```c

#define F_CPU 16000000UL

#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))

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

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

bool initRF();
void initIO();
void faultyRF_Alarm();

int main(void)
{
    initIO();
    initRF();

    while (1)
    {
        if(nrf24_dataReady())
        {
            bit_clear(PORTB, BIT(0));

            nrf24_getData(command_buffer);

            bit_set(PORTD, BIT(7));
            _delay_ms(500);
            commandType currentCommand;
            bool success = decomposeCommand(command_buffer, &currentCommand, parameter);
            if (success) { currentCommand.handlerFunction(); }
            bit_clear(PORTD, BIT(7));
        }

        if (nrf24_checkAvailability()==false) { while(initRF()==false); }
    }
}

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
        nRF24L01
            CE   : PC0                      |   OUTPUT
            CSN  : PC1                      |   OUTPUT
            MISO : PD0 (MSPIM MISO ATMEGA)  |   INPUT
            MOSI : PD1 (MSPIM MOSI ATMEGA)  |   OUTPUT
            SCK  : PD4 (MSPIM XCK)          |   OUTPUT
    */
    DDRD = 0b11111110;
    DDRB = 0b00101001;
    DDRC = 0b11011111;

    PORTD = 0b00000000;
    PORTC = 0b00000000;
    PORTB = 0b00000000;
}

bool initRF(){
    uint8_t tx_address[5] = {0xD7,0xD7,0xD7,0xD7,0xD7};
    uint8_t rx_address[5] = {0xE7,0xE7,0xE7,0xE7,0xE7};

    initliazeMemory();

    /* Power down module */
    nrf24_powerDown();

    nrf24_init();

    /* Channel #112 , payload length: 32 */
    nrf24_config(112,32);

    /* Check module configuration */
    if (nrf24_checkConfig()==false) { faultyRF_Alarm(); return false; }

    /* Set the device addresses */
    nrf24_tx_address(tx_address);
    nrf24_rx_address(rx_address);

    /* Power up in receive mode */
    nrf24_powerUpRx();

    return true;
}

void faultyRF_Alarm(){
```

```c
    bit_clear(PORTD, BIT(7));
    for (uint8_t x = 0; x < 6; x++)
    {
        bit_flip(PORTD, BIT(7));
        _delay_ms(125);

    }
}
```

```c


#ifndef COMMAND_HANDLER_H_
#define COMMAND_HANDLER_H_

#include <stdbool.h>
#include <stdint.h>

#ifndef nullptr
#define nullptr ((void *)0)
#endif

#ifndef F_CPU
#define F_CPU               16000000UL
#endif

#define AVAILABLE_COMMANDS 6
#define COMMAND_BUFFER_SIZE 32
#define PARAMETER_BUFFER_SIZE 28

#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

typedef struct commandType {
    const char *commandBase;
    uint8_t nParameters;
    void (*handlerFunction)();
} commandType;

void *parameter[3];
uint8_t *command_buffer;
extern bool initliazeMemory();
bool memoryInitialized;
extern void TURN_RELAY_ON_HANDLE(), TURN_RELAY_OFF_HANDLE(), BUILT_IN_LED_TEST_HANDLER(), TURN_EVERYTHING_ON_HANDLER(), TURN_EVERYTHING_OFF_HANDLER(), CALL_NURSE_HANDLE();

extern void composeCommand(void* output_buffer, commandType* commandT, void** inputParameter);
extern bool decomposeCommand(void* input_buffer, commandType* commandT, void** outputParameter);


#endif /* COMMAND_HANDLER_H_ */
```

```c

#include "Command_Handler.h"
#include "nrf24.h"
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <avr/io.h>
#include <util/delay.h>


const commandType availableCommand[AVAILABLE_COMMANDS] = {
    { .commandBase = "TURN_RELAY_ON", .nParameters = 1, .handlerFunction = &TURN_RELAY_ON_HANDLE},
    { .commandBase = "TURN_RELAY_OFF", .nParameters = 1, .handlerFunction = &TURN_RELAY_OFF_HANDLE},
    { .commandBase = "BUILT_IN_LED_TEST", .nParameters = 0, .handlerFunction = &BUILT_IN_LED_TEST_HANDLER},
    { .commandBase = "TURN_EVERYTHING_ON", .nParameters = 0, .handlerFunction = &TURN_EVERYTHING_ON_HANDLER},
    { .commandBase = "TURN_EVERYTHING_OFF", .nParameters = 0, .handlerFunction = &TURN_EVERYTHING_OFF_HANDLER},
    { .commandBase = "CALL_NURSE", .nParameters = 0, .handlerFunction = &CALL_NURSE_HANDLE}
};

bool initliazeMemory(){
    if(memoryInitialized) return false;
    parameter[0] = (void*)calloc(28,1);
    parameter[1] = (void*)calloc(28,1);
    parameter[2] = (void*)calloc(28,1);
    command_buffer = (uint8_t*)calloc(32,1);
    if(parameter[0]==nullptr||parameter[1]==nullptr||parameter[2]==nullptr||command_buffer==nullptr) return false;
    memoryInitialized = true;
    return true;
}


void composeCommand(void* output_buffer, commandType* commandT, void** inputParameter){
    strcpy(output_buffer, commandT->commandBase);
    char* startParamPTR = (char*)(output_buffer+strlen(commandT->commandBase));
    char* endParamPTR = (char*)(startParamPTR+1+strlen(*inputParameter));

    for (uint8_t index = 0; index < commandT->nParameters; index++){
        *startParamPTR='[';
        strcpy(startParamPTR+1, *inputParameter);
        *endParamPTR=']';
        startParamPTR=(endParamPTR+1);
        if (index!=(commandT->nParameters-1)){
            inputParameter++;
            uint8_t len = strlen(*inputParameter);
            endParamPTR = (char*)(startParamPTR+len+1);
        }
    }
    *startParamPTR='\0';
}

bool decomposeCommand(void* input_buffer, commandType* commandT, void** outputParameter){

    for (uint8_t index = 0; index < AVAILABLE_COMMANDS; index++){
        if (memmem(input_buffer, COMMAND_BUFFER_SIZE, availableCommand[index].commandBase, strlen(availableCommand[index].commandBase))!=nullptr)
        {
            *commandT = availableCommand[index]; break;
        }
        else if (index==(AVAILABLE_COMMANDS-1)) { return false;}
    }

    for (uint8_t x = 0; x < commandT->nParameters; x++){
        uint8_t* startNumPTR = memchr(input_buffer, '[', COMMAND_BUFFER_SIZE);
        uint8_t* endNumPTR = memchr(input_buffer, ']', COMMAND_BUFFER_SIZE);
        if (startNumPTR==nullptr||endNumPTR==nullptr) { if(x==0) return false; break; }
        (*startNumPTR) = 0x20;
        (*endNumPTR) = 0x20;
        startNumPTR++;
        uint32_t bytes = ((endNumPTR)) - ((startNumPTR));
        if (bytes>PARAMETER_BUFFER_SIZE) return false;
        memcpy(outputParameter[x], startNumPTR, bytes);
    }

    return true;
}

void TURN_RELAY_ON_HANDLE() {
    uint8_t relayIndex =  atoi(parameter[0]);
    switch (relayIndex) {
        case 0:
        bit_set(PORTD, BIT(3));
        break;
        case 1:
        bit_set(PORTD, BIT(2));
        break;
        case 2:
        bit_set(PORTD, BIT(6));
        break;
        case 3:
        bit_set(PORTD, BIT(5));
        break;
    }
```

```c
}

void TURN_RELAY_OFF_HANDLE() {
    uint8_t relayIndex =  atoi(parameter[0]);
    switch (relayIndex) {
        case 0:
        bit_clear(PORTD, BIT(3));
        break;
        case 1:
        bit_clear(PORTD, BIT(2));
        break;
        case 2:
        bit_clear(PORTD, BIT(6));
        break;
        case 3:
        bit_clear(PORTD, BIT(5));
        break;
    }
}

void BUILT_IN_LED_TEST_HANDLER(){
    for (uint8_t x = 0; x < 8; x++) {
        bit_flip(PORTD, BIT(7));
        bit_flip(PORTB, BIT(0));
        _delay_ms(250);
    }
    bit_clear(PORTD, BIT(7));
    bit_clear(PORTB, BIT(0));
}

void TURN_EVERYTHING_ON_HANDLER(){
    bit_set(PORTD, BIT(3));
    bit_set(PORTD, BIT(2));
    bit_set(PORTD, BIT(6));
    bit_set(PORTD, BIT(5));
}

void TURN_EVERYTHING_OFF_HANDLER(){
    bit_clear(PORTD, BIT(3));
    bit_clear(PORTD, BIT(2));
    bit_clear(PORTD, BIT(6));
    bit_clear(PORTD, BIT(5));
}

void CALL_NURSE_HANDLE(){
    bit_set(PORTD, BIT(5));
    _delay_ms(500);
    bit_clear(PORTD, BIT(5));
    _delay_ms(500);
    bit_set(PORTD, BIT(5));
    _delay_ms(500);
    bit_clear(PORTD, BIT(5));
    _delay_ms(500);
    bit_set(PORTD, BIT(5));
    _delay_ms(500);
    bit_clear(PORTD, BIT(5));
}

```

```
#ifndef NRF24
#define NRF24

#include "nRF24L01_Definitions.h"
#include <stdint.h>
#include <stdbool.h>
#include <util/delay.h>

#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

void    nrf24_init();
void    nrf24_rx_address(uint8_t* adr);
void    nrf24_tx_address(uint8_t* adr);
void    nrf24_config(uint8_t channel, uint8_t pay_length);
bool    nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len);
bool    nrf24_checkConfig();
bool    nrf24_checkAvailability();


uint8_t nrf24_dataReady();
uint8_t nrf24_isSending();
uint8_t nrf24_getStatus();
uint8_t nrf24_rxFifoEmpty();

void    nrf24_send(uint8_t* value);
void    nrf24_getData(uint8_t* data);

uint8_t nrf24_payloadLength();

uint8_t nrf24_lastMessageStatus();
uint8_t nrf24_retransmissionCount();

uint8_t nrf24_payload_length();

void    nrf24_powerUpRx();
void    nrf24_powerUpTx();
void    nrf24_powerDown();

uint8_t spi_transfer(uint8_t tx);
void    nrf24_transmitSync(uint8_t* dataout,uint8_t len);
void    nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len);
void    nrf24_configRegister(uint8_t reg, uint8_t value);
void    nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len);
void    nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len);

```

```c
extern void nrf24_setupPins();

extern void nrf24_ce_digitalWrite(uint8_t state);

extern void nrf24_csn_digitalWrite(uint8_t state);

extern void nrf24_sck_digitalWrite(uint8_t state);

extern void nrf24_mosi_digitalWrite(uint8_t state);

extern uint8_t nrf24_miso_digitalRead();

#endif

```

```c

#define UCPHA0 1
#define F_CPU 8000000UL
#define BAUD_RATE 9600UL
#define UBRR_VALUE ((F_CPU)/(2UL*BAUD_RATE))-1

#include "nrf24.h"
#include <avr/io.h>

uint8_t payload_len;
uint8_t selectedChannel;

void nrf24_init()
{
    nrf24_setupPins();
    nrf24_ce_digitalWrite(LOW);
    nrf24_csn_digitalWrite(HIGH);
}

void nrf24_config(uint8_t channel, uint8_t pay_length)
{
    /* Use static payload length ... */
    payload_len = pay_length;
    selectedChannel = channel;
    // Set RF channel
    nrf24_configRegister(RF_CH,channel);
    // Set length of incoming payload
    nrf24_configRegister(RX_PW_P0, 0x00); // Auto-ACK pipe ...
    nrf24_configRegister(RX_PW_P1, payload_len); // Data payload pipe
    nrf24_configRegister(RX_PW_P2, 0x00); // Pipe not used
    nrf24_configRegister(RX_PW_P3, 0x00); // Pipe not used
    nrf24_configRegister(RX_PW_P4, 0x00); // Pipe not used
    nrf24_configRegister(RX_PW_P5, 0x00); // Pipe not used
    // 1 Mbps, TX gain: 0dbm
    nrf24_configRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR));
    // CRC enable, 1 byte CRC length
    nrf24_configRegister(CONFIG,nrf24_CONFIG);
    // Auto Acknowledgment
    nrf24_configRegister(EN_AA,(1<<ENAA_P0)|(1<<ENAA_P1)|(0<<ENAA_P2)|(0<<ENAA_P3)|(0<<ENAA_P4)|(0<<ENAA_P5));
    // Enable RX addresses
    nrf24_configRegister(EN_RXADDR,(1<<ERX_P0)|(1<<ERX_P1)|(0<<ERX_P2)|(0<<ERX_P3)|(0<<ERX_P4)|(0<<ERX_P5));
    // Auto retransmit delay: 1000 us and Up to 15 retransmit trials
    nrf24_configRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC));
    // Dynamic length configurations: No dynamic length
    nrf24_configRegister(DYNPD,(0<<DPL_P0)|(0<<DPL_P1)|(0<<DPL_P2)|(0<<DPL_P3)|(0<<DPL_P4)|(0<<DPL_P5));

}

bool nrf24_checkConfig(){
    // Check all registers
    if (nrf24_checkRegister(RF_CH, selectedChannel,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P0, 0x00,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P1, payload_len,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P2, 0x00,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P3, 0x00,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P4, 0x00,1)==false) return false;
    if (nrf24_checkRegister(RX_PW_P5, 0x00,1)==false) return false;
    if (nrf24_checkRegister(RF_SETUP, (0<<RF_DR)|((0x03)<<RF_PWR),1)==false) return false;
    if (nrf24_checkRegister(CONFIG,nrf24_CONFIG,1)==false) return false;
    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;
    if (nrf24_checkRegister(SETUP_RETR,(0x04<<ARD)|(0x0F<<ARC),1)==false) return false;
    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;

    return true;
}

bool nrf24_checkAvailability(){
    if (nrf24_checkRegister(RF_CH, selectedChannel,1)==true) { return true; } else { return false;}
}



/* Set the RX address */
void nrf24_rx_address(uint8_t * adr)
{
    nrf24_ce_digitalWrite(LOW);
    nrf24_writeRegister(RX_ADDR_P1,adr,nrf24_ADDR_LEN);
    nrf24_ce_digitalWrite(HIGH);
}

/* Returns the payload length */
uint8_t nrf24_payload_length()
{
    return payload_len;
}

/* Set the TX address */
void nrf24_tx_address(uint8_t* adr)
{
    /* RX_ADDR_P0 must be set to the sending addr for auto ack to work. */
    nrf24_writeRegister(RX_ADDR_P0,adr,nrf24_ADDR_LEN);
    nrf24_writeRegister(TX_ADDR,adr,nrf24_ADDR_LEN);
}

/* Checks if data is available for reading */
/* Returns 1 if data is ready ... */
```

```c
uint8_t nrf24_dataReady()
{
    // See note in getData() function - just checking RX_DR isn't good enough
    uint8_t status = nrf24_getStatus();

    // We can short circuit on RX_DR, but if it's not set, we still need
    // to check the FIFO for any pending packets
    if ( status & (1 << RX_DR) )
    {
        return 1;
    }

    return !nrf24_rxFifoEmpty();;
}

/* Checks if receive FIFO is empty or not */
uint8_t nrf24_rxFifoEmpty()
{
    uint8_t fifoStatus;

    nrf24_readRegister(FIFO_STATUS,&fifoStatus,1);

    return (fifoStatus & (1 << RX_EMPTY));
}

/* Returns the length of data waiting in the RX fifo */
uint8_t nrf24_payloadLength()
{
    uint8_t status;
    nrf24_csn_digitalWrite(LOW);
    spi_transfer(R_RX_PL_WID);
    status = spi_transfer(0x00);
    nrf24_csn_digitalWrite(HIGH);
    return status;
}

/* Reads payload bytes into data array */
void nrf24_getData(uint8_t* data)
{
    /* Pull down chip select */
    nrf24_csn_digitalWrite(LOW);

    /* Send cmd to read rx payload */
    spi_transfer( R_RX_PAYLOAD );

    /* Read payload */
    nrf24_transferSync(data,data,payload_len);

    /* Pull up chip select */
    nrf24_csn_digitalWrite(HIGH);

    /* Reset status register */
    nrf24_configRegister(STATUS,(1<<RX_DR));
}

/* Returns the number of retransmissions occured for the last message */
uint8_t nrf24_retransmissionCount()
{
    uint8_t rv;
    nrf24_readRegister(OBSERVE_TX,&rv,1);
    rv = rv & 0x0F;
    return rv;
}

// Sends a data package to the default address. Be sure to send the correct
// amount of bytes as configured as payload on the receiver.
void nrf24_send(uint8_t* value)
{
    /* Go to Standby-I first */
    nrf24_ce_digitalWrite(LOW);

    /* Set to transmitter mode , Power up if needed */
    nrf24_powerUpTx();

    /* Do we really need to flush TX fifo each time ? */
    #if 1
    /* Pull down chip select */
    nrf24_csn_digitalWrite(LOW);

    /* Write cmd to flush transmit FIFO */
    spi_transfer(FLUSH_TX);

    /* Pull up chip select */
    nrf24_csn_digitalWrite(HIGH);
    #endif

    /* Pull down chip select */
    nrf24_csn_digitalWrite(LOW);

    /* Write cmd to write payload */
    spi_transfer(W_TX_PAYLOAD);

    /* Write payload */
    nrf24_transmitSync(value,payload_len);

    /* Pull up chip select */
    nrf24_csn_digitalWrite(HIGH);

    /* Start the transmission */
    nrf24_ce_digitalWrite(HIGH);
}

uint8_t nrf24_isSending()
{
    uint8_t status;

    /* read the current status */
    status = nrf24_getStatus();

    /* if sending successful (TX_DS) or max retries exceded (MAX_RT). */
    if((status & ((1 << TX_DS)  | (1 << MAX_RT))))
    {
        return 0; /* false */
    }

    return 1; /* true */

}

uint8_t nrf24_getStatus()
{
    uint8_t rv;
    nrf24_csn_digitalWrite(LOW);
    rv = spi_transfer(NOP);
    nrf24_csn_digitalWrite(HIGH);
    return rv;
}

uint8_t nrf24_lastMessageStatus()
{
    uint8_t rv;

    rv = nrf24_getStatus();

    /* Transmission went OK */
    if((rv & ((1 << TX_DS))))
    {
        return NRF24_TRANSMISSON_OK;
    }
    /* Maximum retransmission count is reached */
    /* Last message probably went missing ... */
    else if((rv & ((1 << MAX_RT))))
    {
        return NRF24_MESSAGE_LOST;
    }
    /* Probably still sending ... */
    else
    {
        return 0xFF;
    }
}

void nrf24_powerUpRx()
{
    nrf24_csn_digitalWrite(LOW);
    spi_transfer(FLUSH_RX);
    nrf24_csn_digitalWrite(HIGH);

    nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));

    nrf24_ce_digitalWrite(LOW);
    nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(1<<PRIM_RX)));
    nrf24_ce_digitalWrite(HIGH);

    _delay_ms(5);
}

void nrf24_powerUpTx()
{
    nrf24_configRegister(STATUS,(1<<RX_DR)|(1<<TX_DS)|(1<<MAX_RT));

    nrf24_configRegister(CONFIG,nrf24_CONFIG|((1<<PWR_UP)|(0<<PRIM_RX)));

    _delay_ms(5);
}

void nrf24_powerDown()
{
    nrf24_ce_digitalWrite(LOW);
    nrf24_configRegister(CONFIG,nrf24_CONFIG);

    _delay_ms(5);
}

uint8_t spi_transfer(uint8_t tx)
{
    uint8_t i = 0;
    uint8_t rx = 0;

    nrf24_sck_digitalWrite(LOW);

    for(i=0;i<8;i++)
    {

        if(tx & (1<<(7-i)))
        {
            nrf24_mosi_digitalWrite(HIGH);
        }
        else
        {
            nrf24_mosi_digitalWrite(LOW);
        }

        nrf24_sck_digitalWrite(HIGH);

        rx = rx << 1;
        if(nrf24_miso_digitalRead())
        {
            rx |= 0x01;
        }

        nrf24_sck_digitalWrite(LOW);

    }

    return rx;
}

/* send and receive multiple bytes over SPI */
void nrf24_transferSync(uint8_t* dataout,uint8_t* datain,uint8_t len)
{
    uint8_t i;

    for(i=0;i<len;i++)
    {
        datain[i] = spi_transfer(dataout[i]);
    }

}

/* send multiple bytes over SPI */
void nrf24_transmitSync(uint8_t* dataout,uint8_t len)
{
    uint8_t i;

    for(i=0;i<len;i++)
    {
        spi_transfer(dataout[i]);
    }

}

/* Clocks only one byte into the given nrf24 register */
void nrf24_configRegister(uint8_t reg, uint8_t value)
{
    nrf24_csn_digitalWrite(LOW);
    spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
    spi_transfer(value);
    nrf24_csn_digitalWrite(HIGH);
}

/* Read single register from nrf24 */
void nrf24_readRegister(uint8_t reg, uint8_t* value, uint8_t len)
{
    nrf24_csn_digitalWrite(LOW);
    spi_transfer(R_REGISTER | (REGISTER_MASK & reg));
    nrf24_transferSync(value,value,len);
    nrf24_csn_digitalWrite(HIGH);
}

```

```c
/* Write to a single register of nrf24 */
void nrf24_writeRegister(uint8_t reg, uint8_t* value, uint8_t len)
{
    nrf24_csn_digitalWrite(LOW);
    spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
    nrf24_transmitSync(value,len);
    nrf24_csn_digitalWrite(HIGH);
}

/* Check single register from nrf24 */
bool nrf24_checkRegister(uint8_t reg, uint8_t desiredValue, uint8_t len)
{
    uint8_t registerValue;
    nrf24_readRegister(reg,&registerValue,len);
    if (registerValue==desiredValue) { return true; } else { return false; }
}

#define RF_DDR  DDRD
#define RF_PORT PORTD
#define RF_PIN  PIND

#define CE_CSN_DDR  DDRC
#define CE_CSN_PORT PORTC
#define CE_CSN_PIN  PINC

#define MISO_BIT_POS    0
#define MOSI_BIT_POS    1
#define SCK_BIT_POS     4

#define CE_BIT_POS      0
#define CSN_BIT_POS     1

#define set_bit(reg,bit) reg |= (1<<bit)
#define clr_bit(reg,bit) reg &= ~(1<<bit)
#define check_bit(reg,bit) (reg&(1<<bit))

/* ------------------------------------------------------------------------ */

void nrf24_setupPins()
{
    set_bit(CE_CSN_DDR, CE_BIT_POS); // CE output
    set_bit(CE_CSN_DDR, CSN_BIT_POS); // CSN output

    clr_bit(RF_DDR, MISO_BIT_POS); // MISO input
    set_bit(RF_DDR, MOSI_BIT_POS); // MOSI output
    set_bit(RF_DDR, SCK_BIT_POS); // SCK output
}
/* ------------------------------------------------------------------------ */
void nrf24_ce_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(CE_CSN_PORT, CE_BIT_POS);
    }
    else
    {
        clr_bit(CE_CSN_PORT, CE_BIT_POS);
    }
}
/* ------------------------------------------------------------------------- */
void nrf24_csn_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(CE_CSN_PORT, CSN_BIT_POS);
    }
    else
    {
        clr_bit(CE_CSN_PORT, CSN_BIT_POS);
    }
}
/* ------------------------------------------------------------------------- */
void nrf24_sck_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(RF_PORT, SCK_BIT_POS);
    }
    else
    {
        clr_bit(RF_PORT, SCK_BIT_POS);
    }
}
/* ------------------------------------------------------------------------- */
void nrf24_mosi_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(RF_PORT, MOSI_BIT_POS);
    }
    else
    {
        clr_bit(RF_PORT, MOSI_BIT_POS);
    }
}
/* ------------------------------------------------------------------------- */
uint8_t nrf24_miso_digitalRead()
{
    return check_bit(RF_PIN, MISO_BIT_POS);
}
/* ------------------------------------------------------------------------- */

```