SAPLM

Código de fuente: Módulo principal (Lenguaje: AVR-GCC)

Contenidos:

main.c
Command_Handler.c
Command_Handler.h
UART_Bluetooth.c
UART_Bluetooth.h
crc.c
crc.h
nrf24.c
nrf24.h

```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
```

```c

#include "Command_Handler.h"
#include "UART_Bluetooth.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;
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
```

SAPLM

Código de fuente: Módulo de potencia (Lenguaje: AVR-GCC)

Contenidos:

```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
```

SAPLM

Código de fuente: Módulo motriz (Lenguaje: AVR-GCC)

Contenidos:

```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)
{
    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(); }
        }

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


void initIO(){
    /*
        Input/Output pin initialization
        1 : OUTPUT | 0 : INPUT | 0b76543210 Bit order
        ATTACHMENTS
            NURSE SIGN  : PB0               |   OUTPUT
            GREEN LED   : PB1               |   OUTPUT      (SWAPPED IN PCB)
            RED LED     : PB2               |   OUTPUT
        STEP MOTOR A (CURTAIN)
            TERMINAL NO.1 : PD0             |   OUTPUT
            TERMINAL NO.2 : PD1             |   OUTPUT
            TERMINAL NO.3 : PD2             |   OUTPUT
            TERMINAL NO.4 : PD3             |   OUTPUT
        STEP MOTOR B (STRETCHER)
            TERMINAL NO.1 : PD4             |   OUTPUT
```

```
51              TERMINAL NO.2 : PD5             |   OUTPUT
52              TERMINAL NO.3 : PD6             |   OUTPUT
53              TERMINAL NO.4 : PD7             |   OUTPUT
54          nRF24L01
55              CE   : PC0                      |   OUTPUT
56              CSN  : PC1                      |   OUTPUT
57              MISO : PD0 (MSPIM MISO ATMEGA)  |   INPUT
58              MOSI : PD1 (MSPIM MOSI ATMEGA)  |   OUTPUT
59              SCK  : PD4 (MSPIM XCK)          |   OUTPUT
60      */
61      DDRD = 0b11111111;
62      DDRB = 0b00101111;
63      DDRC = 0b11011111;
64  }
65
66
67
68
69
70
```

```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  void HandleAvailableCommand(){
71      lastMessageCommandType.handlerFunction();
72  }
73
74  CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t
      parameterCount, uint8_t targetBoardID){
75      memset(command_buffer, 0, 32);
76      command_buffer[0] = SOH;
77      if (lastMessagePID==0xFF) { lastMessagePID++; } else { lastMessagePID = 0; }
78      command_buffer[1] = lastMessagePID;
79      command_buffer[2] = targetBoardID;
80      command_buffer[3] = currentModuleID;
81      command_buffer[4] = STX;
82      command_buffer[5] = targetTypeID;
83
84      if (parameterCount>12) return PARAMETER_COUNT_OVERSIZE;
85
86      uint8_t* parameterStart = &command_buffer[6];
87
88      for (uint8_t x = 0; x < parameterCount; x++){
89          *parameterStart = parameter[x].byteLength;
90          memcpy(parameterStart+1, parameter[x].startingPointer, parameter
              [x].byteLength);
91          parameterStart+=(parameter[x].byteLength)+1;
92      }
93
94      crc_t crc;
95      crc = crc_init();
96      uint8_t crc_length = ((parameterStart)-(&command_buffer[0]));
97      crc = crc_update(crc, &command_buffer[0], crc_length);
```

```
 98      crc = crc_finalize(crc);
 99
100      *parameterStart = ETX;
101      *(parameterStart+1) = crc;
102      *(parameterStart+2) = ETB;
103
104      return SUCCESFUL_COMPOSITION;
105  }
106
107  void writeParameterValue(uint8_t parameterIndex, void* parameterData, uint8_t
       parameterByteLength){
108      parameter[parameterIndex].startingPointer = (uint8_t*) realloc(parameter
           [parameterIndex].startingPointer, parameterByteLength);
109      memcpy(parameter[parameterIndex].startingPointer, parameterData,
           parameterByteLength);
110      parameter[parameterIndex].byteLength = parameterByteLength;
111  }
112
113  void UPDATE_ALL_DEVICES_VALUE_H() {
114      for (uint8_t x = 0; x < AVAILABLE_DEVICES;x++)
115      {
116          deviceStoredValue[x] = *((uint8_t*)parameter[x].startingPointer);
117
118          switch (x) {
119              case 0:
120                  STRETCHER_POS_CHANGE_HANDLE(deviceStoredValue[x]);
121              break;
122              case 1:
123                  CURTAIN_POS_CHANGE_HANDLE(deviceStoredValue[x]);
124              break;
125              case 2:
126                  if (deviceStoredValue[x]==0xFF){
127                      for (uint8_t x = 0; x < 6; x++)
128                      {
129                          bit_flip(PORTB, BIT(0));
130                          bit_flip(PORTB, BIT(1));
131                          bit_flip(PORTB, BIT(2));
132                          _delay_ms(200);
133                      }
134                      bit_clear(PORTB, BIT(0));
135                      bit_clear(PORTB, BIT(1));
136                      bit_clear(PORTB, BIT(2));
137                  }
138              break;
139          }
140      }
141
142  }
143
144  #define MOTOR_DELAY_MS 1
145  #define CURTAIN_CALIBRATION_CONSTANT 200
146  #define STRETCHER_CALIBRATION_CONSTANT 50
```

```
147
148 void UPDATE_DEVICE_VALUE_H() {
149     const uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
150     const uint8_t deviceValue = *((uint8_t*)parameter[1].startingPointer);
151
152     switch (deviceIndex) {
153         case 0:
154             STRETCHER_POS_CHANGE_HANDLE(deviceValue);
155         break;
156         case 1:
157             CURTAIN_POS_CHANGE_HANDLE(deviceValue);
158         break;
159         case 2:
160             for (uint8_t x = 0; x < 6; x++)
161             {
162                 bit_flip(PORTB, BIT(0));
163                 bit_flip(PORTB, BIT(1));
164                 bit_flip(PORTB, BIT(2));
165                 _delay_ms(200);
166             }
167             bit_clear(PORTB, BIT(0));
168             bit_clear(PORTB, BIT(1));
169             bit_clear(PORTB, BIT(2));
170         break;
171     }
172
173     deviceStoredValue[deviceIndex] = deviceValue;
174
175 }
176
177 void GET_ALL_DEVICES_VALUE_H() {}
178
179 void GET_DEVICE_VALUE_H() {
180     _delay_ms(100);
181     uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
182     writeParameterValue(0, &deviceIndex, 1);
183     writeParameterValue(1, &deviceStoredValue[deviceIndex], 2);
184     ComposeMessageToBuffer(UPDATE_DEVICE_VALUE_ID, 2, 0x7C);
185
186     nrf24_initRF_SAFE(MAIN_BOARD, TRANSMIT);
187     nrf24_send(command_buffer);
188     while(nrf24_isSending());
189     uint8_t messageStatus = nrf24_lastMessageStatus();
190 }
191 void MESSAGE_STATUS_H() {}
192
193
194 uint8_t previousCurtainPosition = 0;
195 uint8_t previousStretcherPosition = 0;
196
197
198 void CURTAIN_POS_CHANGE_HANDLE(uint8_t positionToMove){
```

```
199     bit_set(PORTB, BIT(1));
200     bit_set(PORTB, BIT(2));
201
202
203     if (positionToMove<8) {
204         uint16_t degreesToMove = abs(positionToMove-previousCurtainPosition)
              *CURTAIN_CALIBRATION_CONSTANT;
205
206         if((positionToMove-previousCurtainPosition)>0){
207             for (uint16_t x = 0; x < degreesToMove;x++){
208                 PORTD = 0b00000011;
209                 _delay_ms(MOTOR_DELAY_MS);
210                 PORTD = 0b00000110;
211                 _delay_ms(MOTOR_DELAY_MS);
212                 PORTD = 0b00001100;
213                 _delay_ms(MOTOR_DELAY_MS);
214                 PORTD = 0b00001001;
215                 _delay_ms(MOTOR_DELAY_MS);
216             }
217             }else{
218             for (uint16_t x = 0; x < degreesToMove;x++){
219                 PORTD = 0b00001100;
220                 _delay_ms(MOTOR_DELAY_MS);
221                 PORTD = 0b00000110;
222                 _delay_ms(MOTOR_DELAY_MS);
223                 PORTD = 0b00000011;
224                 _delay_ms(MOTOR_DELAY_MS);
225                 PORTD = 0b00001001;
226                 _delay_ms(MOTOR_DELAY_MS);
227             }
228         }
229
230         PORTD = 0b00000000;
231         previousCurtainPosition = positionToMove;
232     }
233     bit_clear(PORTB, BIT(1));
234     bit_clear(PORTB, BIT(2));
235 }
236
237 void STRETCHER_POS_CHANGE_HANDLE(uint8_t positionToMove){
238     bit_set(PORTB, BIT(1));
239     bit_set(PORTB, BIT(2));
240
241     if (positionToMove<4) {
242         uint16_t degreesToMove = abs(positionToMove-previousStretcherPosition)
              *STRETCHER_CALIBRATION_CONSTANT;
243
244         if((positionToMove-previousCurtainPosition)>0){
245             for (uint16_t x = 0; x < degreesToMove;x++){
246                 PORTD = 0b00110000;
247                 _delay_ms(MOTOR_DELAY_MS);
248                 PORTD = 0b01100000;
```

```
249                 _delay_ms(MOTOR_DELAY_MS);
250                 PORTD = 0b11000000;
251                 _delay_ms(MOTOR_DELAY_MS);
252                 PORTD = 0b10010000;
253                 _delay_ms(MOTOR_DELAY_MS);
254             }
255             }else{
256             for (uint16_t x = 0; x < degreesToMove;x++){
257                 PORTD = 0b11000000;
258                 _delay_ms(MOTOR_DELAY_MS);
259                 PORTD = 0b01100000;
260                 _delay_ms(MOTOR_DELAY_MS);
261                 PORTD = 0b00110000;
262                 _delay_ms(MOTOR_DELAY_MS);
263                 PORTD = 0b10010000;
264                 _delay_ms(MOTOR_DELAY_MS);
265             }
266         }
267 
268         PORTD = 0b00000000;
269         previousStretcherPosition = positionToMove;
270     }
271     bit_clear(PORTB, BIT(1));
272     bit_clear(PORTB, BIT(2));
273 }
```

```
 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  #include "nrf24.h"
22
23  #ifndef BIT_MANIPULATION_MACRO
24  #define BIT_MANIPULATION_MACRO 1
25  #define bit_get(p,m) ((p) & (m))
26  #define bit_set(p,m) ((p) |= (m))
27  #define bit_clear(p,m) ((p) &= ~(m))
28  #define bit_flip(p,m) ((p) ^= (m))
29  #define bit_write(c,p,m) (c ? bit_set(p,m) : bit_clear(p,m))
30  #define BIT(x) (0x01 << (x))
31  #define LONGBIT(x) ((unsigned long)0x00000001 << (x))
32  #endif
33
34  #define currentModuleID 0x03
35  #define SOH 0x01
36  #define STX 0x02
37  #define ETX 0x03
38  #define ETB 0x17
39  #define ON_STATE    0xFF
40  #define OFF_STATE   0x00
41
42  typedef struct CommandType {
43      void (*handlerFunction)();
44  } CommandType;
45
46  typedef enum   {
47      SUCCESFUL_DECOMPOSITION,
48      WRONG_HEADER_SEGMENTATION,
49      WRONG_FOOTER_SEGMENTATION,
50      WRONG_CHECKSUM_CONSISTENCY,
51      WRONG_MODULE_ID,
52      UNDEFINED_COMMAND_CODE,
```

```
53      PARAMETER_DATA_OVERFLOW,
54      PARAMETER_COUNT_OVERSIZE,
55      RETRANSMISSION_FAILED,
56      SUCCESFUL_RETRANSMISSION,
57      SUCCESFUL_COMPOSITION
58  } CommandStatus;
59
60
61  typedef enum {
62      RF_SUCCESFUL_TRANSMISSION,
63      RF_UNREACHEABLE_MODULE,
64      RF_ACKNOWLEDGE_FAILED
65  } RF_TransmissionStatus;
66
67  typedef enum   {
68                  UPDATE_ALL_DEVICES_VALUE_ID,
69                  UPDATE_DEVICE_VALUE_ID,
70                  GET_ALL_DEVICES_VALUE_ID,
71                  GET_DEVICE_VALUE_ID,
72                  MESSAGE_STATUS_ID
73  } CommandTypeID;
74
75  typedef struct {
76      void *startingPointer;
77      uint8_t byteLength;
78  } Parameter;
79
80  Parameter parameter[12];
81  uint8_t *command_buffer;
82  bool memoryInitialized;
83  uint8_t lastMessagePID;
84  CommandType lastMessageCommandType;
85  uint8_t lastTargetModuleID;
86  uint8_t lastTransmitterModuleID;
87
88
89  #define AVAILABLE_DEVICES 3
90  uint8_t deviceStoredValue[AVAILABLE_DEVICES];       //Uint8, las posiciones no se
       guardan en grados
91
92
93
94  void STRETCHER_POS_CHANGE_HANDLE(uint8_t positionToMove);
95  void CURTAIN_POS_CHANGE_HANDLE(uint8_t positionToMove);
96
97  extern void UPDATE_ALL_DEVICES_VALUE_H(), UPDATE_DEVICE_VALUE_H(),
      GET_ALL_DEVICES_VALUE_H(), GET_DEVICE_VALUE_H(), MESSAGE_STATUS_H();
98  extern CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t
      parameterCount, uint8_t targetBoardID);
99  extern CommandStatus DecomposeMessageFromBuffer();
100 extern void writeParameterValue(uint8_t parameterIndex, void* parameterData,
      uint8_t parameterByteLength);
```

```
101  extern void HandleAvailableCommand();
102  extern bool initliazeMemory();
103
104  #endif /* COMMAND_HANDLER_H_ */
```

```
 1
 2  #define UCPHA0 1
 3  #define BAUD_RATE 9600UL
 4  #define UBRR_VALUE ((F_CPU)/(2UL*BAUD_RATE))-1
 5
 6  #include "nrf24.h"
 7  #include "Command_Handler.h"
 8
 9  uint8_t payload_len;
10  uint8_t selectedChannel;
11
12  uint8_t MOTORIZED_BOARD_ADDR[5] =   {0xF0,0xF0,0xF0,0xF0,0xC9};
13  uint8_t MAIN_BOARD_ADDR[5] =            {0xA4,0xA4,0xA4,0xA4,0xA4};
14  uint8_t POWER_BOARD_ADDR[5] =       {0xF0,0xF0,0xF0,0xF0,0xF0};
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 = &MOTORIZED_BOARD_ADDR[0];
18
19  uint8_t GENERAL_RF_CHANNEL = 112;
20
21
22
23  void nrf24_init()
24  {
25      nrf24_setupPins();
26      nrf24_ce_digitalWrite(LOW);
27      nrf24_csn_digitalWrite(HIGH);
28  }
29
30  void nrf24_config(uint8_t channel, uint8_t pay_length)
31  {
32      /* Use static payload length ... */
33      payload_len = pay_length;
34      selectedChannel = channel;
35      // Set RF channel
36      nrf24_configRegister(RF_CH,channel);
37      // Set length of incoming payload
38      nrf24_configRegister(RX_PW_P0, 0x00); // Auto-ACK pipe ...
39      nrf24_configRegister(RX_PW_P1, payload_len); // Data payload pipe
40      nrf24_configRegister(RX_PW_P2, 0x00); // Pipe not used
41      nrf24_configRegister(RX_PW_P3, 0x00); // Pipe not used
42      nrf24_configRegister(RX_PW_P4, 0x00); // Pipe not used
43      nrf24_configRegister(RX_PW_P5, 0x00); // Pipe not used
44      // 1 Mbps, TX gain: 0dbm
45      nrf24_configRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR));
46      // CRC enable, 1 byte CRC length
47      nrf24_configRegister(CONFIG,nrf24_CONFIG);
48      // Auto Acknowledgment
49      nrf24_configRegister(EN_AA,(1<<ENAA_P0)|(1<<ENAA_P1)|(0<<ENAA_P2)|
          (0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5));
50      // Enable RX addresses
```

```
51      nrf24_configRegister(EN_RXADDR,(1<<ERX_P0)|(1<<ERX_P1)|(0<<ERX_P2)|
          (0<<ERX_P3)|(0<<ERX_P4)|(0<<ERX_P5));
52      // Auto retransmit delay: 1000 us and Up to 15 retransmit trials
53      nrf24_configRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC));
54      // Dynamic length configurations: No dynamic length
55      nrf24_configRegister(DYNPD,(0<<DPL_P0)|(0<<DPL_P1)|(0<<DPL_P2)|(0<<DPL_P3)|
          (0<<DPL_P4)|(0<<DPL_P5));
56
57  }
58
59  bool nrf24_checkConfig(){
60      // Check all registers
61      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==false) return false;
62      if (nrf24_checkRegister(RX_PW_P0, 0x00,1)==false) return false;
63      if (nrf24_checkRegister(RX_PW_P1, payload_len,1)==false) return false;
64      if (nrf24_checkRegister(RX_PW_P2, 0x00,1)==false) return false;
65      if (nrf24_checkRegister(RX_PW_P3, 0x00,1)==false) return false;
66      if (nrf24_checkRegister(RX_PW_P4, 0x00,1)==false) return false;
67      if (nrf24_checkRegister(RX_PW_P5, 0x00,1)==false) return false;
68      if (nrf24_checkRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR),1)==false)
          return false;
69      if (nrf24_checkRegister(CONFIG,nrf24_CONFIG,1)==false) return false;
70      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;
71      if (nrf24_checkRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC),1)==false) return
          false;
72      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;
73
74      return true;
75  }
76
77  bool nrf24_checkAvailability(){
78      if (nrf24_checkRegister(RF_CH, selectedChannel,1)==true) { return true; }
          else { return false;}
79  }
80
81
82
83
84  void faultyRF_Alarm(){
85      CLEAR_FAULTY_RF_LED;
86      for (uint8_t x = 0; x < 6; x++)
87      {
88          FLIP_FAULTY_RF_LED;
89          _delay_ms(125);
90      }
91      _delay_ms(250);
92  }
93
94
95
```

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

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

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

```
252     rv = nrf24_getStatus();
253
254     /* Transmission went OK */
255     if((rv & ((1 << TX_DS))))
256     {
257         return NRF24_TRANSMISSON_OK;
258     }
259     /* Maximum retransmission count is reached */
260     /* Last message probably went missing ... */
261     else if((rv & ((1 << MAX_RT))))
262     {
263         return NRF24_MESSAGE_LOST;
264     }
265     /* Probably still sending ... */
266     else
267     {
268         return 0xFF;
269     }
270 }
271
272 void nrf24_powerUpRx()
273 {
274     nrf24_csn_digitalWrite(LOW);
275     spi_transfer(FLUSH_RX);
276     nrf24_csn_digitalWrite(HIGH);
277
278     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
279
280     nrf24_ce_digitalWrite(LOW);
281     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(1<<PRIM_RX)));
282     nrf24_ce_digitalWrite(HIGH);
283 }
284
285 void nrf24_powerUpTx()
286 {
287     nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));
288
289     nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(0<<PRIM_RX)));
290 }
291
292 void nrf24_powerDown()
293 {
294     nrf24_ce_digitalWrite(LOW);
295     nrf24_configRegister(CONFIG,nrf24_CONFIG);
296 }
297
298 uint8_t spi_transfer(uint8_t tx)
299 {
300     uint8_t i = 0;
301     uint8_t rx = 0;
302
303     nrf24_sck_digitalWrite(LOW);
```

```
304
305     for(i=0;i<8;i++)
306     {
307
308         if(tx & (1<<(7-i)))
309         {
310             nrf24_mosi_digitalWrite(HIGH);
311         }
312         else
313         {
314             nrf24_mosi_digitalWrite(LOW);
315         }
316
317         nrf24_sck_digitalWrite(HIGH);
318
319         rx = rx << 1;
320         if(nrf24_miso_digitalRead())
321         {
322             rx |= 0x01;
323         }
324
325         nrf24_sck_digitalWrite(LOW);
326
327     }
328
329     return rx;
330 }
331
332 /* send and receive multiple bytes over SPI */
333 void nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len)
334 {
335     uint8_t i;
336
337     for(i=0;i<len;i++)
338     {
339         datain[i] = spi_transfer(dataout[i]);
340     }
341
342 }
343
344 /* send multiple bytes over SPI */
345 void nrf24_transmitSync(uint8_t* dataout,uint8_t len)
346 {
347     uint8_t i;
348
349     for(i=0;i<len;i++)
350     {
351         spi_transfer(dataout[i]);
352     }
353
354 }
355
```

```
356 /* Clocks only one byte into the given nrf24 register */
357 void nrf24_configRegister(uint8_t reg, uint8_t value)
358 {
359     nrf24_csn_digitalWrite(LOW);
360     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
361     spi_transfer(value);
362     nrf24_csn_digitalWrite(HIGH);
363 }
364 
365 /* Read single register from nrf24 */
366 void nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len)
367 {
368     nrf24_csn_digitalWrite(LOW);
369     spi_transfer(R_REGISTER | (REGISTER_MASK & reg));
370     nrf24_transferSync(value,value,len);
371     nrf24_csn_digitalWrite(HIGH);
372 }
373 
374 /* Write to a single register of nrf24 */
375 void nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len)
376 {
377     nrf24_csn_digitalWrite(LOW);
378     spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
379     nrf24_transmitSync(value,len);
380     nrf24_csn_digitalWrite(HIGH);
381 }
382 
383 /* Check single register from nrf24 */
384 bool nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len)
385 {
386     uint8_t registerValue;
387     nrf24_readRegister(reg,&registerValue,len);
388     if (registerValue==desiredValue) { return true; } else { return false; }
389 }
390 
391 #define RF_DDR  DDRC
392 #define RF_PORT PORTC
393 #define RF_PIN  PINC
394 
395 #define set_bit(reg,bit) reg |= (1<<bit)
396 #define clr_bit(reg,bit) reg &= ~(1<<bit)
397 #define check_bit(reg,bit) (reg&(1<<bit))
398 
399 /* ---------------------------------------------------------------------------- */
400 
401 void nrf24_setupPins()
402 {
403     set_bit(RF_DDR,0); // CE output
404     set_bit(RF_DDR,1); // CSN output
405     set_bit(RF_DDR,2); // SCK output
406     set_bit(RF_DDR,3); // MOSI output
407     clr_bit(RF_DDR,4); // MISO input
```

```
408 }
409 /* ------------------------------------------------------------------------ */
410 void nrf24_ce_digitalWrite(uint8_t state)
411 {
412     if(state)
413     {
414         set_bit(RF_PORT,0);
415     }
416     else
417     {
418         clr_bit(RF_PORT,0);
419     }
420 }
421 /* ------------------------------------------------------------------------ */
422 void nrf24_csn_digitalWrite(uint8_t state)
423 {
424     if(state)
425     {
426         set_bit(RF_PORT,1);
427     }
428     else
429     {
430         clr_bit(RF_PORT,1);
431     }
432 }
433 /* ------------------------------------------------------------------------ */
434 void nrf24_sck_digitalWrite(uint8_t state)
435 {
436     if(state)
437     {
438         set_bit(RF_PORT,2);
439     }
440     else
441     {
442         clr_bit(RF_PORT,2);
443     }
444 }
445 /* ------------------------------------------------------------------------ */
446 void nrf24_mosi_digitalWrite(uint8_t state)
447 {
448     if(state)
449     {
450         set_bit(RF_PORT,3);
451     }
452     else
453     {
454         clr_bit(RF_PORT,3);
455     }
456 }
457 /* ------------------------------------------------------------------------ */
458 uint8_t nrf24_miso_digitalRead()
459 {
```

```
460     return check_bit(RF_PIN,4);
461 }
462 /* ------------------------------------------------------------------------ */
463
464 void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode){
465
466     initliazeMemory();
467     bool successfulRfInit = false;
468
469     while(successfulRfInit==false){
470         nrf24_powerDown();
471         nrf24_init();
472         nrf24_config(GENERAL_RF_CHANNEL,32);
473         if (nrf24_checkConfig()) { successfulRfInit = true; } else
            { faultyRF_Alarm(); }
474     }
475
476     if (initMode==TRANSMIT){
477         nrf24_tx_address(CURRENT_BOARD_ADDRESS);
478         nrf24_rx_address(BOARD_ADDRESS[boardIndex]);
479         }else{
480         nrf24_tx_address(BOARD_ADDRESS[boardIndex]);
481         nrf24_rx_address(CURRENT_BOARD_ADDRESS);
482     }
483     nrf24_powerUpRx();
484 }
```

```
#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 AVAILABLE_COMMAND_BOARDS    2
#define CLEAR_FAULTY_RF_LED         bit_clear(PORTB, BIT(1))
#define FLIP_FAULTY_RF_LED          bit_flip(PORTB, BIT(1))


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

enum CommandsBoard {
    MAIN_BOARD = 0,
    POWER_BOARD = 1,
    MOTORIZED_BOARD = 2
};
typedef enum CommandsBoard CommandsBoard;

```

```
53  extern void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode);
54
55  void    nrf24_init();
56  void    nrf24_rx_address(uint8_t* adr);
57  void    nrf24_tx_address(uint8_t* adr);
58  void    nrf24_config(uint8_t channel, uint8_t pay_length);
59  bool    nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len);
60  bool    nrf24_checkConfig();
61  bool    nrf24_checkAvailability();
62
63  void faultyRF_Alarm();
64
65
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
```

```
 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
```