos++;
50      UDR0 = *(command_buffer);
51
52      while (transmissionState());
```

```
53
54  }
55
56  bool transmissionState(){
57      // True : Currently transmitting | False : Transmission finished
58      if (uartBufferPos!=uartTxMessageEnd)
59      {
60          return true;
61      }
62      else
63      {
64          return false;
65      }
66  }
67
68
69  void setupReceiveMode(){
70      while (!(UCSR0A & (1<<UDRE0)));
71      uartBufferPos = command_buffer;
72
73      UCSR0A |= (1<<RXC0) | (1<<TXC0);
74      UCSR0B &=~(1<<TXEN0) &~(1<<TXCIE0);
75      UCSR0B |= (1<<RXEN0) | (1<<RXCIE0);
76      sei();
77  }
78
79  bool catchModuleReply(){
80      nrf24_initRF_SAFE((lastTargetModuleID-1), RECEIVE); // CONNECTION TO MODULE:
          GENERAL RF CHANNEL 112 (lastTargetModuleID-1) offset 1
81      uint8_t targetModuleID = lastTargetModuleID;
82      uint8_t RF_TIME_OUT;
83      while(RF_TIME_OUT!=0xFF)
84      {
85          if(nrf24_dataReady()){
86              nrf24_getData(command_buffer);
87              CommandStatus status = DecomposeMessageFromBuffer();
88              if
                  (status==SUCCESFUL_DECOMPOSITION&&lastTargetModuleID==targetModuleI
                  D) {
89                  transmitMessageSync(command_buffer, 32);
90                  return true;
91              }
92          }
93          RF_TIME_OUT++; _delay_ms(2);
94      }
95      return false;
96  }
97
98  void processReceivedLine(){
99      commandAvailable = false;
100
101     CommandStatus status = DecomposeMessageFromBuffer();
```

```
102     if(status==SUCCESFUL_DECOMPOSITION) {
103         if (lastTargetModuleID==MAIN_MODULE){
104             //Executed by main module
105             HandleAvailableCommand();
106         } else {
107             //Retransmitted to other module
108 
109             RF_TransmissionStatus RF_Status = RetransmissionToModule();
110 
111             //Catch module reply
112 
113             //bool didModuleRelpy = catchModuleReply();
114 
115             // Send RF STATUS
116             switch (RF_Status) {
117                 case RF_UNREACHEABLE_MODULE:
118                 writeParameterValue(0, &(uint8_t){RETRANSMISSION_FAILED}, 1);
119                 break;
120                 case RF_ACKNOWLEDGE_FAILED:
121                 writeParameterValue(0, &(uint8_t){RETRANSMISSION_FAILED}, 1);
122                 break;
123                 case RF_SUCCESFUL_TRANSMISSION:
124                 writeParameterValue(0, &(uint8_t){SUCCESFUL_RETRANSMISSION}, 1);
125                 break;
126             }
127             ComposeMessageToBuffer(MESSAGE_STATUS_ID, 1, PHONE_MODULE);
128             transmitMessageSync(command_buffer, 32);
129 
130 
131         }
132     }else {
133 }
134 
135 
136 }
137 
138 void disableUART(){
139     UCSR0B &=~(1<<TXEN0) &~(1<<TXCIE0);
140     UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);
141 }
142 
143 ISR(USART_TX_vect){
144     if (uartBufferPos!=uartTxMessageEnd){
145         UDR0 = *uartBufferPos;
146         uartBufferPos++;
147     }
148 }
149 
150 ISR(USART_RX_vect){
151     if(uartBufferPos!=(command_buffer+uartBufferSize)) {
152         *uartBufferPos=UDR0;
153         if ((*uartBufferPos==ETB)&&(DecomposeMessageFromBuffer()
```

```
            ==SUCCESFUL_DECOMPOSITION)) {
154             disableUART(); commandAvailable = true;
155         }
156         else if(*uartBufferPos==uartCarriageReturnChar) {
157
158             bool hasToReturnCarriage = true;
159             uint8_t* savedUartBufferPos = uartBufferPos+1;
160
161             for (uint8_t x = 1; x < 4; x++) {
162                 if ((uartBufferPos-x)<command_buffer) uartBufferPos =
                      command_buffer+(uartBufferSize-1);
163                 if (*(uartBufferPos-x)!=uartCarriageReturnChar)
                      { hasToReturnCarriage = false; break; }
164             }
165             if (hasToReturnCarriage) {
166                  uartBufferPos = command_buffer;
167
168             } else {
169                 uartBufferPos = savedUartBufferPos;
170             }
171
172         } else {
173             uartBufferPos++;
174         }
175
176     } else {
177         uartBufferPos = command_buffer;
178         *uartBufferPos=UDR0;
179     }
180 }
```

```
1
2
3  #ifndef UART_BLUETOOTH_H_
4  #define UART_BLUETOOTH_H_
5
6
7  #include <stdbool.h>
8  #include <stdint.h>
9
10 #ifndef F_CPU
11 #define F_CPU           16000000UL
12 #endif
13
14 #ifndef BAUD
15 #define BAUD            9600
16 #endif
17
18 #ifndef BRC
19 #define BRC             F_CPU/8/BAUD-1
20 #endif
21
22 #ifndef nullptr
23 #define nullptr         nullptr ((void*)0)
24 #endif
25
26 #define uartBufferSize          32
27 #define uartEndMsgChar          '$'
28 #define uartCarriageReturnChar  0x7F
29
30 #ifndef BIT_MANIPULATION_MACRO
31 #define BIT_MANIPULATION_MACRO 1
32 #define bit_get(p,m) ((p) & (m))
33 #define bit_set(p,m) ((p) |= (m))
34 #define bit_clear(p,m) ((p) &= ~(m))
35 #define bit_flip(p,m) ((p) ^= (m))
36 #define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
37 #define BIT(x) (0x01 << (x))
38 #define LONGBIT(x) ((unsigned long)0x00000001 << (x))
39 #endif
40
41
42 extern bool commandAvailable;
43
44 extern void initBluetoothUart();
45 extern void transmitMessage(uint8_t* message, uint8_t length);
46 extern void transmitMessageSync(uint8_t* message, uint8_t length);
47 extern bool transmissionState();
48 extern void setupReceiveMode();
49 extern void processReceivedLine();
50 extern void disableUART();
51
52
```

```
53
54  #endif /* UART_BLUETOOTH_H_ */
```

```c
/**
 * \file
 * Functions and types for CRC checks.
 *
 * Generated on Wed Sep 11 13:55:53 2019
 * by pycrc v0.9.2, https://pycrc.org
 * using the configuration:
 *  - Width         = 8
 *  - Poly          = 0x07
 *  - XorIn         = 0x00
 *  - ReflectIn     = False
 *  - XorOut        = 0x00
 *  - ReflectOut    = False
 *  - Algorithm     = bit-by-bit-fast
 */
#include "crc.h"     /* include the header file generated with pycrc */
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>



crc_t crc_update(crc_t crc, const void *data, size_t data_len)
{
    const unsigned char *d = (const unsigned char *)data;
    unsigned int i;
    bool bit;
    unsigned char c;

    while (data_len--) {
        c = *d++;
        for (i = 0x80; i > 0; i >>= 1) {
            bit = crc & 0x80;
            if (c & i) {
                bit = !bit;
            }
            crc <<= 1;
            if (bit) {
                crc ^= 0x07;
            }
        }
        crc &= 0xff;
    }
    return crc & 0xff;
}

```

```c
/**
 * \file
 * Functions and types for CRC checks.
 *
 * Generated on Wed Sep 11 13:56:48 2019
 * by pycrc v0.9.2, https://pycrc.org
 * using the configuration:
 *  - Width         = 8
 *  - Poly          = 0x07
 *  - XorIn         = 0x00
 *  - ReflectIn     = False
 *  - XorOut        = 0x00
 *  - ReflectOut    = False
 *  - Algorithm     = bit-by-bit-fast
 *
 * This file defines the functions crc_init(), crc_update() and crc_finalize().
 *
 * The crc_init() function returns the inital \c crc value and must be called
 * before the first call to crc_update().
 * Similarly, the crc_finalize() function must be called after the last call
 * to crc_update(), before the \c crc is being used.
 * is being used.
 *
 * The crc_update() function can be called any number of times (including zero
 * times) in between the crc_init() and crc_finalize() calls.
 *
 * This pseudo-code shows an example usage of the API:
 * \code{.c}
 * crc_t crc;
 * unsigned char data[MAX_DATA_LEN];
 * size_t data_len;
 *
 * crc = crc_init();
 * while ((data_len = read_data(data, MAX_DATA_LEN)) > 0) {
 *     crc = crc_update(crc, data, data_len);
 * }
 * crc = crc_finalize(crc);
 * \endcode
 */
#ifndef CRC_H
#define CRC_H

#include <stdlib.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif


/**
 * The definition of the used algorithm.
```

```
53   *
54   * This is not used anywhere in the generated code, but it may be used by the
55   * application code to call algorithm-specific code, if desired.
56   */
57  #define CRC_ALGO_BIT_BY_BIT_FAST 1
58
59
60  /**
61   * The type of the CRC values.
62   *
63   * This type must be big enough to contain at least 8 bits.
64   */
65  typedef uint_fast8_t crc_t;
66
67
68  /**
69   * Calculate the initial crc value.
70   *
71   * \return     The initial crc value.
72   */
73  static inline crc_t crc_init(void)
74  {
75      return 0x00;
76  }
77
78
79  /**
80   * Update the crc value with new data.
81   *
82   * \param[in] crc      The current crc value.
83   * \param[in] data     Pointer to a buffer of \a data_len bytes.
84   * \param[in] data_len Number of bytes in the \a data buffer.
85   * \return             The updated crc value.
86   */
87  crc_t crc_update(crc_t crc, const void *data, size_t data_len);
88
89
90  /**
91   * Calculate the final crc value.
92   *
93   * \param[in] crc  The current crc value.
94   * \return     The final crc value.
95   */
96  static inline crc_t crc_finalize(crc_t crc)
97  {
98      return crc;
99  }
100
101
102 #ifdef __cplusplus
103 }           /* closing brace for extern "C" */
104 #endif
```

```
105
106 #endif      /* CRC_H */
107
```

```
1
2  #define UCPHA0 1
3
4  #include "nrf24.h"
5  #include "UART_Bluetooth.h"
6
7  volatile uint8_t payload_len;
8  volatile uint8_t selectedChannel;
9
10 uint8_t MOTORIZED_BOARD_ADDR[5] =   {0xF0,0xF0,0xF0,0xF0,0xC9};
11 uint8_t MAIN_BOARD_ADDR[5] =           {0xA4,0xA4,0xA4,0xA4,0xA4};
12 uint8_t POWER_BOARD_ADDR[5] =       {0xF0,0xF0,0xF0,0xF0,0xF0};
13
14 uint8_t NULL_ADDR[5] =      {0x00,0x00,0x00,0x00,0x00};
15
16 uint8_t* BOARD_ADDRESS[3] = {&MAIN_BOARD_ADDR[0], &POWER_BOARD_ADDR[0],
     &MOTORIZED_BOARD_ADDR[0]};
17 uint8_t* CURRENT_BOARD_ADDRESS = &MAIN_BOARD_ADDR[0];
18
19 const uint8_t GENERAL_RF_CHANNEL = 112;
20
21
22 void nrf24_init()
23 {
24     nrf24_setupPins();
25     nrf24_ce_digitalWrite(LOW);
26     nrf24_csn_digitalWrite(HIGH);
27 }
28
29 void nrf24_config(uint8_t channel, uint8_t pay_length)
30 {
31     /* Use static payload length ... */
32     payload_len = pay_length;
33     selectedChannel = channel;
34
35     // Set RF channel
36     nrf24_configRegister(RF_CH,channel);
37
38     // Set length of incoming payload
39     nrf24_configRegister(RX_PW_P0, 0x00); // Auto-ACK pipe ...
40     nrf24_configRegister(RX_PW_P1, payload_len); // Data payload pipe
41     nrf24_configRegister(RX_PW_P2, 0x00); // Pipe not used
42     nrf24_configRegister(RX_PW_P3, 0x00); // Pipe not used
43     nrf24_configRegister(RX_PW_P4, 0x00); // Pipe not used
44     nrf24_configRegister(RX_PW_P5, 0x00); // Pipe not used
45
46     // 1 Mbps, TX gain: 0dbm
47     nrf24_configRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR));
48
49     // CRC enable, 1 byte CRC length
50     nrf24_configRegister(CONFIG,nrf24_CONFIG);
51
```

```
52      // Auto Acknowledgment
53      nrf24_configRegister(EN_AA,(1<<ENAA_P0)|(1<<ENAA_P1)|(0<<ENAA_P2)|
          (0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5));
54
55      // Enable RX addresses
56      nrf24_configRegister(EN_RXADDR,(1<<ERX_P0)|(1<<ERX_P1)|(0<<ERX_P2)|
          (0<<ERX_P3)|(0<<ERX_P4)|(0<<ERX_P5));
57
58      // Auto retransmit delay: 1000 us and Up to 15 retransmit trials
59      nrf24_configRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC));
60
61      // Dynamic length configurations: No dynamic length
62      nrf24_configRegister(DYNPD,(0<<DPL_P0)|(0<<DPL_P1)|(0<<DPL_P2)|(0<<DPL_P3)|
          (0<<DPL_P4)|(0<<DPL_P5));
63
64  }
65
66
67
68  bool nrf24_checkConfig(){
69      // Check all registers
70      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==false) return false;
71      if (nrf24_checkRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR),1)==false)
          return false;
72      if (nrf24_checkRegister(CONFIG,nrf24_CONFIG,1)==false) return false;
73      if (nrf24_checkRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC),1)==false) return
          false;
74      if (nrf24_checkRegister(DYNPD,(0<<DPL_P0)|(0<<DPL_P1)|(0<<DPL_P2)|
          (0<<DPL_P3)|(0<<DPL_P4)|(0<<DPL_P5),1)==false) return false;
75
76      return true;
77  }
78
79  bool nrf24_checkAvailability(){
80      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==true) { return true; }
          else { return false;}
81  }
82
83
84
85
86  void faultyRF_Alarm(){
87      CLEAR_FAULTY_RF_LED;
88      for (uint8_t x = 0; x < 6; x++)
89      {
90          FLIP_FAULTY_RF_LED;
91          _delay_ms(125);
92      }
93      _delay_ms(250);
94  }
95
96
```

```
 97
 98 /* Set the RX address */
 99 void nrf24_rx_address(uint8_t * adr)
100 {
101     nrf24_ce_digitalWrite(LOW);
102     nrf24_writeRegister(RX_ADDR_P1,adr,nrf24_ADDR_LEN);
103     nrf24_ce_digitalWrite(HIGH);
104 }
105
106 /* Set the secondary RX address */
107 void nrf24_secondary_rx_address(uint8_t * adr)
108 {
109     nrf24_ce_digitalWrite(LOW);
110     nrf24_writeRegister(RX_ADDR_P2,adr,1);  // One byte long
111     nrf24_ce_digitalWrite(HIGH);
112 }
113
114
115 /* Returns the payload length */
116 uint8_t nrf24_payload_length()
117 {
118     return payload_len;
119 }
120
121 /* Set the TX address */
122 void nrf24_tx_address(uint8_t* adr)
123 {
124     /* RX_ADDR_P0 must be set to the sending addr for auto ack to work. */
125     nrf24_writeRegister(RX_ADDR_P0,adr,nrf24_ADDR_LEN);
126     nrf24_writeRegister(TX_ADDR,adr,nrf24_ADDR_LEN);
127 }
128
129 /* Checks if data is available for reading */
130 /* Returns 1 if data is ready ... */
131 uint8_t nrf24_dataReady()
132 {
133     // See note in getData() function - just checking RX_DR isn't good enough
134     uint8_t status = nrf24_getStatus();
135
136     // We can short circuit on RX_DR, but if it's not set, we still need
137     // to check the FIFO for any pending packets
138     if ( status & (1 << RX_DR) )
139     {
140         return 1;
141     }
142
143     return !nrf24_rxFifoEmpty();;
144 }
145
146 /* Checks if receive FIFO is empty or not */
147 uint8_t nrf24_rxFifoEmpty()
148 {
```

```
149     uint8_t fifoStatus;
150
151     nrf24_readRegister(FIFO_STATUS,&fifoStatus,1);
152
153     return (fifoStatus & (1 << RX_EMPTY));
154 }
155
156 /* Returns the length of data waiting in the RX fifo */
157 uint8_t nrf24_payloadLength()
158 {
159     uint8_t status;
160     nrf24_csn_digitalWrite(LOW);
161     spi_transfer(R_RX_PL_WID);
162     status = spi_transfer(0x00);
163     nrf24_csn_digitalWrite(HIGH);
164     return status;
165 }
166
167 /* Reads payload bytes into data array */
168 void nrf24_getData(uint8_t* data)
169 {
170     /* Pull down chip select */
171     nrf24_csn_digitalWrite(LOW);
172
173     /* Send cmd to read rx payload */
174     spi_transfer( R_RX_PAYLOAD );
175
176     /* Read payload */
177     nrf24_transferSync(data,data,payload_len);
178
179     /* Pull up chip select */
180     nrf24_csn_digitalWrite(HIGH);
181
182     /* Reset status register */
183     nrf24_configRegister(STATUS,(1<<RX_DR));
184 }
185
186 /* Returns the number of retransmissions occured for the last message */
187 uint8_t nrf24_retransmissionCount()
188 {
189     uint8_t rv;
190     nrf24_readRegister(OBSERVE_TX,&rv,1);
191     rv = rv & 0x0F;
192     return rv;
193 }
194
195 // Sends a data package to the default address. Be sure to send the correct
196 // amount of bytes as configured as payload on the receiver.
197 void nrf24_send(uint8_t* value)
198 {
199     /* Go to Standby-I first */
200     nrf24_ce_digitalWrite(LOW);
```

```
201
202     /* Set to transmitter mode , Power up if needed */
203     nrf24_powerUpTx();
204
205     /* Do we really need to flush TX fifo each time ? */
206     #if 1
207     /* Pull down chip select */
208     nrf24_csn_digitalWrite(LOW);
209
210     /* Write cmd to flush transmit FIFO */
211     spi_transfer(FLUSH_TX);
212
213     /* Pull up chip select */
214     nrf24_csn_digitalWrite(HIGH);
215     #endif
216
217     /* Pull down chip select */
218     nrf24_csn_digitalWrite(LOW);
219
220     /* Write cmd to write payload */
221     spi_transfer(W_TX_PAYLOAD);
222
223     /* Write payload */
224     nrf24_transmitSync(value,payload_len);
225
226     /* Pull up chip select */
227     nrf24_csn_digitalWrite(HIGH);
228
229     /* Start the transmission */
230     nrf24_ce_digitalWrite(HIGH);
231 }
232
233 uint8_t nrf24_isSending()
234 {
235     uint8_t status;
236
237     /* read the current status */
238     status = nrf24_getStatus();
239
240     /* if sending successful (TX_DS) or max retries exceded (MAX_RT). */
241     if((status & ((1 << TX_DS)  | (1 << MAX_RT))))
242     {
243         return 0; /* false */
244     }
245
246     return 1; /* true */
247
248 }
249
250 uint8_t nrf24_getStatus()
251 {
252     uint8_t rv;
```

```
253     nrf24_csn_digitalWrite(LOW);
254     rv = spi_transfer(NOP);
255     nrf24_csn_digitalWrite(HIGH);
256     return rv;
257 }
258 
259 uint8_t nrf24_lastMessageStatus()
260 {
261     uint8_t rv;
262 
263     rv = nrf24_getStatus();
264 
265     /* Transmission went OK */
266     if((rv & ((1 << TX_DS))))
267     {
268         return NRF24_TRANSMISSON_OK;
269     }
270     /* Maximum retransmission count is reached */
271     /* Last message probably went missing ... */
272     else if((rv & ((1 << MAX_RT))))
273     {
274         return NRF24_MESSAGE_LOST;
275     }
276     /* Probably still sending ... */
277     else
278     {
279         return 0xFF;
280     }
281 }
282 
283 void nrf24_powerUpRx()
284 {
285     nrf24_csn_digitalWrite(LOW);
286     spi_transfer(FLUSH_RX);
287     nrf24_csn_digitalWrite(HIGH);
288 
289     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
290 
291     nrf24_ce_digitalWrite(LOW);
292     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(1<<PRIM_RX)));
293     nrf24_ce_digitalWrite(HIGH);
294 }
295 
296 void nrf24_powerUpTx()
297 {
298     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
299 
300     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(0<<PRIM_RX)));
301 }
302 
303 void nrf24_powerDown()
304 {
```

```
305     nrf24_ce_digitalWrite(LOW);
306     nrf24_configRegister(CONFIG,nrf24_CONFIG);
307 }
308 
309 uint8_t spi_transfer(uint8_t tx)
310 {
311     uint8_t i = 0;
312     uint8_t rx = 0;
313 
314     nrf24_sck_digitalWrite(LOW);
315 
316     for(i=0;i<8;i++)
317     {
318 
319         if(tx & (1<<(7-i)))
320         {
321             nrf24_mosi_digitalWrite(HIGH);
322         }
323         else
324         {
325             nrf24_mosi_digitalWrite(LOW);
326         }
327 
328         nrf24_sck_digitalWrite(HIGH);
329 
330         rx = rx << 1;
331         if(nrf24_miso_digitalRead())
332         {
333             rx |= 0x01;
334         }
335 
336         nrf24_sck_digitalWrite(LOW);
337 
338     }
339 
340     return rx;
341 }
342 
343 /* send and receive multiple bytes over SPI */
344 void nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len)
345 {
346     uint8_t i;
347 
348     for(i=0;i<len;i++)
349     {
350         datain[i] = spi_transfer(dataout[i]);
351     }
352 
353 }
354 
355 /* send multiple bytes over SPI */
356 void nrf24_transmitSync(uint8_t* dataout,uint8_t len)
```

```
357 {
358     uint8_t i;
359 
360     for(i=0;i<len;i++)
361     {
362         spi_transfer(dataout[i]);
363     }
364 
365 }
366 
367 /* Clocks only one byte into the given nrf24 register */
368 void nrf24_configRegister(uint8_t reg, uint8_t value)
369 {
370     nrf24_csn_digitalWrite(LOW);
371     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
372     spi_transfer(value);
373     nrf24_csn_digitalWrite(HIGH);
374 }
375 
376 /* Read single register from nrf24 */
377 void nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len)
378 {
379     nrf24_csn_digitalWrite(LOW);
380     spi_transfer(R_REGISTER | (REGISTER_MASK & reg));
381     nrf24_transferSync(value,value,len);
382     nrf24_csn_digitalWrite(HIGH);
383 }
384 
385 /* Write to a single register of nrf24 */
386 void nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len)
387 {
388     nrf24_csn_digitalWrite(LOW);
389     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
390     nrf24_transmitSync(value,len);
391     nrf24_csn_digitalWrite(HIGH);
392 }
393 
394 /* Check single register from nrf24 */
395 bool nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len)
396 {
397     uint8_t registerValue;
398     nrf24_readRegister(reg,&registerValue,len);
399     if (registerValue==desiredValue) { return true; } else { return false; }
400 }
401 
402 #define RF_DDR  DDRC
403 #define RF_PORT PORTC
404 #define RF_PIN  PINC
405 
406 #define set_bit(reg,bit) reg |= (1<<bit)
407 #define clr_bit(reg,bit) reg &= ~(1<<bit)
408 #define check_bit(reg,bit) (reg&(1<<bit))
```

```
409
410 /* ---------------------------------------------------------------------- */
411
412 void nrf24_setupPins()
413 {
414     set_bit(RF_DDR,0); // CE output
415     set_bit(RF_DDR,1); // CSN output
416     set_bit(RF_DDR,2); // SCK output
417     set_bit(RF_DDR,3); // MOSI output
418     clr_bit(RF_DDR,4); // MISO input
419 }
420 /* ---------------------------------------------------------------------- */
421 void nrf24_ce_digitalWrite(uint8_t state)
422 {
423     if(state)
424     {
425         set_bit(RF_PORT,0);
426     }
427     else
428     {
429         clr_bit(RF_PORT,0);
430     }
431 }
432 /* ---------------------------------------------------------------------- */
433 void nrf24_csn_digitalWrite(uint8_t state)
434 {
435     if(state)
436     {
437         set_bit(RF_PORT,1);
438     }
439     else
440     {
441         clr_bit(RF_PORT,1);
442     }
443 }
444 /* ---------------------------------------------------------------------- */
445 void nrf24_sck_digitalWrite(uint8_t state)
446 {
447     if(state)
448     {
449         set_bit(RF_PORT,2);
450     }
451     else
452     {
453         clr_bit(RF_PORT,2);
454     }
455 }
456 /* ---------------------------------------------------------------------- */
457 void nrf24_mosi_digitalWrite(uint8_t state)
458 {
459     if(state)
460     {
```

```
461         set_bit(RF_PORT,3);
462     }
463     else
464     {
465         clr_bit(RF_PORT,3);
466     }
467 }
468 /* ------------------------------------------------------------------------ */
469 uint8_t nrf24_miso_digitalRead()
470 {
471     return check_bit(RF_PIN,4);
472 }
473 /* ------------------------------------------------------------------------ */
474
475 void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode){
476
477     initliazeMemory();
478     bool successfulRfInit = false;
479
480     while(successfulRfInit==false){
481         nrf24_powerDown();
482         nrf24_init();
483         nrf24_config(GENERAL_RF_CHANNEL,32);
484         if (nrf24_checkConfig()) { successfulRfInit = true; } else
              { faultyRF_Alarm(); }
485     }
486
487
488
489     if (initMode==RECEIVE){
490         nrf24_tx_address(CURRENT_BOARD_ADDRESS);
491         nrf24_rx_address(BOARD_ADDRESS[boardIndex]);
492         }else{
493         nrf24_tx_address(BOARD_ADDRESS[boardIndex]);
494         nrf24_rx_address(CURRENT_BOARD_ADDRESS);
495     }
496
497
498     nrf24_powerUpRx();
499 }
```

```
#ifndef NRF24
#define NRF24

#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include "nRF24L01_Definitions.h"
#include "Command_Handler.h"
#include <stdint.h>
#include <stdbool.h>
#include <avr/io.h>
#include <avr/delay.h>



#ifndef BIT_MANIPULATION_MACRO
#define BIT_MANIPULATION_MACRO 1
#define bit_get(p,m) ((p) & (m))
#define bit_set(p,m) ((p) |= (m))
#define bit_clear(p,m) ((p) &= ~(m))
#define bit_flip(p,m) ((p) ^= (m))
#define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
#define BIT(x) (0x01 << (x))
#define LONGBIT(x) ((unsigned long)0x00000001 << (x))
#endif

#define LOW 0
#define HIGH 1
#define nrf24_ADDR_LEN 5
#define nrf24_CONFIG ((1<<EN_CRC)|(0<<CRCO))
#define NRF24_TRANSMISSON_OK 0
#define NRF24_MESSAGE_LOST   1

#define CLEAR_FAULTY_RF_LED         bit_clear(PORTD, BIT(7))
#define FLIP_FAULTY_RF_LED          bit_flip(PORTD, BIT(7))


enum TransmissionMode {
    RECEIVE,
    TRANSMIT
};
typedef enum TransmissionMode TransmissionMode;

enum CommandsBoard {
    MAIN_BOARD_RF = 0,
    POWER_BOARD_RF = 1,
    MOTORIZED_BOARD_RF = 2
};
typedef enum CommandsBoard CommandsBoard;

extern void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode);
```

```
53
54  void    nrf24_init();
55  void    nrf24_rx_address(uint8_t* adr);
56  void    nrf24_tx_address(uint8_t* adr);
57  void    nrf24_config(uint8_t channel, uint8_t pay_length);
58  bool    nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len);
59  bool    nrf24_checkConfig();
60  bool    nrf24_checkAvailability();
61
62  void faultyRF_Alarm();
63
64  uint8_t selectedTX_ADDRESS;
65  uint8_t selectedRX_ADDRESS;
66
67  uint8_t nrf24_dataReady();
68  uint8_t nrf24_isSending();
69  uint8_t nrf24_getStatus();
70  uint8_t nrf24_rxFifoEmpty();
71
72  void    nrf24_send(uint8_t* value);
73  void    nrf24_getData(uint8_t* data);
74
75  uint8_t nrf24_payloadLength();
76
77  uint8_t nrf24_lastMessageStatus();
78  uint8_t nrf24_retransmissionCount();
79
80  uint8_t nrf24_payload_length();
81
82  void    nrf24_powerUpRx();
83  void    nrf24_powerUpTx();
84  void    nrf24_powerDown();
85
86  uint8_t spi_transfer(uint8_t tx);
87  void    nrf24_transmitSync(uint8_t* dataout,uint8_t len);
88  void    nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len);
89  void    nrf24_configRegister(uint8_t reg, uint8_t value);
90  void    nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len);
91  void    nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len);
92
93  extern void nrf24_setupPins();
94
95  extern void nrf24_ce_digitalWrite(uint8_t state);
96
97  extern void nrf24_csn_digitalWrite(uint8_t state);
98
99  extern void nrf24_sck_digitalWrite(uint8_t state);
100
101 extern void nrf24_mosi_digitalWrite(uint8_t state);
102
103 extern uint8_t nrf24_miso_digitalRead();
104
```

```
105  #endif
106
```