**Приложение Б**

**Листинг программы**

**main.c**

// Аннаев А.К., ЦНТУ, 2018

// Розробка програмного забезпечення ретранслятора USB -UART

#include <stdbool.h>

#include <stdint.h>

#include "driverlib/interrupt.h"

#include "driverlib/sysctl.h"

#include "driverlib/systick.h"

#include "utils/pinout.h"

#include "exchange.h"

#include "resource.h"

#include "utils/wakeConverter.h"

#include "usb\_interface.h"

#include "uart\_interface.h"

uint32\_t g\_ui32SysClock = 0;

uint32\_t ui32PLLRate;

uint32\_t g\_ui32SysTickCount = 0;

#ifdef DEBUG

void

\_\_error\_\_(char \*pcFilename, uint32\_t ui32Line)

{

while(1)

{

}

}

#endif

// Обработчик SysTick

void SysTickIntHandler(void)

{

g\_ui32SysTickCount++;

}

int main(void)

{

//Конфигурирование системного тактирования

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ | SYSCTL\_OSC\_MAIN |

SYSCTL\_USE\_PLL | SYSCTL\_CFG\_VCO\_480),

120000000); SysCtlVCOGet(SYSCTL\_XTAL\_25MHZ, &ui32PLLRate);

PinoutSet(false, true);

SysTickPeriodSet(g\_ui32SysClock / SYSTICKS\_PER\_SECOND);

SysTickIntEnable();

SysTickEnable();

InitializeUART();

InitializeUSB();

while(true) {

if(g\_ui32Flags & COMMAND\_STATUS\_UPDATE) {

IntMasterDisable();

g\_ui32Flags &= ~COMMAND\_STATUS\_UPDATE;

IntMasterEnable();

}

if(g\_ui32Flags & COMMAND\_DATA\_RECEIVED ) {

ProcessData();

g\_ui32Flags &= ~COMMAND\_DATA\_RECEIVED ;

}

if(g\_ui32Flags & COMMAND\_TRANSMIT) {

StartTesting();

g\_ui32Flags &= ~COMMAND\_TRANSMIT ;

}

}

}

**usb\_inteface.c**

#include <stdbool.h>

#include <stdint.h>

#include "inc/hw\_types.h"

#include "driverlib/usb.h"

#include "usblib/usblib.h"

#include "usblib/usbcdc.h"

#include "usblib/usb-ids.h"

#include "usblib/device/usbdevice.h"

#include "usblib/device/usbdcdc.h"

#include "usblib/device/usbdcomp.h"

#include "usb\_interface.h"

const uint8\_t g\_pui8LangDescriptor[] = {

4,

USB\_DTYPE\_STRING,

USBShort(USB\_LANG\_EN\_US)

};

const uint8\_t g\_pui8ManufacturerString[] = {

(17 + 1) \* 2,

USB\_DTYPE\_STRING,

'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,

't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,

};

const uint8\_t g\_pui8ProductString[] = {

2 + (16 \* 2),

USB\_DTYPE\_STRING,

'V', 0, 'i', 0, 'r', 0, 't', 0, 'u', 0, 'a', 0, 'l', 0, ' ', 0,

'C', 0, 'O', 0, 'M', 0, ' ', 0, 'P', 0, 'o', 0, 'r', 0, 't', 0

};

const uint8\_t g\_pui8SerialNumberString[] = {

2 + (8 \* 2),

USB\_DTYPE\_STRING,

'1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0

};

const uint8\_t g\_pui8ControlInterfaceString[] = {

2 + (21 \* 2),

USB\_DTYPE\_STRING,

'A', 0, 'C', 0, 'M', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 't', 0,

'r', 0, 'o', 0, 'l', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0,

'r', 0, 'f', 0, 'a', 0, 'c', 0, 'e', 0

};

const uint8\_t g\_pui8ConfigString[] = {

2 + (26 \* 2),

USB\_DTYPE\_STRING,

'S', 0, 'e', 0, 'l', 0, 'f', 0, ' ', 0, 'P', 0, 'o', 0, 'w', 0,

'e', 0, 'r', 0, 'e', 0, 'd', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0,

'f', 0, 'i', 0, 'g', 0, 'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0,

'o', 0, 'n', 0

};

const uint8\_t \* const g\_pui8StringDescriptors[] = {

g\_pui8LangDescriptor,

g\_pui8ManufacturerString,

g\_pui8ProductString,

g\_pui8SerialNumberString,

g\_pui8ControlInterfaceString,

g\_pui8ConfigString

};

#define NUM\_STRING\_DESCRIPTORS (sizeof(g\_pui8StringDescriptors) / \

sizeof(uint8\_t \*))

tUSBDCDCDevice g\_psCDCDevice = {

USB\_VID\_TI\_1CBE,

USB\_PID\_SERIAL,

0,

USB\_CONF\_ATTR\_SELF\_PWR,

ControlHandler,

(void \*)&g\_psCDCDevice,

USBBufferEventCallback,

(void \*)&g\_psRxBuffer,

USBBufferEventCallback,

(void \*)&g\_psTxBuffer,

g\_pui8StringDescriptors,

NUM\_STRING\_DESCRIPTORS

};

uint8\_t TEMP\_BULK\_BUFFER[UART\_BUFFER\_SIZE];

uint8\_t g\_ppui8USBRxBuffer[UART\_BUFFER\_SIZE];

tUSBBuffer g\_psRxBuffer = {

false, // This is a receive buffer.

ReceiveDataHandler, // pfnCallback

(void \*)&g\_psCDCDevice, // Callback data is our device pointer.

USBDCDCPacketRead, // pfnTransfer

USBDCDCRxPacketAvailable, // pfnAvailable

(void \*)&g\_psCDCDevice, // pvHandle

g\_ppui8USBRxBuffer, // pcBuffer

UART\_BUFFER\_SIZE, // ulBufferSize

};

uint8\_t g\_ppcUSBTxBuffer[UART\_BUFFER\_SIZE];

tUSBBuffer g\_psTxBuffer = {

true, // This is a transmit buffer.

TransmitDataHandler, // pfnCallback

(void \*)&g\_psCDCDevice, // Callback data is our device pointer.

USBDCDCPacketWrite, // pfnTransfer

USBDCDCTxPacketAvailable, // pfnAvailable

(void \*)&g\_psCDCDevice, // pvHandle

g\_ppcUSBTxBuffer, // pcBuffer

UART\_BUFFER\_SIZE, // ulBufferSize

};

**usb\_interface.h**

#ifndef USB\_INTERFACE\_H

#define USB\_INTERFACE\_H

#include <stdint.h>

#include "usblib/usblib.h"

#include "inc/hw\_types.h"

#include "driverlib/usb.h"

#include "usblib/usblib.h"

#include "usblib/usbcdc.h"

#include "usblib/usb-ids.h"

#include "usblib/device/usbdevice.h"

#include "usblib/device/usbdcdc.h"

#include "usblib/device/usbdcomp.h"

#define RX\_BUFFER\_SIZE 1024

#define TX\_BUFFER\_SIZE 1024

#define UART\_BUFFER\_SIZE 1024

#define DESCRIPTOR\_DATA\_SIZE (COMPOSITE\_DCDC\_SIZE + COMPOSITE\_DCDC\_SIZE)

extern uint8\_t RXBUFFER[RX\_BUFFER\_SIZE];

extern uint8\_t TXBUFFER[TX\_BUFFER\_SIZE];

extern uint8\_t TEMP\_BULK\_BUFFER[UART\_BUFFER\_SIZE];

#define COMMAND\_DATA\_RECEIVED 0x00000001

#define COMMAND\_STATUS\_UPDATE 0x00000002

#define COMMAND\_TRANSMIT 0x00000004

extern uint32\_t g\_ui32Flags;

extern bool g\_bUSBConfigured;

extern bool IsTransmit;

extern uint32\_t RxHandlerEcho

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData);

extern uint32\_t ReceiveDataHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData);

extern uint32\_t TxHandlerEcho

(void \*pvlCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData);

extern uint32\_t TransmitDataHandler

(void \*pvlCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData);

extern uint32\_t ControlHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData);

extern uint32\_t EventHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgData, void \*pvMsgData);

extern tUSBBuffer g\_psTxBuffer;

extern tUSBBuffer g\_psRxBuffer;

extern tUSBDCDCDevice g\_psCDCDevice;

extern uint8\_t g\_pui8USBTxBuffer[];

extern uint8\_t g\_pui8USBRxBuffer[];

bool TransmitData(uint8\_t \*src, uint16\_t src\_len);

void InitializeUSB();

void CheckForSerialStateChange(const tUSBDCDCDevice \*psDevice, int32\_t i32Errors);

void PackageTxLedOn();

void PackageTxLedOff();

void HostRequestLedOn();

void HostRequestLedOff();

#endif // USB\_INTERFACE\_H

**usb\_api.c**

#include <stdbool.h>

#include <stdint.h>

#include "inc/hw\_uart.h"

#include "inc/hw\_memmap.h"

#include "driverlib/pin\_map.h"

#include "driverlib/interrupt.h"

#include "driverlib/gpio.h"

#include "driverlib/sysctl.h"

#include "driverlib/uart.h"

#include "utils/common.h"

#include "utils/crc.h"

#include "usb\_interface.h"

#include "uart\_interface.h"

#include "utils/exchange\_protocol.h"

#include "resource.h"

uint32\_t g\_ui32Flags = 0;

bool IsTransmit = true;

uint8\_t RXBUFFER[RX\_BUFFER\_SIZE];

uint8\_t TXBUFFER[TX\_BUFFER\_SIZE];

bool g\_bUSBConfigured = false;

void PackageTxLedOn()

{

GPIOPinWrite(GPIO\_PORTN\_BASE, GPIO\_PIN\_1, GPIO\_PIN\_1);

}

void PackageTxLedOff()

{

GPIOPinWrite(GPIO\_PORTN\_BASE, GPIO\_PIN\_1, 0);

}

void HostRequestLedOn()

{

GPIOPinWrite(GPIO\_PORTN\_BASE, GPIO\_PIN\_0, 1);

}

void HostRequestLedOff()

{

GPIOPinWrite(GPIO\_PORTN\_BASE, GPIO\_PIN\_0, 0);

}

void CheckForSerialStateChange(const tUSBDCDCDevice \*psDevice, int32\_t i32Errors)

{

unsigned short usSerialState;

usSerialState = USB\_CDC\_SERIAL\_STATE\_TXCARRIER | USB\_CDC\_SERIAL\_STATE\_RXCARRIER;

if(i32Errors) {

//

// At least one error is being notified so translate from our hardware

// error bits into the correct state markers for the USB notification.

if(i32Errors & UART\_DR\_OE)

usSerialState |= USB\_CDC\_SERIAL\_STATE\_OVERRUN;

if(i32Errors & UART\_DR\_PE)

usSerialState |= USB\_CDC\_SERIAL\_STATE\_PARITY;

if(i32Errors & UART\_DR\_FE)

usSerialState |= USB\_CDC\_SERIAL\_STATE\_FRAMING;

if(i32Errors & UART\_DR\_BE)

usSerialState |= USB\_CDC\_SERIAL\_STATE\_BREAK;

USBDCDCSerialStateChange((void \*)psDevice, usSerialState);

}

}

bool SetLineCoding(tLineCoding \*psLineCoding)

{

uint32\_t ui32Config;

bool bRetcode;

bRetcode = true;

switch(psLineCoding->ui8Databits) {

case 5: ui32Config = UART\_CONFIG\_WLEN\_5;

break;

case 6: ui32Config = UART\_CONFIG\_WLEN\_6;

break;

case 7: ui32Config = UART\_CONFIG\_WLEN\_7;

break;

case 8: ui32Config = UART\_CONFIG\_WLEN\_8;

break;

default:

{

ui32Config = UART\_CONFIG\_WLEN\_8;

bRetcode = false;

break;

}

}

switch(psLineCoding->ui8Parity)

{

case USB\_CDC\_PARITY\_NONE: ui32Config |= UART\_CONFIG\_PAR\_NONE;

break;

case USB\_CDC\_PARITY\_ODD: ui32Config |= UART\_CONFIG\_PAR\_ODD;

break;

case USB\_CDC\_PARITY\_EVEN: ui32Config |= UART\_CONFIG\_PAR\_EVEN;

break;

case USB\_CDC\_PARITY\_MARK: ui32Config |= UART\_CONFIG\_PAR\_ONE;

break;

case USB\_CDC\_PARITY\_SPACE: ui32Config |= UART\_CONFIG\_PAR\_ZERO;

break;

default: {

ui32Config |= UART\_CONFIG\_PAR\_NONE;

bRetcode = false;

break;

}

}

// По умолчанию 1, 1.5 отсутствует

switch(psLineCoding->ui8Stop) {

case USB\_CDC\_STOP\_BITS\_1: ui32Config |= UART\_CONFIG\_STOP\_ONE;

break;

case USB\_CDC\_STOP\_BITS\_2: ui32Config |= UART\_CONFIG\_STOP\_TWO;

break;

default: {

ui32Config = UART\_CONFIG\_STOP\_ONE;

bRetcode |= false;

break;

}

}

SetUartConfig(psLineCoding->ui32Rate, ui32Config);

return(bRetcode);

}

void GetLineCoding(tLineCoding \*psLineCoding)

{

uint32\_t ui32Config, ui32Rate;

GetUartConfig(&ui32Rate,&ui32Config);

psLineCoding->ui32Rate = ui32Rate;

//

// Translate the configuration word length field into the format expected

// by the host.

switch(ui32Config & UART\_CONFIG\_WLEN\_MASK) {

case UART\_CONFIG\_WLEN\_8: psLineCoding->ui8Databits = 8;

break;

case UART\_CONFIG\_WLEN\_7: psLineCoding->ui8Databits = 7;

break;

case UART\_CONFIG\_WLEN\_6: psLineCoding->ui8Databits = 6;

break;

case UART\_CONFIG\_WLEN\_5: psLineCoding->ui8Databits = 5;

break;

}

switch(ui32Config & UART\_CONFIG\_PAR\_MASK)

{

case UART\_CONFIG\_PAR\_NONE: psLineCoding->ui8Parity = USB\_CDC\_PARITY\_NONE;

break;

case UART\_CONFIG\_PAR\_ODD: psLineCoding->ui8Parity = USB\_CDC\_PARITY\_ODD;

break;

case UART\_CONFIG\_PAR\_EVEN: psLineCoding->ui8Parity = USB\_CDC\_PARITY\_EVEN;

break;

case UART\_CONFIG\_PAR\_ONE:

psLineCoding->ui8Parity = USB\_CDC\_PARITY\_MARK;

break;

case UART\_CONFIG\_PAR\_ZERO: psLineCoding->ui8Parity = USB\_CDC\_PARITY\_SPACE;

break;

}

switch(ui32Config & UART\_CONFIG\_STOP\_MASK)

{

case UART\_CONFIG\_STOP\_ONE: psLineCoding->ui8Stop = USB\_CDC\_STOP\_BITS\_1;

break;

case UART\_CONFIG\_STOP\_TWO: psLineCoding->ui8Stop = USB\_CDC\_STOP\_BITS\_2;

break;

}

}

uint32\_t ControlHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData)

{

uint32\_t ui32IntsOff;

switch(ui32Event) {

case USB\_EVENT\_CONNECTED: {

g\_bUSBConfigured = true;

//

//

USBBufferFlush(&g\_psTxBuffer);

USBBufferFlush(&g\_psRxBuffer);

//

// Tell the main loop to update the display.

ui32IntsOff = IntMasterDisable();

g\_ui32Flags |= COMMAND\_STATUS\_UPDATE;

if(!ui32IntsOff) {

IntMasterEnable();

}

break;

}

case USB\_EVENT\_DISCONNECTED: {

g\_bUSBConfigured = false;

ui32IntsOff = IntMasterDisable();

g\_ui32Flags |= COMMAND\_STATUS\_UPDATE;

if(!ui32IntsOff)

IntMasterEnable();

break;

}

case USBD\_CDC\_EVENT\_GET\_LINE\_CODING: {

GetLineCoding(pvMsgData);

break;

}

case USBD\_CDC\_EVENT\_SET\_LINE\_CODING: {

SetLineCoding(pvMsgData);

break;

}

case USBD\_CDC\_EVENT\_SET\_CONTROL\_LINE\_STATE: {

SetControlLineState((unsigned short)ui32MsgValue);

break;

}

case USBD\_CDC\_EVENT\_SEND\_BREAK: {

SendBreak(true);

break;

}

case USBD\_CDC\_EVENT\_CLEAR\_BREAK: {

SendBreak(false);

break;

}

case USB\_EVENT\_SUSPEND:

case USB\_EVENT\_RESUME:

break;

default:

break;

}

return(0);

}

uint32\_t TransmitDataHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData)

{

switch(ui32Event) {

case USB\_EVENT\_TX\_COMPLETE:

{

IsTransmit = true;

HostRequestLedOn();

break;

}

default:

{

break;

}

}

return(0);

}

uint32\_t ReceiveDataHandler

(void \*pvCBData, uint32\_t ui32Event, uint32\_t ui32MsgValue, void \*pvMsgData){

const tUSBDCDCDevice \*psCDCDevice;

const tUSBBuffer \*pBufferRx;

static uint32\_t ui32ReceiveCounter;

static uint32\_t crc32;

static bool isFrameStartMarker;

static bool isStaffingMarker;

static bool isReceive;

switch(ui32Event)

{

case USB\_EVENT\_RX\_AVAILABLE:

{

psCDCDevice = (const tUSBDCDCDevice \*)pvCBData;

pBufferRx = (const tUSBBuffer \*)psCDCDevice->pvRxCBData;

uint32\_t ui32ReadDataLength = USBBufferRead(pBufferRx, TEMP\_BULK\_BUFFER,

USBBufferDataAvailable(pBufferRx)) ;

if(ui32ReadDataLength != 0)

{

for(int idx =0; idx < ui32ReadDataLength; idx++)

{

uint8\_t temp = TEMP\_BULK\_BUFFER[idx];

if(temp == FEND)

{

ui32ReceiveCounter = 0;

crc32 = PACK\_CRC32\_START;

isReceive = true;

isFrameStartMarker = true;

continue;

}

if(!isFrameStartMarker)

{

continue;

}

if( temp == FESC)

{

isStaffingMarker = true;

continue;

}

if(isStaffingMarker)

{

isStaffingMarker = false;

if(temp == TFEND)

{

RXBUFFER[ui32ReceiveCounter++] = FEND;

}

else if(temp == TFESC)

{

// если заменяем 0xDB

RXBUFFER[ui32ReceiveCounter++] = FESC;

}

else

{

// не корректное значение (нет стаффинга)

ui32ReceiveCounter = 0;

isReceive = false;

isFrameStartMarker = false;

crc32 = PACK\_CRC32\_START;

continue;

}

}else

{

RXBUFFER[ui32ReceiveCounter++] = temp;

}

uint32\_t size = 0;

if(ui32ReceiveCounter >= EXC\_HEADER\_SIZE)

{

size = BYTES\_TO\_U16(&RXBUFFER[LEN\_INDEX]);

}

if(ui32ReceiveCounter <= (EXC\_HEADER\_SIZE + size))

{

crc32 = crc32\_IEEE\_802\_step(RXBUFFER[ui32ReceiveCounter-1], crc32);

}

if(ui32ReceiveCounter == (EXCHANGE\_MIN\_PACK + size))

{

uint32\_t hash = BYTES\_TO\_U32(&RXBUFFER[ui32ReceiveCounter - EXC\_HASH\_SIZE]);

//

// Если контрольная сумма не совпала, отбрасываем пакет

if( hash == crc32)

{

isReceive = true;

isFrameStartMarker = false;

break;

}

else

{

ui32ReceiveCounter = 0;

isReceive = false;

isFrameStartMarker = false;

crc32 = PACK\_CRC32\_START;

}

}

}

}

if(isReceive)

{

isReceive = false;

g\_ui32Flags |= COMMAND\_DATA\_RECEIVED ;

}

}

break;

case USB\_EVENT\_DATA\_REMAINING:

return(0);

case USB\_EVENT\_REQUEST\_BUFFER:

return(0);

default:

break;

}

return(0);

}

void InitializeUSB()

{

g\_bUSBConfigured = false;

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_USB0);

USBBufferInit(&g\_psTxBuffer);

USBBufferInit(&g\_psRxBuffer);

USBDCDeatureSet(0, USBLIB\_FEATURE\_CPUCLK, &g\_ui32SysClock);

USBDCDFeatureSet(0, USBLIB\_FEATURE\_USBPLL, &ui32PLLRate);

USBDCDCInit(0, &g\_psCDCDevice);

}

uart\_interface.h

#ifndef UART\_INTERFACE\_H

#define UART\_INTERFACE\_H

#include <stdbool.h>

#include <stdint.h>

#define DEFAULT\_BIT\_RATE 115200

#define DEFAULT\_UART\_CONFIG (UART\_CONFIG\_WLEN\_8 | UART\_CONFIG\_PAR\_NONE | \

UART\_CONFIG\_STOP\_ONE)

#define UART\_SYSPERIPH SYSCTL\_PERIPH\_UART6

#define UART\_GPIO\_PERIPH SYSCTL\_PERIPH\_GPIOP

#define UART\_PORT\_BASE GPIO\_PORTP\_BASE

#define UART\_RX\_CHANNEL GPIO\_PP0\_U6RX

#define UART\_TX\_CHANNEL GPIO\_PP1\_U6TX

#define UART\_INT\_NUM INT\_UART6

#define UART\_BASE UART6\_BASE

#define TX\_GPIO\_PIN GPIO\_PIN\_1

#define RX\_GPIO\_PIN GPIO\_PIN\_0

#define MAX\_BAUD\_RATE 1000000

extern bool g\_bSendingBreak;

uint32\_t GetUartBits();

void InitializeUART();

void SetUartConfig(uint32\_t rate, uint32\_t config);

void GetUartConfig(uint32\_t \*rate, uint32\_t \*config);

void SetControlLineState(unsigned short usState);

void SendBreak(bool bSend);

int32\_t ReadUARTData(void);

#endif //UART\_INTERFACE\_H

**uart\_api.c**

#include <stdbool.h>

#include <stdint.h>

#include "inc/hw\_ints.h"

#include "inc/hw\_memmap.h"

#include "driverlib/gpio.h"

#include "driverlib/pin\_map.h"

#include "driverlib/interrupt.h"

#include "driverlib/sysctl.h"

#include "driverlib/uart.h"

#include "uart\_interface.h"

#include "usb\_interface.h"

#include "resource.h"

bool g\_bSendingBreak = false;

uint32\_t g\_ui32UARTTxCount = 0;

uint32\_t g\_ui32UARTRxCount = 0;

uint32\_t GetUartBits()

{

uint32\_t ui32Config, ui32Rate;

uint32\_t total\_bit\_amount = 0;

UARTConfigGetExpClk(UART\_BASE, g\_ui32SysClock, &ui32Rate, &ui32Config);

switch(ui32Config & UART\_CONFIG\_WLEN\_MASK)

{

case UART\_CONFIG\_WLEN\_8: total\_bit\_amount += 8;

break;

case UART\_CONFIG\_WLEN\_7: total\_bit\_amount += 7;

break;

case UART\_CONFIG\_WLEN\_6: total\_bit\_amount += 6;

break;

case UART\_CONFIG\_WLEN\_5: total\_bit\_amount += 5;

break;

}

total\_bit\_amount +=2;

switch(ui32Config & UART\_CONFIG\_STOP\_MASK)

{

case UART\_CONFIG\_STOP\_ONE: total\_bit\_amount+= 1 ;

break;

case UART\_CONFIG\_STOP\_TWO: total\_bit\_amount += 2 ;

break;

}

return total\_bit\_amount;

}

void SetUartConfig(uint32\_t rate, uint32\_t config)

{

UARTConfigSetExpClk(UART\_BASE, g\_ui32SysClock, rate, config);

}

void GetUartConfig(uint32\_t \*rate, uint32\_t \*config)

{

UARTConfigGetExpClk(UART\_BASE, g\_ui32SysClock, rate, config);

}

void SetControlLineState(unsigned short usState)

{}

void USBUARTIntHandler(void)

{

uint32\_t ui32Ints;

int32\_t i32Errors;

ui32Ints = UARTIntStatus(UART\_BASE, true);

UARTIntClear(UART\_BASE, ui32Ints);

if(ui32Ints & UART\_INT\_TX)

{

USBUARTPrimeTransmit(UART6\_BASE);

}

if(ui32Ints & (UART\_INT\_RX | UART\_INT\_RT)) {

i32Errors = ReadUARTData();

CheckForSerialStateChange(&g\_psCDCDevice, i32Errors);

}

}

void USBUARTPrimeTransmit(uint32\_t ui32Base)

{

uint32\_t ui32Read;

uint8\_t ucChar;

if(g\_bSendingBreak)

{

return;

}

while(UARTSpaceAvail(ui32Base))

{

ui32Read = USBBufferRead((tUSBBuffer \*)&g\_psRxBuffer, &ucChar, 1);

if(ui32Read)

{

UARTCharPut(ui32Base, ucChar);

g\_ui32UARTTxCount++;

}

else

{

return;

}

}

}

int32\_t ReadUARTData(void)

{

int32\_t i32Char, i32Errors;

uint32\_t ui32Space = 0;

i32Errors = 0;

while(ui32Space && UARTCharsAvail(UART\_BASE))

{

i32Char = UARTCharGetNonBlocking(UART\_BASE);

if(!(i32Char & ~0xFF))

{

uint8\_t ucChar = (uint8\_t)(i32Char & 0xFF);

ui32Space--;

}

else

{

i32Errors |= i32Char;

}

g\_ui32UARTRxCount++;

}

return(i32Errors);

}

void SendBreak(bool bSend)

{

if(!bSend)

{

UARTBreakCtl(UART\_BASE, false);

g\_bSendingBreak = false;

}

else

{

UARTBreakCtl(UART\_BASE, true);

g\_bSendingBreak = true;

}

}

void InitializeUART()

{

SysCtlPeripheralEnable(UART\_SYSPERIPH);

SysCtlPeripheralEnable(UART\_GPIO\_PERIPH);

GPIOPinConfigure(UART\_RX\_CHANNEL);

GPIOPinConfigure(UART\_TX\_CHANNEL);

GPIOPinTypeUART(UART\_PORT\_BASE, RX\_GPIO\_PIN | TX\_GPIO\_PIN);

UARTConfigSetExpClk(UART\_BASE, g\_ui32SysClock, DEFAULT\_BIT\_RATE,

DEFAULT\_UART\_CONFIG);

UARTFIFOLevelSet(UART\_BASE, UART\_FIFO\_TX4\_8, UART\_FIFO\_RX4\_8);

uint32\_t config = (UART\_INT\_OE |

UART\_INT\_BE | UART\_INT\_PE |

UART\_INT\_FE | UART\_INT\_RT |

UART\_INT\_TX | UART\_INT\_RX);

UARTIntClear(UART\_BASE, UARTIntStatus(UART\_BASE, false));

UARTIntEnable(UART\_BASE, config);

IntEnable(UART\_INT\_NUM);

}