```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); }

    }
}


```

```
void initIO(){
    /*
        Input/Output pin initialization
        1 : OUTPUT | 0 : INPUT | 0b76543210 Bit order
        ATTACHMENTS
            RED LED     : PD7                 |   OUTPUT
            GREEN LED   : PB0                 |   OUTPUT
        HC-05
            TX          : PD0 (RX ATMEGA)     |   INPUT
            RX          : PD1 (TX ATMEGA)     |   OUTPUT
            KEY/ENABLE  : PD2                 |   OUTPUT
            STATE       : PC5                 |   INPUT
        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;
}
```

```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
```

```c
                     UNDEFINED_COMMAND_CODE;
                    lastMessageCommandType = commandList[*(headerStart+5)];
                    lastMessagePID = *(headerStart+1);

                    uint8_t* parameterStart = headerStart+6;

                    for (uint8_t x = 0; x < 12; x++) {
                        realloc(parameter[x].startingPointer, *parameterStart);
                        parameter[x].byteLength = *parameterStart;
                        memcpy(parameter[x].startingPointer,parameterStart+1, *parameterStart);
                        parameterStart+=((*parameterStart)+1);
                        if (parameterStart>=(footerEnd-2)) break;
                    }

                    return SUCCESFUL_DECOMPOSITION;
                }
            }
        }
    }
    return WRONG_HEADER_SEGMENTATION;
}

CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t parameterCount, uint8_t targetBoardID){

    memset(command_buffer, 0, 32);

    command_buffer[0] = SOH;
    if (lastMessagePID==0xFF) { lastMessagePID++; } else { lastMessagePID = 0; }
    command_buffer[1] = lastMessagePID;
    command_buffer[2] = targetBoardID;
    command_buffer[3] = currentModuleID;
    command_buffer[4] = STX;
    command_buffer[5] = targetTypeID;

    if (parameterCount>12) return PARAMETER_COUNT_OVERSIZE;

    uint8_t* parameterStart = &command_buffer[6];

    for (uint8_t x = 0; x < parameterCount; x++){
        *parameterStart = parameter[x].byteLength;
        memcpy(parameterStart+1, parameter[x].startingPointer, parameter[x].byteLength);
        parameterStart+=(parameter[x].byteLength)+1;
    }

    crc_t crc;
    crc = crc_init();
    uint8_t crc_length = ((parameterStart)-(&command_buffer[0]));
    crc = crc_update(crc, &command_buffer[0], crc_length);
    crc = crc_finalize(crc);

    *parameterStart = ETX;
    *(parameterStart+1) = crc;
    *(parameterStart+2) = ETB;

    return SUCCESFUL_COMPOSITION;
}

void HandleAvailableCommand(){
    lastMessageCommandType.handlerFunction();
}

RF_TransmissionStatus RetransmissionToModule(){
    nrf24_initRF_SAFE((lastTargetModuleID-1), TRANSMIT);    // CONNECTION TO MODULE: GENERAL RF CHANNEL 112, (lastTargetModuleID-1) offset 1
    nrf24_send(command_buffer);
    while(nrf24_isSending());

    uint8_t messageStatus = nrf24_lastMessageStatus();
    if(messageStatus == NRF24_TRANSMISSON_OK) { return RF_SUCCESFUL_TRANSMISSION; }
    else if(messageStatus == NRF24_MESSAGE_LOST) { return RF_UNREACHEABLE_MODULE;}
    return RF_UNREACHEABLE_MODULE;
}

void RetransmissionToPhone(){
    transmitMessageSync(command_buffer, 32);
}



void writeParameterValue(uint8_t parameterIndex, uint8_t* parameterData, uint8_t parameterByteLength){
    parameter[parameterIndex].startingPointer = (uint8_t*) realloc(parameter[parameterIndex].startingPointer, parameterByteLength);
    memcpy(parameter[parameterIndex].startingPointer, parameterData, parameterByteLength);
    parameter[parameterIndex].byteLength = parameterByteLength;
}

void UPDATE_ALL_DEVICES_VALUE_H() {}
void UPDATE_DEVICE_VALUE_H() {}
void GET_ALL_DEVICES_VALUE_H() {
    _delay_ms(100);

    uint8_t boardState[2];

    ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, POWER_MODULE);
    nrf24_initRF_SAFE(POWER_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE: GENERAL RF CHANNEL 112
    nrf24_send(command_buffer);
```

```c
    while(nrf24_isSending());

    uint8_t messageStatus = nrf24_lastMessageStatus();
    if(messageStatus == NRF24_TRANSMISSON_OK) { boardState[0] = 0xFF; }
    else if(messageStatus == NRF24_MESSAGE_LOST) { boardState[0]= 0x00; }

    _delay_ms(50);

    ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, MOTOR_MODULE);
    nrf24_initRF_SAFE(MOTORIZED_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE: GENERAL RF CHANNEL 112
    nrf24_send(command_buffer);
    while(nrf24_isSending());

    uint8_t messageStatusSecond = nrf24_lastMessageStatus();
    if(messageStatusSecond == NRF24_TRANSMISSON_OK) { boardState[1] = 0xFF; }
    else if(messageStatusSecond == NRF24_MESSAGE_LOST) { boardState[1]= 0x00; }


    writeParameterValue(0, &boardState[0], 1);
    writeParameterValue(1, &boardState[1], 1);
    ComposeMessageToBuffer(UPDATE_ALL_DEVICES_VALUE_ID, 2, PHONE_MODULE);  // PHONE_MODULE should be lastTransmitterModuleID
    transmitMessageSync(command_buffer, 32);
}

void GET_DEVICE_VALUE_H() {
    _delay_ms(100);
    uint8_t deviceIndex = *((uint8_t*)parameter[0].startingPointer);
    uint8_t deviceValue;

    switch(deviceIndex){
        case 0:
            ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, POWER_MODULE);
            nrf24_initRF_SAFE(POWER_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE: GENERAL RF CHANNEL 112
            nrf24_send(command_buffer);
            while(nrf24_isSending());

            uint8_t messageStatus = nrf24_lastMessageStatus();
            if(messageStatus == NRF24_TRANSMISSON_OK) { deviceValue = 0xFF; }
            else if(messageStatus == NRF24_MESSAGE_LOST) { deviceValue= 0x00; }
            break;
        case 1:
            ComposeMessageToBuffer(MESSAGE_STATUS_ID, 0, MOTOR_MODULE);
            nrf24_initRF_SAFE(MOTORIZED_BOARD_RF, TRANSMIT);    // CONNECTION TO MODULE: GENERAL RF CHANNEL 112
            nrf24_send(command_buffer);
            while(nrf24_isSending());

            uint8_t messageStatusSecond = nrf24_lastMessageStatus();
            if(messageStatusSecond == NRF24_TRANSMISSON_OK) { deviceValue =
              0xFF; }
            else if(messageStatusSecond == NRF24_MESSAGE_LOST) { deviceValue=
              0x00; }
            break;
    }

    writeParameterValue(0, &deviceIndex, 1);
    writeParameterValue(1, &deviceValue, 2);

    ComposeMessageToBuffer(UPDATE_DEVICE_VALUE_ID, 2, PHONE_MODULE);  //
      PHONE_MODULE should be lastTransmitterModuleID

    transmitMessageSync(command_buffer, 32);
}
void MESSAGE_STATUS_H() {}
```

```c


#ifndef COMMAND_HANDLER_H_
#define COMMAND_HANDLER_H_

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

#ifndef F_CPU
#define F_CPU               16000000UL
#endif

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

typedef struct CommandType {
    void (*handlerFunction)();
} CommandType;

typedef enum   {
    SUCCESFUL_DECOMPOSITION,
    WRONG_HEADER_SEGMENTATION,
    WRONG_FOOTER_SEGMENTATION,
    WRONG_CHECKSUM_CONSISTENCY,
    WRONG_MODULE_ID,
    UNDEFINED_COMMAND_CODE,
    PARAMETER_DATA_OVERFLOW,
    PARAMETER_COUNT_OVERSIZE,
    RETRANSMISSION_FAILED,
    SUCCESFUL_RETRANSMISSION,
    SUCCESFUL_COMPOSITION
} CommandStatus;


typedef enum {
    RF_SUCCESFUL_TRANSMISSION,
    RF_UNREACHEABLE_MODULE,
    RF_ACKNOWLEDGE_FAILED
} RF_TransmissionStatus;

typedef enum   {
    UPDATE_ALL_DEVICES_VALUE_ID,
    UPDATE_DEVICE_VALUE_ID,
    GET_ALL_DEVICES_VALUE_ID,
    GET_DEVICE_VALUE_ID,
    MESSAGE_STATUS_ID
} CommandTypeID;

typedef struct {
    void *startingPointer;
    uint8_t byteLength;
} Parameter;

typedef enum   {
    PHONE_MODULE = 0x00,
    MAIN_MODULE = 0x01,
    POWER_MODULE = 0x02,
    MOTOR_MODULE = 0x03,
} ModuleInternalCode;


#define currentModuleID MAIN_MODULE

#define SOH 0x01
#define STX 0x02
#define ETX 0x03
#define ETB 0x17
#define ON_STATE    0xFF
#define OFF_STATE   0x00

#define AVAILABLE_DEVICES 4
uint16_t device_value[AVAILABLE_DEVICES];

uint8_t *command_buffer;
Parameter parameter[12];
bool memoryInitialized;

uint8_t lastMessagePID;
uint8_t lastTargetModuleID;
uint8_t lastTransmitterModuleID;
CommandType lastMessageCommandType;

extern bool initliazeMemory();
extern void UPDATE_ALL_DEVICES_VALUE_H(), UPDATE_DEVICE_VALUE_H(), GET_ALL_DEVICES_VALUE_H(), GET_DEVICE_VALUE_H(), MESSAGE_STATUS_H();
extern CommandStatus DecomposeMessageFromBuffer();
extern void HandleAvailableCommand();
extern RF_TransmissionStatus RetransmissionToModule();
extern CommandStatus ComposeMessageToBuffer(CommandTypeID targetTypeID, uint8_t parameterCount, uint8_t targetBoardID);
void writeParameterValue(uint8_t parameterIndex, uint8_t* parameterData, uint8_t parameterByteLength);
void RetransmissionToPhone();
#endif /* COMMAND_HANDLER_H_ */
```

```c


#include "UART_Bluetooth.h"
#include <avr/io.h>
#include <avr/interrupt.h>
#include "Command_Handler.h"
#include "nrf24.h"
#include <stdlib.h>
#include <string.h>

uint8_t* uartBufferPos;
uint8_t* uartTxMessageEnd;
bool commandAvailable;

void initBluetoothUart(){
    // UART Initialization : 8-bit : No parity bit : 1 stop bit
    UBRR0H = (BRC >> 8); UBRR0L =  BRC;             // UART BAUDRATE
    UCSR0A |= (1 << U2X0);                          // DOUBLE UART SPEED
    UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);        // 8-BIT CHARACTER SIZE

    // Setup UART buffer
    initliazeMemory();
    uartBufferPos = command_buffer;
}

void transmitMessage(uint8_t* message, uint8_t length){
    while (!(UCSR0A & (1<<UDRE0)));
    uartBufferPos = command_buffer;
    uartTxMessageEnd = (command_buffer+length);
    memcpy(command_buffer, message, length);
    UCSR0A |= (1<<TXC0) | (1<<RXC0);
    UCSR0B |= (1<<TXEN0) | (1<<TXCIE0);
    UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);

    uartBufferPos++;
    UDR0 = *(command_buffer);
}

void transmitMessageSync(uint8_t* message, uint8_t length){
    while (!(UCSR0A & (1<<UDRE0)));
    uartBufferPos = command_buffer;
    uartTxMessageEnd = (command_buffer+length);
    memcpy(command_buffer, message, length);
    UCSR0A |= (1<<TXC0) | (1<<RXC0);
    UCSR0B |= (1<<TXEN0) | (1<<TXCIE0);
    UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);
    sei();

    uartBufferPos++;
    UDR0 = *(command_buffer);

    while (transmissionState());
```

```c

}

bool transmissionState(){
    // True : Currently transmitting | False : Transmission finished
    if (uartBufferPos!=uartTxMessageEnd)
    {
        return true;
    }
    else
    {
        return false;
    }
}


void setupReceiveMode(){
    while (!(UCSR0A & (1<<UDRE0)));
    uartBufferPos = command_buffer;

    UCSR0A |= (1<<RXC0) | (1<<TXC0);
    UCSR0B &=~(1<<TXEN0) &~(1<<TXCIE0);
    UCSR0B |= (1<<RXEN0) | (1<<RXCIE0);
    sei();
}

bool catchModuleReply(){
    nrf24_initRF_SAFE((lastTargetModuleID-1), RECEIVE); // CONNECTION TO MODULE: GENERAL RF CHANNEL 112 (lastTargetModuleID-1) offset 1
    uint8_t targetModuleID = lastTargetModuleID;
    uint8_t RF_TIME_OUT;
    while(RF_TIME_OUT!=0xFF)
    {
        if(nrf24_dataReady()){
            nrf24_getData(command_buffer);
            CommandStatus status = DecomposeMessageFromBuffer();
            if (status==SUCCESFUL_DECOMPOSITION&&lastTargetModuleID==targetModuleID) {
              transmitMessageSync(command_buffer, 32);
              return true;
            }
        }
        RF_TIME_OUT++; _delay_ms(2);
    }
    return false;
}

void processReceivedLine(){
    commandAvailable = false;

    CommandStatus status = DecomposeMessageFromBuffer();
    if(status==SUCCESFUL_DECOMPOSITION) {
        if (lastTargetModuleID==MAIN_MODULE){
            //Executed by main module
            HandleAvailableCommand();
        } else {
            //Retransmitted to other module

            RF_TransmissionStatus RF_Status = RetransmissionToModule();

            //Catch module reply

            //bool didModuleRelpy = catchModuleReply();

            // Send RF STATUS
            switch (RF_Status) {
                case RF_UNREACHEABLE_MODULE:
                writeParameterValue(0, &(uint8_t){RETRANSMISSION_FAILED}, 1);
                break;
                case RF_ACKNOWLEDGE_FAILED:
                writeParameterValue(0, &(uint8_t){RETRANSMISSION_FAILED}, 1);
                break;
                case RF_SUCCESFUL_TRANSMISSION:
                writeParameterValue(0, &(uint8_t){SUCCESFUL_RETRANSMISSION}, 1);
                break;
            }
            ComposeMessageToBuffer(MESSAGE_STATUS_ID, 1, PHONE_MODULE);
            transmitMessageSync(command_buffer, 32);


        }
    }else {
}


}

void disableUART(){
    UCSR0B &=~(1<<TXEN0) &~(1<<TXCIE0);
    UCSR0B &=~(1<<RXEN0) &~(1<<RXCIE0);
}

ISR(USART_TX_vect){
    if (uartBufferPos!=uartTxMessageEnd){
        UDR0 = *uartBufferPos;
        uartBufferPos++;
    }
}

ISR(USART_RX_vect){
    if(uartBufferPos!=(command_buffer+uartBufferSize)) {
        *uartBufferPos=UDR0;
        if ((*uartBufferPos==ETB)&&(DecomposeMessageFromBuffer()
          ==SUCCESFUL_DECOMPOSITION)) {
            disableUART(); commandAvailable = true;
        }
        else if(*uartBufferPos==uartCarriageReturnChar) {

            bool hasToReturnCarriage = true;
            uint8_t* savedUartBufferPos = uartBufferPos+1;

            for (uint8_t x = 1; x < 4; x++) {
                if ((uartBufferPos-x)<command_buffer) uartBufferPos = command_buffer+(uartBufferSize-1);
                if (*(uartBufferPos-x)!=uartCarriageReturnChar) { hasToReturnCarriage = false; break; }
            }
            if (hasToReturnCarriage) {
                 uartBufferPos = command_buffer;

            } else {
                uartBufferPos = savedUartBufferPos;
            }

        } else {
            uartBufferPos++;
        }

    } else {
        uartBufferPos = command_buffer;
        *uartBufferPos=UDR0;
    }
}
```

```c


#ifndef UART_BLUETOOTH_H_
#define UART_BLUETOOTH_H_


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

#ifndef F_CPU
#define F_CPU           16000000UL
#endif

#ifndef BAUD
#define BAUD            9600
#endif

#ifndef BRC
#define BRC             F_CPU/8/BAUD-1
#endif

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

#define uartBufferSize          32
#define uartEndMsgChar          '$'
#define uartCarriageReturnChar  0x7F

#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


extern bool commandAvailable;

extern void initBluetoothUart();
extern void transmitMessage(uint8_t* message, uint8_t length);
extern void transmitMessageSync(uint8_t* message, uint8_t length);
extern bool transmissionState();
extern void setupReceiveMode();
extern void processReceivedLine();
extern void disableUART();


```

```

#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;
}

```

```
/**
 * \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.
 *
 * This is not used anywhere in the generated code, but it may be used by the
 * application code to call algorithm-specific code, if desired.
 */
#define CRC_ALGO_BIT_BY_BIT_FAST 1


/**
 * The type of the CRC values.
 *
 * This type must be big enough to contain at least 8 bits.
 */
typedef uint_fast8_t crc_t;


/**
 * Calculate the initial crc value.
 *
 * \return     The initial crc value.
 */
static inline crc_t crc_init(void)
{
    return 0x00;
}


/**
 * Update the crc value with new data.
 *
 * \param[in] crc      The current crc value.
 * \param[in] data     Pointer to a buffer of \a data_len bytes.
 * \param[in] data_len Number of bytes in the \a data buffer.
 * \return             The updated crc value.
 */
crc_t crc_update(crc_t crc, const void *data, size_t data_len);


/**
 * Calculate the final crc value.
 *
 * \param[in] crc  The current crc value.
 * \return     The final crc value.
 */
static inline crc_t crc_finalize(crc_t crc)
{
    return crc;
}


#ifdef __cplusplus
}           /* closing brace for extern "C" */
#endif
```

```c

#endif      /* CRC_H */

```

```c

#define UCPHA0 1

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

volatile uint8_t payload_len;
volatile uint8_t selectedChannel;

uint8_t MOTORIZED_BOARD_ADDR[5] =   {0xF0,0xF0,0xF0,0xF0,0xC9};
uint8_t MAIN_BOARD_ADDR[5] =            {0xA4,0xA4,0xA4,0xA4,0xA4};
uint8_t POWER_BOARD_ADDR[5] =       {0xF0,0xF0,0xF0,0xF0,0xF0};

uint8_t NULL_ADDR[5] =      {0x00,0x00,0x00,0x00,0x00};

uint8_t* BOARD_ADDRESS[3] = {&MAIN_BOARD_ADDR[0], &POWER_BOARD_ADDR[0], &MOTORIZED_BOARD_ADDR[0]};
uint8_t* CURRENT_BOARD_ADDRESS = &MAIN_BOARD_ADDR[0];

const uint8_t GENERAL_RF_CHANNEL = 112;


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(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(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;}
}




void faultyRF_Alarm(){
    CLEAR_FAULTY_RF_LED;
    for (uint8_t x = 0; x < 6; x++)
    {
        FLIP_FAULTY_RF_LED;
        _delay_ms(125);
    }
    _delay_ms(250);
}


```

```

/* 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);
}

/* Set the secondary RX address */
void nrf24_secondary_rx_address(uint8_t * adr)
{
    nrf24_ce_digitalWrite(LOW);
    nrf24_writeRegister(RX_ADDR_P2,adr,1);  // One byte long
    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 ... */
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()
{
```

```c
    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);
}

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)));
}

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

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)
```

```c
{
    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);
}

/* 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  DDRC
#define RF_PORT PORTC
#define RF_PIN  PINC

#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(RF_DDR,0); // CE output
    set_bit(RF_DDR,1); // CSN output
    set_bit(RF_DDR,2); // SCK output
    set_bit(RF_DDR,3); // MOSI output
    clr_bit(RF_DDR,4); // MISO input
}
/* -------------------------------------------------------------------------- */
void nrf24_ce_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(RF_PORT,0);
    }
    else
    {
        clr_bit(RF_PORT,0);
    }
}
/* -------------------------------------------------------------------------- */
void nrf24_csn_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(RF_PORT,1);
    }
    else
    {
        clr_bit(RF_PORT,1);
    }
}
/* -------------------------------------------------------------------------- */
void nrf24_sck_digitalWrite(uint8_t state)
{
    if(state)
    {
        set_bit(RF_PORT,2);
    }
    else
    {
        clr_bit(RF_PORT,2);
    }
}
/* -------------------------------------------------------------------------- */
void nrf24_mosi_digitalWrite(uint8_t state)
{
    if(state)
    {
```

```c
        set_bit(RF_PORT,3);
    }
    else
    {
        clr_bit(RF_PORT,3);
    }
}
/* ------------------------------------------------------------------------- */
uint8_t nrf24_miso_digitalRead()
{
    return check_bit(RF_PIN,4);
}
/* ------------------------------------------------------------------------- */

void nrf24_initRF_SAFE(uint8_t boardIndex,TransmissionMode initMode){

    initliazeMemory();
    bool successfulRfInit = false;

    while(successfulRfInit==false){
        nrf24_powerDown();
        nrf24_init();
        nrf24_config(GENERAL_RF_CHANNEL,32);
        if (nrf24_checkConfig()) { successfulRfInit = true; } else { faultyRF_Alarm(); }
    }



    if (initMode==RECEIVE){
        nrf24_tx_address(CURRENT_BOARD_ADDRESS);
        nrf24_rx_address(BOARD_ADDRESS[boardIndex]);
        }else{
        nrf24_tx_address(BOARD_ADDRESS[boardIndex]);
        nrf24_rx_address(CURRENT_BOARD_ADDRESS);
    }


    nrf24_powerUpRx();
}
```

```c
#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);
```

```

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();

void faultyRF_Alarm();

uint8_t selectedTX_ADDRESS;
uint8_t selectedRX_ADDRESS;

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);

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

```