```c
#ifndef F_CPU
#define F_CPU 16000000UL
#endif
#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 "nrf24.h"

void initIO();

int main(void)
{
    //sei();    // Interrupts on
    initIO();
    nrf24_initRF_SAFE(MAIN_BOARD, RECEIVE); // CONNECTION TO MAIN BOARD : GENERAL RF CHANNEL 112

    while (1)
    {
        if(nrf24_dataReady())
        {
            nrf24_getData(command_buffer);
            CommandStatus status = DecomposeMessageFromBuffer();
            if (status==SUCCESFUL_DECOMPOSITION) { HandleAvailableCommand(); } else
            {
                bit_flip(PORTD, BIT(7)); _delay_ms(250); bit_flip(PORTD, BIT(7));
            }
        }

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


void initIO(){
    /*
        Input/Output pin initialization
        1 : OUTPUT | 0 : INPUT | 0b76543210 Bit order
        ATTACHMENTS
            RELAY 0     : PD3               |   OUTPUT
            RELAY 1     : PD2               |   OUTPUT
            RELAY 2     : PD6               |   OUTPUT
            RELAY 3     : PD5               |   OUTPUT
            RED LED     : PD7               |   OUTPUT
            GREEN LED   : PB0               |   OUTPUT
```

```
50          nRF24L01
51              CE   : PC0                       |   OUTPUT
52              CSN  : PC1                       |   OUTPUT
53              MISO : PD0 (MSPIM MISO ATMEGA)   |   INPUT
54              MOSI : PD1 (MSPIM MOSI ATMEGA)   |   OUTPUT
55              SCK  : PD4 (MSPIM XCK)           |   OUTPUT
56      */
57      DDRD = 0b11111110;
58      DDRB = 0b00101001;
59      DDRC = 0b11011111;
60  }
61
62
63
64
65
66
```

```c

#include "Command_Handler.h"
#include "nrf24.h"
#include "crc.h"



const CommandType commandList[] = {
    { .handlerFunction = &UPDATE_ALL_DEVICES_VALUE_H},
    { .handlerFunction = &UPDATE_DEVICE_VALUE_H},
    { .handlerFunction = &GET_ALL_DEVICES_VALUE_H},
    { .handlerFunction = &GET_DEVICE_VALUE_H},
    { .handlerFunction = &MESSAGE_STATUS_H}
};
#define commandListLength (uint8_t)(sizeof commandList/sizeof commandList[0])

bool initliazeMemory(){
    if(memoryInitialized) return false;
    parameter[0].startingPointer = (void*)calloc(23,1);
    parameter[1].startingPointer = (void*)calloc(2,1);
    parameter[2].startingPointer = (void*)calloc(2,1);
    for (uint8_t x = 3; x<12; x++) parameter[x].startingPointer = (void*)calloc(1,1);
    command_buffer = (uint8_t*)calloc(32,1);
    if(command_buffer==NULL) return false;
    for (uint8_t x = 0; x<12; x++) { if(parameter[x].startingPointer==NULL) return false; }
    memoryInitialized = true;
    return true;
}

CommandStatus DecomposeMessageFromBuffer(){
    // Search for header
    uint8_t* headerStart = command_buffer;
    uint8_t* footerEnd = command_buffer+31;

    for(;headerStart!=(command_buffer+22);headerStart++){
        if (*headerStart==SOH&&(*(headerStart+4)==STX)){
            for(;footerEnd!=(command_buffer+6);footerEnd--){
                if (*footerEnd==ETB&&(*(footerEnd-2)==ETX)){
                    uint8_t netMessageLength = ((footerEnd-2)-headerStart);
                    crc_t crc;
                    crc = crc_init();
                    crc = crc_update(crc, headerStart, netMessageLength);
                    crc = crc_finalize(crc);
                    if (*(footerEnd-1)!=crc) return WRONG_CHECKSUM_CONSISTENCY;
                    if (*(headerStart+2)!=currentModuleID&&*(headerStart+2)!=0xFF&&currentModuleID!=0x01) return WRONG_MODULE_ID;
                    lastTargetModuleID = *(headerStart+2);
                    lastTransmitterModuleID = *(headerStart+3);
                    if (*(headerStart+5)>commandListLength-1) return UNDEFINED_COMMAND_CODE;
```

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

```
 98      *(parameterStart+2) = ETB;
 99
100      return SUCCESFUL_COMPOSITION;
101  }
102
103  void HandleAvailableCommand(){
104      lastMessageCommandType.handlerFunction();
105  }
106
107  RF_TransmissionStatus RetransmissionToModule(){
108      nrf24_initRF_SAFE(lastTargetModuleID, TRANSMIT);    // CONNECTION TO MODULE:
           GENERAL RF CHANNEL 112
109      nrf24_send(command_buffer);
110      while(nrf24_isSending());
111
112      uint8_t messageStatus = nrf24_lastMessageStatus();
113      if(messageStatus == NRF24_TRANSMISSON_OK) { return
           RF_SUCCESFUL_TRANSMISSION; }
114      else if(messageStatus == NRF24_MESSAGE_LOST) { return
           RF_UNREACHEABLE_MODULE;}
115      return RF_UNREACHEABLE_MODULE;
116  }
117
118  void writeParameterValue(uint8_t parameterIndex, void* parameterData, uint8_t
       parameterByteLength){
119      parameter[parameterIndex].startingPointer = (uint8_t*) realloc(parameter
           [parameterIndex].startingPointer, parameterByteLength);
120      memcpy(parameter[parameterIndex].startingPointer, parameterData,
           parameterByteLength);
121      parameter[parameterIndex].byteLength = parameterByteLength;
122  }
123
124  void UPDATE_ALL_DEVICES_VALUE_H() {
125      for (uint8_t x = 0; x < AVAILABLE_DEVICES;x++)
126      {
127          deviceStoredValue[x] = *((uint8_t*)parameter[x].startingPointer);
128          switch (x) {
129              case 0x00:
130                  bit_write(deviceStoredValue[x], PORTD, BIT(3));
131                  break;
132              case 0x01:
133                  bit_write(deviceStoredValue[x], PORTD, BIT(2));
134                  break;
135              case 0x02:
136                  bit_write(deviceStoredValue[x], PORTD, BIT(6));
137                  break;
138              case 0x03:
139                  bit_write(deviceStoredValue[x], PORTD, BIT(5));
140                  break;
141          }
142      }
143
```

```
144
145
146 }
147
148 void UPDATE_DEVICE_VALUE_H() {
149     uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
150     uint8_t deviceValue = *((uint8_t*)parameter[1].startingPointer);
151
152     switch (deviceIndex) {
153         case 0:
154             bit_write(deviceValue, PORTD, BIT(3));
155             break;
156         case 1:
157             bit_write(deviceValue, PORTD, BIT(2));
158             break;
159         case 2:
160             bit_write(deviceValue, PORTD, BIT(6));
161             break;
162         case 3:
163             bit_write(deviceValue, PORTD, BIT(5));
164             break;
165     }
166
167     deviceStoredValue[deviceIndex] = deviceValue;
168
169 }
170 void GET_ALL_DEVICES_VALUE_H() {
171     _delay_ms(50);
172
173     for (uint8_t x = 0; x < AVAILABLE_DEVICES;x++)
174     {
175         writeParameterValue(x, &deviceStoredValue[x], 2);
176     }
177
178     ComposeMessageToBuffer(UPDATE_ALL_DEVICES_VALUE_ID, AVAILABLE_DEVICES,
        PHONE_MODULE);  // PHONE_MODULE deberia ser lastTransmitterModuleID
179
180     nrf24_initRF_SAFE(MAIN_BOARD, TRANSMIT);
181     nrf24_send(command_buffer);
182     while(nrf24_isSending());
183     uint8_t messageStatus = nrf24_lastMessageStatus();
184 }
185 void GET_DEVICE_VALUE_H() {
186     _delay_ms(50);
187     uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
188     writeParameterValue(0, &deviceIndex, 1);
189     writeParameterValue(1, &deviceStoredValue[deviceIndex], 2);
190     ComposeMessageToBuffer(UPDATE_DEVICE_VALUE_ID, 2, PHONE_MODULE);  //
        PHONE_MODULE deberia ser lastTransmitterModuleID
191
192     nrf24_initRF_SAFE(MAIN_BOARD, TRANSMIT);
193     nrf24_send(command_buffer);
```

```
194     while(nrf24_isSending());
195     uint8_t messageStatus = nrf24_lastMessageStatus();
196 }
197
198
199 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 POWER_MODULE
80
81
82  #define SOH 0x01
83  #define STX 0x02
84  #define ETX 0x03
85  #define ETB 0x17
86  #define ON_STATE    0xFF
87  #define OFF_STATE   0x00
88
89  #define AVAILABLE_DEVICES 4
90  uint8_t deviceStoredValue[AVAILABLE_DEVICES];
91
92  uint8_t *command_buffer;
93  Parameter parameter[12];
94  bool memoryInitialized;
95
96  uint8_t lastMessagePID;
97  uint8_t lastTargetModuleID;
98  uint8_t lastTransmitterModuleID;
99  CommandType lastMessageCommandType;
100
101 extern bool initliazeMemory();
102 extern void UPDATE_ALL_DEVICES_VALUE_H(), UPDATE_DEVICE_VALUE_H(),
      GET_ALL_DEVICES_VALUE_H(), GET_DEVICE_VALUE_H(), MESSAGE_STATUS_H();
103 extern CommandStatus DecomposeMessageFromBuffer();
```

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

```
1
2  #define UCPHA0 1
3  #define BAUD_RATE 38400UL
4  #define UBRR_VALUE ((F_CPU)/(2UL*BAUD_RATE))-1
5
6  #include "nrf24.h"
7
8  uint8_t payload_len;
9  uint8_t selectedChannel;
10
11 uint8_t MOTORIZED_BOARD_ADDR[5] =   {0xF0,0xF0,0xF0,0xF0,0xC9};
12 uint8_t MAIN_BOARD_ADDR[5] =           {0xA4,0xA4,0xA4,0xA4,0xA4};
13 uint8_t POWER_BOARD_ADDR[5] =       {0xF0,0xF0,0xF0,0xF0,0xF0};
14
15 uint8_t* BOARD_ADDRESS[3] = {&MAIN_BOARD_ADDR[0], &POWER_BOARD_ADDR[0],
     &MOTORIZED_BOARD_ADDR[0]};
16 uint8_t* CURRENT_BOARD_ADDRESS = &POWER_BOARD_ADDR[0];
17
18 uint8_t GENERAL_RF_CHANNEL = 112;
19
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     // Set RF channel
35     nrf24_configRegister(RF_CH,channel);
36     // Set length of incoming payload
37     nrf24_configRegister(RX_PW_P0, 0x00); // Auto-ACK pipe ...
38     nrf24_configRegister(RX_PW_P1, payload_len); // Data payload pipe
39     nrf24_configRegister(RX_PW_P2, 0x00); // Pipe not used
40     nrf24_configRegister(RX_PW_P3, 0x00); // Pipe not used
41     nrf24_configRegister(RX_PW_P4, 0x00); // Pipe not used
42     nrf24_configRegister(RX_PW_P5, 0x00); // Pipe not used
43     // 1 Mbps, TX gain: 0dbm
44     nrf24_configRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR));
45     // CRC enable, 1 byte CRC length
46     nrf24_configRegister(CONFIG,nrf24_CONFIG);
47     // Auto Acknowledgment
48     nrf24_configRegister(EN_AA,(1<<ENAA_P0)|(1<<ENAA_P1)|(0<<ENAA_P2)|
         (0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5));
49     // Enable RX addresses
50     nrf24_configRegister(EN_RXADDR,(1<<ERX_P0)|(1<<ERX_P1)|(0<<ERX_P2)|
```

```
          (0<<ERX_P3)|(0<<ERX_P4)|(0<<ERX_P5));
51      // Auto retransmit delay: 1000 us and Up to 15 retransmit trials
52      nrf24_configRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC));
53      // Dynamic length configurations: No dynamic length
54      nrf24_configRegister(DYNPD,(0<<DPL_P0)|(0<<DPL_P1)|(0<<DPL_P2)|(0<<DPL_P3)|
          (0<<DPL_P4)|(0<<DPL_P5));
55
56  }
57
58  bool nrf24_checkConfig(){
59      // Check all registers
60      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==false) return false;
61      if (nrf24_checkRegister(RX_PW_P0, 0x00,1)==false) return false;
62      if (nrf24_checkRegister(RX_PW_P1, payload_len,1)==false) return false;
63      if (nrf24_checkRegister(RX_PW_P2, 0x00,1)==false) return false;
64      if (nrf24_checkRegister(RX_PW_P3, 0x00,1)==false) return false;
65      if (nrf24_checkRegister(RX_PW_P4, 0x00,1)==false) return false;
66      if (nrf24_checkRegister(RX_PW_P5, 0x00,1)==false) return false;
67      if (nrf24_checkRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR),1)==false)
          return false;
68      if (nrf24_checkRegister(CONFIG,nrf24_CONFIG,1)==false) return false;
69      if (nrf24_checkRegister(EN_AA,(1<<ENAA_P0)|(1<<ENAA_P1)|(0<<ENAA_P2)|
          (0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5),1)==false) return false;
70      if (nrf24_checkRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC),1)==false) return
          false;
71      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;
72
73      return true;
74  }
75
76  bool nrf24_checkAvailability(){
77      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==true) { return true; }
          else { return false;}
78  }
79
80
81
82
83  void faultyRF_Alarm(){
84      CLEAR_FAULTY_RF_LED;
85      for (uint8_t x = 0; x < 6; x++)
86      {
87          FLIP_FAULTY_RF_LED;
88          _delay_ms(125);
89      }
90      _delay_ms(250);
91  }
92
93
94
95  /* Set the RX address */
```

```
 96  void nrf24_rx_address(uint8_t * adr)
 97  {
 98      nrf24_ce_digitalWrite(LOW);
 99      nrf24_writeRegister(RX_ADDR_P1,adr,nrf24_ADDR_LEN);
100      nrf24_ce_digitalWrite(HIGH);
101  }
102
103  /* Returns the payload length */
104  uint8_t nrf24_payload_length()
105  {
106      return payload_len;
107  }
108
109  /* Set the TX address */
110  void nrf24_tx_address(uint8_t* adr)
111  {
112      /* RX_ADDR_P0 must be set to the sending addr for auto ack to work. */
113      nrf24_writeRegister(RX_ADDR_P0,adr,nrf24_ADDR_LEN);
114      nrf24_writeRegister(TX_ADDR,adr,nrf24_ADDR_LEN);
115  }
116
117  /* Checks if data is available for reading */
118  /* Returns 1 if data is ready ... */
119  uint8_t nrf24_dataReady()
120  {
121      // See note in getData() function - just checking RX_DR isn't good enough
122      uint8_t status = nrf24_getStatus();
123
124      // We can short circuit on RX_DR, but if it's not set, we still need
125      // to check the FIFO for any pending packets
126      if ( status & (1 << RX_DR) )
127      {
128          return 1;
129      }
130
131      return !nrf24_rxFifoEmpty();
132  }
133
134  /* Checks if receive FIFO is empty or not */
135  uint8_t nrf24_rxFifoEmpty()
136  {
137      uint8_t fifoStatus;
138
139      nrf24_readRegister(FIFO_STATUS,&fifoStatus,1);
140
141      return (fifoStatus & (1 << RX_EMPTY));
142  }
143
144  /* Returns the length of data waiting in the RX fifo */
145  uint8_t nrf24_payloadLength()
146  {
147      uint8_t status;
```

```
148     nrf24_csn_digitalWrite(LOW);
149     spi_transfer(R_RX_PL_WID);
150     status = spi_transfer(0x00);
151     nrf24_csn_digitalWrite(HIGH);
152     return status;
153 }
154 
155 /* Reads payload bytes into data array */
156 void nrf24_getData(uint8_t* data)
157 {
158     /* Pull down chip select */
159     nrf24_csn_digitalWrite(LOW);
160 
161     /* Send cmd to read rx payload */
162     spi_transfer( R_RX_PAYLOAD );
163 
164     /* Read payload */
165     nrf24_transferSync(data,data,payload_len);
166 
167     /* Pull up chip select */
168     nrf24_csn_digitalWrite(HIGH);
169 
170     /* Reset status register */
171     nrf24_configRegister(STATUS,(1<<RX_DR));
172 }
173 
174 /* Returns the number of retransmissions occured for the last message */
175 uint8_t nrf24_retransmissionCount()
176 {
177     uint8_t rv;
178     nrf24_readRegister(OBSERVE_TX,&rv,1);
179     rv = rv & 0x0F;
180     return rv;
181 }
182 
183 // Sends a data package to the default address. Be sure to send the correct
184 // amount of bytes as configured as payload on the receiver.
185 void nrf24_send(uint8_t* value)
186 {
187     /* Go to Standby-I first */
188     nrf24_ce_digitalWrite(LOW);
189 
190     /* Set to transmitter mode , Power up if needed */
191     nrf24_powerUpTx();
192 
193     /* Do we really need to flush TX fifo each time ? */
194     #if 1
195         /* Pull down chip select */
196         nrf24_csn_digitalWrite(LOW);
197 
198         /* Write cmd to flush transmit FIFO */
199         spi_transfer(FLUSH_TX);
```

```
200
201         /* Pull up chip select */
202         nrf24_csn_digitalWrite(HIGH);
203     #endif
204
205     /* Pull down chip select */
206     nrf24_csn_digitalWrite(LOW);
207
208     /* Write cmd to write payload */
209     spi_transfer(W_TX_PAYLOAD);
210
211     /* Write payload */
212     nrf24_transmitSync(value,payload_len);
213
214     /* Pull up chip select */
215     nrf24_csn_digitalWrite(HIGH);
216
217     /* Start the transmission */
218     nrf24_ce_digitalWrite(HIGH);
219 }
220
221 uint8_t nrf24_isSending()
222 {
223     uint8_t status;
224
225     /* read the current status */
226     status = nrf24_getStatus();
227
228     /* if sending successful (TX_DS) or max retries exceded (MAX_RT). */
229     if((status & ((1 << TX_DS)  | (1 << MAX_RT))))
230     {
231         return 0; /* false */
232     }
233
234     return 1; /* true */
235 }
236
237 uint8_t nrf24_getStatus()
238 {
239     uint8_t rv;
240     nrf24_csn_digitalWrite(LOW);
241     rv = spi_transfer(NOP);
242     nrf24_csn_digitalWrite(HIGH);
243     return rv;
244 }
245
246 uint8_t nrf24_lastMessageStatus()
247 {
248     uint8_t rv;
249
250     rv = nrf24_getStatus();
251
```

```
252     /* Transmission went OK */
253     if((rv & ((1 << TX_DS))))
254     {
255         return NRF24_TRANSMISSON_OK;
256     }
257     /* Maximum retransmission count is reached */
258     /* Last message probably went missing ... */
259     else if((rv & ((1 << MAX_RT))))
260     {
261         return NRF24_MESSAGE_LOST;
262     }
263     /* Probably still sending ... */
264     else
265     {
266         return 0xFF;
267     }
268 }
269
270 void nrf24_powerUpRx()
271 {
272     nrf24_csn_digitalWrite(LOW);
273     spi_transfer(FLUSH_RX);
274     nrf24_csn_digitalWrite(HIGH);
275
276     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
277
278     nrf24_ce_digitalWrite(LOW);
279     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(1<<PRIM_RX)));
280     nrf24_ce_digitalWrite(HIGH);
281 }
282
283 void nrf24_powerUpTx()
284 {
285     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
286
287     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(0<<PRIM_RX)));
288 }
289
290 void nrf24_powerDown()
291 {
292     nrf24_ce_digitalWrite(LOW);
293     nrf24_configRegister(CONFIG,nrf24_CONFIG);
294 }
295
296 uint8_t spi_transfer(uint8_t tx)
297 {
298     uint8_t i = 0;
299     uint8_t rx = 0;
300
301     nrf24_sck_digitalWrite(LOW);
302
303     for(i=0;i<8;i++)
```

```
304     {
305 
306         if(tx & (1<<(7-i)))
307         {
308             nrf24_mosi_digitalWrite(HIGH);
309         }
310         else
311         {
312             nrf24_mosi_digitalWrite(LOW);
313         }
314 
315         nrf24_sck_digitalWrite(HIGH);
316 
317         rx = rx << 1;
318         if(nrf24_miso_digitalRead())
319         {
320             rx |= 0x01;
321         }
322 
323         nrf24_sck_digitalWrite(LOW);
324 
325     }
326 
327     return rx;
328 }
329 
330 /* send and receive multiple bytes over SPI */
331 void nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len)
332 {
333     uint8_t i;
334 
335     for(i=0;i<len;i++)
336     {
337         datain[i] = spi_transfer(dataout[i]);
338     }
339 
340 }
341 
342 /* send multiple bytes over SPI */
343 void nrf24_transmitSync(uint8_t* dataout,uint8_t len)
344 {
345     uint8_t i;
346 
347     for(i=0;i<len;i++)
348     {
349         spi_transfer(dataout[i]);
350     }
351 
352 }
353 
354 /* Clocks only one byte into the given nrf24 register */
355 void nrf24_configRegister(uint8_t reg, uint8_t value)
```

```
356 {
357     nrf24_csn_digitalWrite(LOW);
358     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
359     spi_transfer(value);
360     nrf24_csn_digitalWrite(HIGH);
361 }
362
363 /* Read single register from nrf24 */
364 void nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len)
365 {
366     nrf24_csn_digitalWrite(LOW);
367     spi_transfer(R_REGISTER | (REGISTER_MASK & reg));
368     nrf24_transferSync(value,value,len);
369     nrf24_csn_digitalWrite(HIGH);
370 }
371
372 /* Write to a single register of nrf24 */
373 void nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len)
374 {
375     nrf24_csn_digitalWrite(LOW);
376     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
377     nrf24_transmitSync(value,len);
378     nrf24_csn_digitalWrite(HIGH);
379 }
380
381 /* Check single register from nrf24 */
382 bool nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len)
383 {
384     uint8_t registerValue;
385     nrf24_readRegister(reg,&registerValue,len);
386     if (registerValue==desiredValue) { return true; } else { return false; }
387 }
388
389 #define RF_DDR  DDRD
390 #define RF_PORT PORTD
391 #define RF_PIN  PIND
392
393 #define CE_CSN_DDR  DDRC
394 #define CE_CSN_PORT PORTC
395 #define CE_CSN_PIN  PINC
396
397 #define MISO_BIT_POS    0
398 #define MOSI_BIT_POS    1
399 #define SCK_BIT_POS     4
400
401 #define CE_BIT_POS      0
402 #define CSN_BIT_POS     1
403
404 #define set_bit(reg,bit) reg |= (1<<bit)
405 #define clr_bit(reg,bit) reg &= ~(1<<bit)
406 #define check_bit(reg,bit) (reg&(1<<bit))
407
```

```
408 /* ------------------------------------------------------------------------ */
409
410 void nrf24_setupPins()
411 {
412     set_bit(CE_CSN_DDR, CE_BIT_POS); // CE output
413     set_bit(CE_CSN_DDR, CSN_BIT_POS); // CSN output
414
415     clr_bit(RF_DDR, MISO_BIT_POS); // MISO input
416     set_bit(RF_DDR, MOSI_BIT_POS); // MOSI output
417     set_bit(RF_DDR, SCK_BIT_POS); // SCK output
418 }
419 /* ------------------------------------------------------------------------ */
420 void nrf24_ce_digitalWrite(uint8_t state)
421 {
422     if(state)
423     {
424         set_bit(CE_CSN_PORT, CE_BIT_POS);
425     }
426     else
427     {
428         clr_bit(CE_CSN_PORT, CE_BIT_POS);
429     }
430 }
431 /* ------------------------------------------------------------------------ */
432 void nrf24_csn_digitalWrite(uint8_t state)
433 {
434     if(state)
435     {
436         set_bit(CE_CSN_PORT, CSN_BIT_POS);
437     }
438     else
439     {
440         clr_bit(CE_CSN_PORT, CSN_BIT_POS);
441     }
442 }
443 /* ------------------------------------------------------------------------ */
444 void nrf24_sck_digitalWrite(uint8_t state)
445 {
446     if(state)
447     {
448         set_bit(RF_PORT, SCK_BIT_POS);
449     }
450     else
451     {
452         clr_bit(RF_PORT, SCK_BIT_POS);
453     }
454 }
455 /* ------------------------------------------------------------------------ */
456 void nrf24_mosi_digitalWrite(uint8_t state)
457 {
458     if(state)
459     {
```

```
460         set_bit(RF_PORT, MOSI_BIT_POS);
461     }
462     else
463     {
464         clr_bit(RF_PORT, MOSI_BIT_POS);
465     }
466 }
467 /* ------------------------------------------------------------------------ */
468 uint8_t nrf24_miso_digitalRead()
469 {
470     return check_bit(RF_PIN, MISO_BIT_POS);
471 }
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     if (initMode==TRANSMIT){
488         nrf24_tx_address(CURRENT_BOARD_ADDRESS);
489         nrf24_rx_address(BOARD_ADDRESS[boardIndex]);
490         }else{
491         nrf24_tx_address(BOARD_ADDRESS[boardIndex]);
492         nrf24_rx_address(CURRENT_BOARD_ADDRESS);
493     }
494     nrf24_powerUpRx();
495 }
```

```
 1  #ifndef NRF24
 2  #define NRF24
 3
 4  #ifndef F_CPU
 5  #define F_CPU 16000000UL
 6  #endif
 7
 8  #include "nRF24L01_Definitions.h"
 9  #include "Command_Handler.h"
10  #include <stdint.h>
11  #include <stdbool.h>
12  #include <avr/io.h>
13  #include <avr/delay.h>
14
15
16
17  #ifndef BIT_MANIPULATION_MACRO
18  #define BIT_MANIPULATION_MACRO 1
19  #define bit_get(p,m) ((p) & (m))
20  #define bit_set(p,m) ((p) |= (m))
21  #define bit_clear(p,m) ((p) &= ~(m))
22  #define bit_flip(p,m) ((p) ^= (m))
23  #define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
24  #define BIT(x) (0x01 << (x))
25  #define LONGBIT(x) ((unsigned long)0x00000001 << (x))
26  #endif
27
28  #define LOW 0
29  #define HIGH 1
30  #define nrf24_ADDR_LEN 5
31  #define nrf24_CONFIG ((1<<EN_CRC)|(0<<CRCO))
32  #define NRF24_TRANSMISSON_OK 0
33  #define NRF24_MESSAGE_LOST   1
34
35  #define CLEAR_FAULTY_RF_LED         bit_clear(PORTD, BIT(7))
36  #define FLIP_FAULTY_RF_LED          bit_flip(PORTD, BIT(7))
37
38
39  enum TransmissionMode {
40      RECEIVE,
41      TRANSMIT
42  };
43  typedef enum TransmissionMode TransmissionMode;
44
45  enum CommandsBoard {
46      MAIN_BOARD = 0,
47      POWER_BOARD = 1,
48      MOTORIZED_BOARD = 2
49  };
50  typedef enum CommandsBoard CommandsBoard;
51
52  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
65
66  uint8_t nrf24_dataReady();
67  uint8_t nrf24_isSending();
68  uint8_t nrf24_getStatus();
69  uint8_t nrf24_rxFifoEmpty();
70
71  void    nrf24_send(uint8_t* value);
72  void    nrf24_getData(uint8_t* data);
73
74  uint8_t nrf24_payloadLength();
75
76  uint8_t nrf24_lastMessageStatus();
77  uint8_t nrf24_retransmissionCount();
78
79  uint8_t nrf24_payload_length();
80
81  void    nrf24_powerUpRx();
82  void    nrf24_powerUpTx();
83  void    nrf24_powerDown();
84
85  uint8_t spi_transfer(uint8_t tx);
86  void    nrf24_transmitSync(uint8_t* dataout,uint8_t len);
87  void    nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len);
88  void    nrf24_configRegister(uint8_t reg, uint8_t value);
89  void    nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len);
90  void    nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len);
91
92  extern void nrf24_setupPins();
93
94  extern void nrf24_ce_digitalWrite(uint8_t state);
95
96  extern void nrf24_csn_digitalWrite(uint8_t state);
97
98  extern void nrf24_sck_digitalWrite(uint8_t state);
99
100 extern void nrf24_mosi_digitalWrite(uint8_t state);
101
102 extern uint8_t nrf24_miso_digitalRead();
103
104 #endif
```

```
 1
 2  /* Memory Map */
 3  #define CONFIG      0x00
 4  #define EN_AA       0x01
 5  #define EN_RXADDR   0x02
 6  #define SETUP_AW    0x03
 7  #define SETUP_RETR  0x04
 8  #define RF_CH       0x05
 9  #define RF_SETUP    0x06
10  #define STATUS      0x07
11  #define OBSERVE_TX  0x08
12  #define CD          0x09
13  #define RX_ADDR_P0  0x0A
14  #define RX_ADDR_P1  0x0B
15  #define RX_ADDR_P2  0x0C
16  #define RX_ADDR_P3  0x0D
17  #define RX_ADDR_P4  0x0E
18  #define RX_ADDR_P5  0x0F
19  #define TX_ADDR     0x10
20  #define RX_PW_P0    0x11
21  #define RX_PW_P1    0x12
22  #define RX_PW_P2    0x13
23  #define RX_PW_P3    0x14
24  #define RX_PW_P4    0x15
25  #define RX_PW_P5    0x16
26  #define FIFO_STATUS 0x17
27  #define DYNPD       0x1C
28
29  /* Bit Mnemonics */
30
31  /* configuration register */
32  #define MASK_RX_DR  6
33  #define MASK_TX_DS  5
34  #define MASK_MAX_RT 4
35  #define EN_CRC      3
36  #define CRCO        2
37  #define PWR_UP      1
38  #define PRIM_RX     0
39
40  /* enable auto acknowledgment */
41  #define ENAA_P5     5
42  #define ENAA_P4     4
43  #define ENAA_P3     3
44  #define ENAA_P2     2
45  #define ENAA_P1     1
46  #define ENAA_P0     0
47
48  /* enable rx addresses */
49  #define ERX_P5      5
50  #define ERX_P4      4
51  #define ERX_P3      3
52  #define ERX_P2      2
```

```
53  #define ERX_P1      1
54  #define ERX_P0      0
55
56  /* setup of address width */
57  #define AW          0 /* 2 bits */
58
59  /* setup of auto re-transmission */
60  #define ARD         4 /* 4 bits */
61  #define ARC         0 /* 4 bits */
62
63  /* RF setup register */
64  #define PLL_LOCK    4
65  #define RF_DR       3
66  #define RF_PWR      1 /* 2 bits */
67
68  /* general status register */
69  #define RX_DR       6
70  #define TX_DS       5
71  #define MAX_RT      4
72  #define RX_P_NO     1 /* 3 bits */
73  #define TX_FULL     0
74
75  /* transmit observe register */
76  #define PLOS_CNT    4 /* 4 bits */
77  #define ARC_CNT     0 /* 4 bits */
78
79  /* fifo status */
80  #define TX_REUSE    6
81  #define FIFO_FULL   5
82  #define TX_EMPTY    4
83  #define RX_FULL     1
84  #define RX_EMPTY    0
85
86  /* dynamic length */
87  #define DPL_P0      0
88  #define DPL_P1      1
89  #define DPL_P2      2
90  #define DPL_P3      3
91  #define DPL_P4      4
92  #define DPL_P5      5
93
94  /* Instruction Mnemonics */
95  #define R_REGISTER    0x00 /* last 4 bits will indicate reg. address */
96  #define W_REGISTER    0x20 /* last 4 bits will indicate reg. address */
97  #define REGISTER_MASK 0x1F
98  #define R_RX_PAYLOAD  0x61
99  #define W_TX_PAYLOAD  0xA0
100 #define FLUSH_TX      0xE1
101 #define FLUSH_RX      0xE2
102 #define REUSE_TX_PL   0xE3
103 #define ACTIVATE      0x50
104 #define R_RX_PL_WID   0x60
```

```
105  #define NOP           0xFF
106
```

```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
```