



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

Contenidos:

main.c

Command_Handler.c

Command_Handler.h

nrf24.c

nrf24.h

nRF24L01_Definitions.h

crc.c

crc.h

```

1  #ifndef F_CPU
2  #define F_CPU 16000000UL
3  #endif
4  #include <avr/io.h>
5  #include <util/delay.h>
6  #include <avr/interrupt.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <stdbool.h>
10 #include <stdint.h>
11
12 #include "nrf24.h"
13
14 void initIO();
15
16 int main(void)
17 {
18     initIO();
19     nrf24_initRF_SAFE(MAIN_BOARD, RECEIVE); // CONNECTION TO MAIN BOARD : GENERAL RF CHANNEL 112
20
21     while (1)
22     {
23         if(nrf24_dataReady())
24         {
25
26             nrf24_getData(command_buffer);
27             CommandStatus status = DecomposeMessageFromBuffer();
28             if (status==SUCCESSFUL_DECOMPOSITION) { HandleAvailableCommand(); }
29         }
30
31         if (nrf24_checkAvailability()==false) { nrf24_initRF_SAFE(MAIN_BOARD, RECEIVE); }
32     }
33 }
34
35
36 void initIO(){
37     /*
38     Input/Output pin initialization
39     1 : OUTPUT | 0 : INPUT | 0b76543210 Bit order
40     ATTACHMENTS
41     NURSE SIGN : PB0 | OUTPUT
42     GREEN LED : PB1 | OUTPUT (SWAPPED IN PCB)
43     RED LED : PB2 | OUTPUT
44     STEP MOTOR A (CURTAIN)
45     TERMINAL NO.1 : PD0 | OUTPUT
46     TERMINAL NO.2 : PD1 | OUTPUT
47     TERMINAL NO.3 : PD2 | OUTPUT
48     TERMINAL NO.4 : PD3 | OUTPUT
49     STEP MOTOR B (STRETCHER)
50     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
```

```

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

```
198     crc = crc_finalize(crc);
199
200     *parameterStart = ETX;
201     *(parameterStart+1) = crc;
202     *(parameterStart+2) = ETB;
203
204     return SUCCESSFUL_COMPOSITION;
205 }
206
207 void writeParameterValue(uint8_t parameterIndex, void* parameterData, uint8_t
parameterByteLength) {
208     parameter[parameterIndex].startingPointer = (uint8_t*) realloc(parameter
[parameterIndex].startingPointer, parameterByteLength);
209     memcpy(parameter[parameterIndex].startingPointer, parameterData,
parameterByteLength);
210     parameter[parameterIndex].byteLength = parameterByteLength;
211 }
212
213 void UPDATE_ALL_DEVICES_VALUE_H() {
214     for (uint8_t x = 0; x < AVAILABLE_DEVICES; x++)
215     {
216         deviceStoredValue[x] = *((uint8_t*)parameter[x].startingPointer);
217
218         switch (x) {
219             case 0:
220                 STRETCHER_POS_CHANGE_HANDLE(deviceStoredValue[x]);
221                 break;
222             case 1:
223                 CURTAIN_POS_CHANGE_HANDLE(deviceStoredValue[x]);
224                 break;
225             case 2:
226                 if (deviceStoredValue[x] == 0xFF) {
227                     for (uint8_t x = 0; x < 6; x++)
228                     {
229                         bit_flip(PORTB, BIT(0));
230                         bit_flip(PORTB, BIT(1));
231                         bit_flip(PORTB, BIT(2));
232                         _delay_ms(200);
233                     }
234                     bit_clear(PORTB, BIT(0));
235                     bit_clear(PORTB, BIT(1));
236                     bit_clear(PORTB, BIT(2));
237                 }
238                 break;
239         }
240     }
241 }
242
243 #define MOTOR_DELAY_MS 1
244 #define CURTAIN_CALIBRATION_CONSTANT 200
245 #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     SUCCESSFUL_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     SUCCESSFUL_RETRANSMISSION,  
57     SUCCESSFUL_COMPOSITION  
58 } CommandStatus;  
59  
60  
61 typedef enum {  
62     RF_SUCCESSFUL_TRANSMISSION,  
63     RF_UNREACHABLE_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],
17                               &MOTORIZED_BOARD_ADDR[0]};
18
19 uint8_t* CURRENT_BOARD_ADDRESS = &MOTORIZED_BOARD_ADDR[0];
20
21 uint8_t GENERAL_RF_CHANNEL = 112;
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)|((0<<03)<<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)|
50                          (0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5));
51     // 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 occurred 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_TRANSMISSION_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 void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode){
464     initliazeMemory();
465     bool successfulRfInit = false;
466     while(successfulRfInit==false){
467         nrf24_powerDown();
468         nrf24_init();
469         nrf24_config(GENERAL_RF_CHANNEL,32);
470         if (nrf24_checkConfig()) { successfulRfInit = true; } else
471             { faultyRF_Alarm(); }
472     }
473     if (initMode==TRANSMIT){
474         nrf24_tx_address(CURRENT_BOARD_ADDRESS);
475         nrf24_rx_address(BOARD_ADDRESS[boardIndex]);
476     }else{
477         nrf24_tx_address(BOARD_ADDRESS[boardIndex]);
478         nrf24_rx_address(CURRENT_BOARD_ADDRESS);
479     }
480     nrf24_powerUpRx();
481 }
482 }
```

```
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_TRANSMISSION_OK 0
33 #define NRF24_MESSAGE_LOST 1
34
35 #define AVAILABLE_COMMAND_BOARDS 2
36 #define CLEAR_FAULTY_RF_LED bit_clear(PORTB, BIT(1))
37 #define FLIP_FAULTY_RF_LED bit_flip(PORTB, BIT(1))
38
39
40 enum TransmissionMode {
41     RECEIVE,
42     TRANSMIT
43 };
44 typedef enum TransmissionMode TransmissionMode;
45
46 enum CommandsBoard {
47     MAIN_BOARD = 0,
48     POWER_BOARD = 1,
49     MOTORIZED_BOARD = 2
50 };
51 typedef enum CommandsBoard CommandsBoard;
52
```

```
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
```

```
1  /**
2   * \file
3   * Functions and types for CRC checks.
4   *
5   * Generated on Wed Sep 11 13:55:53 2019
6   * by pycrc v0.9.2, https://pycrc.org
7   * using the configuration:
8   * - Width      = 8
9   * - Poly       = 0x07
10  * - XorIn      = 0x00
11  * - ReflectIn  = False
12  * - XorOut     = 0x00
13  * - ReflectOut = False
14  * - Algorithm  = bit-by-bit-fast
15  */
16 #include "crc.h" /* include the header file generated with pycrc */
17 #include <stdlib.h>
18 #include <stdint.h>
19 #include <stdbool.h>
20
21
22
23 crc_t crc_update(crc_t crc, const void *data, size_t data_len)
24 {
25     const unsigned char *d = (const unsigned char *)data;
26     unsigned int i;
27     bool bit;
28     unsigned char c;
29
30     while (data_len--) {
31         c = *d++;
32         for (i = 0x80; i > 0; i >>= 1) {
33             bit = crc & 0x80;
34             if (c & i) {
35                 bit = !bit;
36             }
37             crc <<= 1;
38             if (bit) {
39                 crc ^= 0x07;
40             }
41         }
42         crc &= 0xff;
43     }
44     return crc & 0xff;
45 }
46
```

```
1  /**
2   * \file
3   * Functions and types for CRC checks.
4   *
5   * Generated on Wed Sep 11 13:56:48 2019
6   * by pycrc v0.9.2, https://pycrc.org
7   * using the configuration:
8   * - Width      = 8
9   * - Poly       = 0x07
10  * - XorIn      = 0x00
11  * - ReflectIn  = False
12  * - XorOut     = 0x00
13  * - ReflectOut = False
14  * - Algorithm   = bit-by-bit-fast
15  *
16  * This file defines the functions crc_init(), crc_update() and crc_finalize().
17  *
18  * The crc_init() function returns the initial \c crc value and must be called
19  * before the first call to crc_update().
20  * Similarly, the crc_finalize() function must be called after the last call
21  * to crc_update(), before the \c crc is being used.
22  * is being used.
23  *
24  * The crc_update() function can be called any number of times (including zero
25  * times) in between the crc_init() and crc_finalize() calls.
26  *
27  * This pseudo-code shows an example usage of the API:
28  * \code{.c}
29  * crc_t crc;
30  * unsigned char data[MAX_DATA_LEN];
31  * size_t data_len;
32  *
33  * crc = crc_init();
34  * while ((data_len = read_data(data, MAX_DATA_LEN)) > 0) {
35  *     crc = crc_update(crc, data, data_len);
36  * }
37  * crc = crc_finalize(crc);
38  * \endcode
39  */
40 #ifndef CRC_H
41 #define CRC_H
42
43 #include <stdlib.h>
44 #include <stdint.h>
45
46 #ifdef __cplusplus
47 extern "C" {
48 #endif
49
50
51 /**
52  * 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