**HÖHERE TECHNISCHE BUNDESLEHRANSTALT**

**HOLLABRUNN**

Höhere Abteilung für Elektronik – Technische Informatik

|  |  |  |
| --- | --- | --- |
| Klasse / Jahrgang:  5BHEL | Gruppe: | Übungsleiter:  Reisinger |
| Übungsnummer:  3 | Übungstitel: Demoprogramm Messwertvisualisierung – Erzeuger / Verbraucher Problem | |
| Datum der Übung: | Teilnehmer:  Pruggmayer, Mottl | |
| Datum der Abgabe:  25.03.2021 | Schriftführer:  Pruggmayer, Mottl | Unterschrift: |

Inhaltsverzeichnis

1 Aufgabenstellung 3

1.1 Blockschaltbild 3

1.2 Lösungsansatz 3

1.3 ISR (Interrupt Service Routine) 3

1.3.1 Definition 3

1.3.2 Verwendung 3

1.4 Ringbuffer 4

1.4.1 Definition 4

1.4.2 Verwendung 4

1.5 UART-API 4

1.6 decoder-Thread 4

1.7 encoder-Thread 4

1.8 Message-Queue 5

1.8.1 Definition 5

1.8.2 Verwendung 5

1.9 SvVis-API 5

1.10 Cortex – Software 5

1.11 Langsamer Erzeuger(Cortex), schneller Verbraucher(PC) 6

1.12 Schneller Erzeuger(Cortex), langsamer Verbraucher(PC) 7

2 Source Code 8

2.1 main 8

2.2 ring\_pipe 9

2.2.1 Header 9

2.2.2 CPP 10

2.3 serial\_interface 11

2.4 USART 12

2.4.1 Header 12

2.4.2 CPP 13

2.5 SvVis 17

2.5.1 Header 17

2.5.2 CPP 19

2.5.3 SvVis Threads 21

# Aufgabenstellung

Es soll eine **nichtblockierende** Version der SvVis Library implementiert werden, diese soll eine Messwert Virtualisierung für Echtzeitsysteme ermöglichen. Die Visualisierung der Messdaten soll über eine **Acquisition on Event (ACQ\_On, ACQ\_OFF)** ausgelöst werden können. Messdaten werden in einen selbstgeschriebenen Ringbuffer geschrieben, indem sie dann von einem Sende Thread ausgelesen werden. Dieser sendet die Daten, über UART zu einem PC. Ist der PC sehr langsam(niedrige Baudrate) sollen die Messwerte verworfen werden. Es soll auch eine Kommunikation von PC 🡪 Messsystem ermöglicht werden. Diese Implementation soll weiteres auch bei sehr niedrigen Baudraten Funktionen, um z.B. einen langsamen PC zu simulieren.

## Blockschaltbild

## Lösungsansatz

Daten werden im main-Thread(Cortex Software) erzeugt und über eine SvVis-API in eine Message-Queue gelegt. Ein encoder-Thread sendet die Daten an eine UART-API und dieser gibt diese über die seriellen UARTs aus. Werden Daten vom PC gesendet gehen diese über den UART. Sobald Daten ankommen löst der UART eine ISR aus und die Daten werden in einen Ringbuffer gelegt. Der decoder-Thread liest diese Daten über die UART-API ein und legt diese in eine Message-Queue ab. Die Daten werden anschließend von der SvVis-API an die Cortex Software weitergeleitet.

## ISR (Interrupt Service Routine)

### Definition

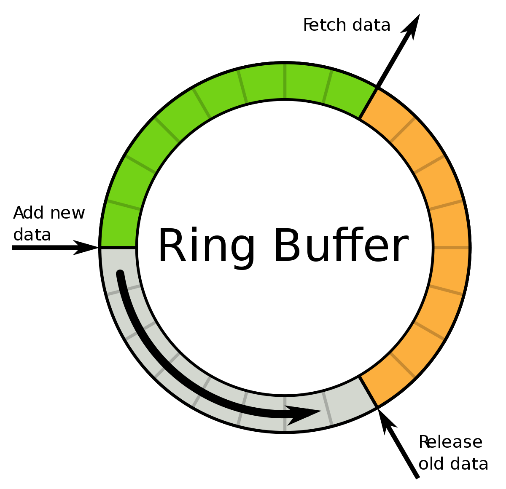
Ein auslösendes Ereignis wird Unterbrechungsanforderung (IRQ) genannt. Nach dieser Anforderung führt der Prozessor eine Unterbrechungsroutine (Interrupt Service Routine ISR) aus.

### Verwendung

Die ISR wird ausgelöst in diesem Fall hardwareseitig durch die UART-Peripherie ausgelöst sobald Daten ankommen.

## Ringbuffer

### Definition

Ein Ringbuffer speichert die Daten kontinuierlich in einem gewissen Zeitraum und überschreibt diese nach dem Ablaufen einer vorgegeben Zeit wieder, um den Speicherplatz für neue Daten wieder freizugeben.

### Verwendung

In unserer Anwendung ist das Release-data direkt nach dem Fetch-data. Darüber hinaus ist ein Range-check eingebaut worden damit man den Ringbuffer nicht überfüllen kann.

## UART-API

Diese API verwaltet das Initialisiere der UARTS, Senden und Empfangen von Daten über die UARTS.

## decoder-Thread

Dieser Thread wandelt den kontinuierlichen Strom an binär-Daten in SvVis Daten Pakete um und gibt diese dann in die Message-Queue von empfangen Daten.

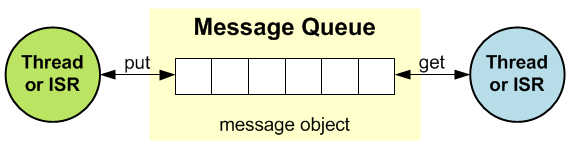
## encoder-Thread

Dieser Thread liest die Message-Queue aus und wandelt kontinuierlich die SvVis Daten Pakete in einen Strom von binär-Daten um und übergibt diese Daten an die UART-API.

## Message-Queue

### Definition

Diese Queue ist eine grundlegende Methode, um Messages zwischen Threads weiterzugeben.

In diesem Model schreibt ein Thread ausschließlich auf diese Queue und ein anderer Thread liest ausschließlich aus. Diese Operation ist eher mehr eine I/O als ein direkter Zugriff auf die Information

### Verwendung

In dieser Anwendung wird die Message-Queue als Zwischenspeicher für zu sendende Pakete als auch für zu empfangende Pakete verwendet.

## SvVis-API

Diese API wurde entwickelt, um SvVis Daten Pakete zu empfangen und zu senden.

## Cortex – Software

Diese Software simuliert Messdaten indem sie ein Dreieck-Signal generiert.

const int16\_t data\_tri[] = {0, 10, 20, 30, 40, 50, 40, 30, 20, 10};

for(size\_t i=0; i<data\_tri\_len; i++)

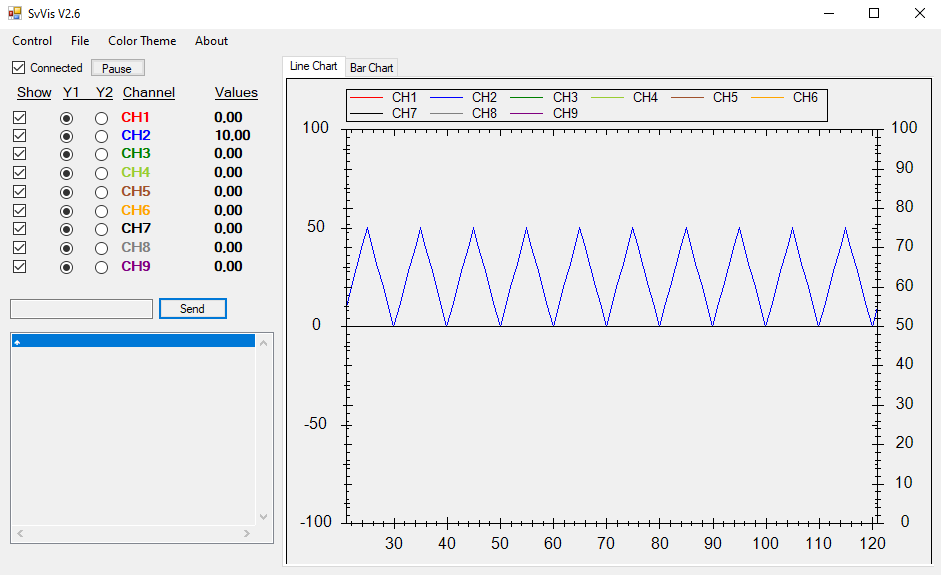
{

svvis.send\_i16(1, data\_tri[i]);

}

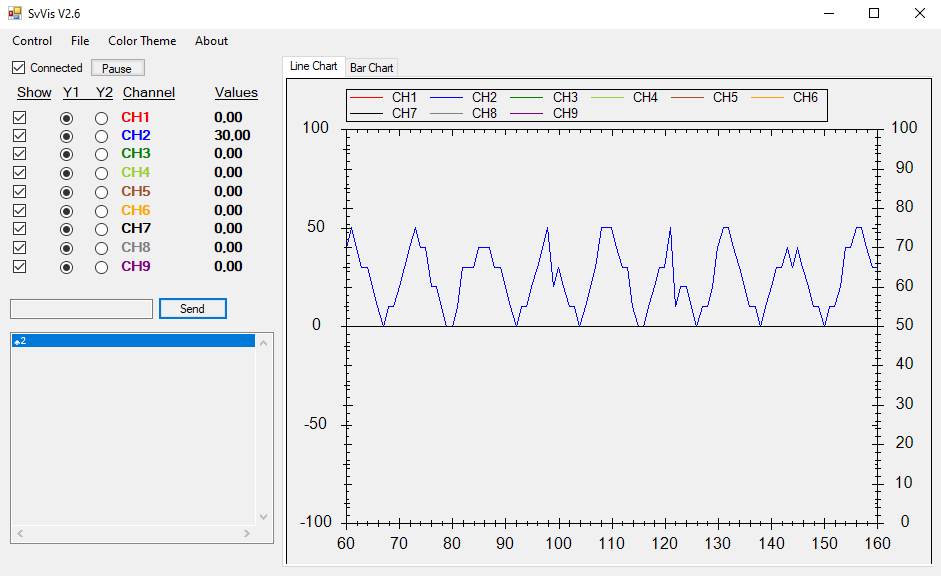
## Langsamer Erzeuger(Cortex), schneller Verbraucher(PC)

Solange der Erzeuger (Cortex) langsamer ist als der Verbraucher(PC) werden keine Daten verworfen. Und man sieht ein schönes Dreieck – Signal. Um dieses zu simulieren wurde zwischen den Ausgaben der Werte eine Wartefunktion aufgerufen.



## Schneller Erzeuger(Cortex), langsamer Verbraucher(PC)

Hier sieht man was passiert wenn der Erzeuger z.B. Cortex schneller Daten produziert als die serielle Schnittstelle übertragen kann. Daraus folgt das daten verworfen werden da der Verbraucher (PC) sehr langsam (niedrige Baudrate) ist. Man kann erkennen das die schönen Dreiecke nicht mehr so schön sind.



# Source Code

## main

#include "USART.hpp"

#include "SvVis.hpp"

usart::usart usart1;

SvVis::SvVis svvis;

SvVis::message\_t msg;

void \_\_NO\_RETURN main\_thread(void \*arg)

{

    for(;;)

    {

        svvis.recv\_msg(msg);

        svvis.send\_msg(msg);

    }

}

const int16\_t data\_tri[] = {0, 10, 20, 30, 40, 50, 40, 30, 20, 10};

const size\_t data\_tri\_len = sizeof(data\_tri) / sizeof(data\_tri[0]);

void \_\_NO\_RETURN send\_tri\_data(void \*arg)

{

    for(;;)

    {

        for(size\_t i=0; i<data\_tri\_len; i++)

        {

            svvis.send\_i16(1, data\_tri[i]);

            osDelay(1000);

        }

    }

}

int main()

{

    osKernelInitialize();

    usart1.init(USART1, 9600);

    svvis.init(usart1);

    osThreadNew(main\_thread, nullptr, nullptr);

    osThreadNew(send\_tri\_data, nullptr, nullptr);

    osKernelStart();

}

## ring\_pipe

### Header

/\*\*

 \* @file ring\_pipe.hpp

 \* @author Pruggmayer Clemens

 \* @brief Definition of a RT-capable byte-wise data pipe

 \* @version 0.1

 \* @date 2021-03-18

 \*

 \* @copyright Copyright (c) 2021

 \*

 \*/

#pragma once

#include <cmsis\_os2.h>

#define nullptr NULL

class ring\_pipe

{

    osMemoryPoolId\_t \_mem;

    osSemaphoreId\_t  \_slots\_data, \_slots\_empty;

    uint8\_t \*\_m\_start, \*\_m\_end,

            \*\_d\_start, \*\_d\_end;

    bool \_enable\_put;

protected:

    uint8\_t &access(uint8\_t \*p);

public:

    osStatus\_t init(size\_t maxsize);

    osStatus\_t enaple\_put(bool newstate);

    osStatus\_t put(uint8\_t data, uint32\_t timeout);

    osStatus\_t pop(uint8\_t &data, uint32\_t timeout);

    bool       is\_empty(void);

};

struct ring\_pipe\_pair

{

    ring\_pipe \*src, \*dst;

};

### CPP

/\*\*

 \* @file ring\_pipe.cpp

 \* @author Pruggmayer Clemens

 \* @brief Implementation of a RT-capable byte-wise data pipe

 \* @version 0.1

 \* @date 2021-03-18

 \*

 \* @copyright Copyright (c) 2021

 \*

 \*/

#include "ring\_pipe.hpp"

uint8\_t &ring\_pipe::access(uint8\_t \*p)

{

    return \_m\_start[ (p - \_m\_start) % (\_m\_end - \_m\_start) ];

}

osStatus\_t ring\_pipe::init(size\_t maxsize)

{

    this->\_slots\_data = osSemaphoreNew(maxsize, 0, NULL);

    this->\_slots\_empty = osSemaphoreNew(maxsize, maxsize, NULL);

    this->\_mem = osMemoryPoolNew(1, maxsize, NULL);

    this->\_m\_start = (uint8\_t\*)osMemoryPoolAlloc(this->\_mem, 0);

    this->\_m\_end = \_m\_start + maxsize;

    this->\_d\_start = this->\_d\_end = this->\_m\_start;

    this->\_enable\_put = true;

    return osOK;

}

osStatus\_t ring\_pipe::enaple\_put(bool newstate)

{

    this->\_enable\_put = newstate;

    return osOK;

}

osStatus\_t ring\_pipe::put(uint8\_t data, uint32\_t timeout)

{

    if(!this->\_enable\_put) {return osErrorParameter;}

    osStatus\_t status;

    // remove 1 empty slot, insert data, and add 1 data slot

    if( (status = osSemaphoreAcquire(this->\_slots\_empty, timeout)) != osOK) {return status;} // return on timeout

    this->access(this->\_d\_end++) = data;

    osSemaphoreRelease(this->\_slots\_data);

    return osOK;

}

osStatus\_t ring\_pipe::pop(uint8\_t &data, uint32\_t timeout)

{

    osStatus\_t status;

    if( (status = osSemaphoreAcquire(this->\_slots\_data, timeout)) != osOK) {return status;} // return on timeout

    data = this->access(this->\_d\_start++);

    osSemaphoreRelease(this->\_slots\_empty);

    return osOK;

}

bool ring\_pipe::is\_empty(void)

{

    return osSemaphoreGetCount(this->\_slots\_data) == 0;

}

## serial\_interface

/\*\*

 \* @file serial\_interface.hpp

 \* @author Pruggmayer Clemens

 \* @brief Definition of a serial Interface for sending/received data

 \* @version 0.1

 \* @date 2021-03-18

 \*

 \* @copyright Copyright (c) 2021

 \*

 \*/

#pragma once

#include "ring\_pipe.hpp"

namespace serial

{

    class interface

    {

    public:

        virtual osStatus\_t pop(uint8\_t &data, uint32\_t timeout) = 0;

        virtual osStatus\_t put(uint8\_t data, uint32\_t timeout) = 0;

        virtual osStatus\_t flush(void) = 0;

        virtual void       put\_blocking(const void \*data, size\_t len) = 0;

    };

} // namespace serial

## USART

### Header

/\*\*

 \* @file USART.hpp

 \* @author Pruggmayer Clemens

 \* @brief Definition for USART Interface implementing the serial::interface

 \* @version 0.1

 \* @date 2021-03-18

 \*

 \* @copyright Copyright (c) 2021

 \*

 \*/

#pragma once

#include "stm32f10x\_usart.h"

#include "stm32f10x\_gpio.h"

#include "stm32f10x\_rcc.h"

#include "misc.h"

#include "serial\_interface.hpp"

#include "ring\_pipe.hpp"

namespace usart

{

    class usart : public serial::interface

    {

    private:

        USART\_TypeDef \*usartn;

        osThreadId\_t send\_thread;

        ring\_pipe \*recv\_pipe;

        bool done\_sending;

    public:

        bool init(USART\_TypeDef \*usartn, uint32\_t baud, size\_t recv\_pipe\_size = 64);

        virtual osStatus\_t pop(uint8\_t &data, uint32\_t timeout);

        virtual osStatus\_t put(uint8\_t data, uint32\_t timeout);

        virtual osStatus\_t flush(void);

        virtual void       put\_blocking(const void \*data, size\_t len);

    };

    namespace usart1

    {

        extern ring\_pipe queue;

        extern ::usart::usart \*handler;

    } // namespace usart1

    namespace usart2

    {

        extern ring\_pipe queue;

        extern ::usart::usart \*handler;

    } // namespace usart2

    namespace usart3

    {

        extern ring\_pipe queue;

        extern ::usart::usart \*handler;

    } // namespace usart3

    namespace flags

    {

        const uint32\_t done\_sending = 0x00000001;

    } // namespace flags

} // namespace usart

### CPP

/\*\*

 \* @file USART.cpp

 \* @author Pruggmayer Clemens

 \* @brief Implementation for USART Interface implementing the serial::interface

 \* @version 0.1

 \* @date 2021-03-18

 \*

 \* @copyright Copyright (c) 2021

 \*

 \*/

#include "USART.hpp"

namespace usart

{

    // global variables from header file

    // becuase inline variables don't work

    // with uvision c++ compiler

    namespace usart1

    {

        ring\_pipe pipe;

        ::usart::usart \*handler = nullptr;

    } // namespace usart1

    namespace usart2

    {

        ring\_pipe pipe;

        ::usart::usart \*handler = nullptr;

    } // namespace usart2

    namespace usart3

    {

        ring\_pipe pipe;

        ::usart::usart \*handler = nullptr;

    } // namespace usart3

} // namespace usart

// ISR for USART1, put received byte into input queue for usart1

extern "C" void USART1\_IRQHandler(void)

{

    USART\_ClearITPendingBit(USART1, USART\_IT\_RXNE);

    char input;

    input = USART\_ReceiveData(USART1);

    usart::usart1::pipe.put(input, 0);

    return;

}

// ISR for USART2, put received byte into input queue for usart2

extern "C" void USART2\_IRQHandler(void)

{

    USART\_ClearITPendingBit(USART2, USART\_IT\_RXNE);

    char input;

    input = USART\_ReceiveData(USART2);

    usart::usart2::pipe.put(input, 0);

    return;

}

// ISR for USART3, put received byte into input queue for usart3

extern "C" void USART3\_IRQHandler(void)

{

    USART\_ClearITPendingBit(USART3, USART\_IT\_RXNE);

    char input;

    input = USART\_ReceiveData(USART3);

    usart::usart3::pipe.put(input, 0);

    return;

}

osStatus\_t usart::usart::pop(uint8\_t &data, uint32\_t timeout)

{

    // get a byte from the input queue, recv\_byte is set in the init function

    return this->recv\_pipe->pop(data, timeout);

}

osStatus\_t usart::usart::put(uint8\_t data, uint32\_t timeout)

{

    while(USART\_GetFlagStatus(this->usartn, USART\_FLAG\_TC) == RESET) {osThreadYield();}

    USART\_SendData(this->usartn, data);

    return osOK;

}

osStatus\_t usart::usart::flush(void)

{

    return osOK;

}

void usart::usart::put\_blocking(const void \*data, size\_t len)

{

    uint8\_t \*buf = (uint8\_t\*)data;

    for(size\_t i=0; i<len; i++)

    {

        this->put(buf[i], osWaitForever);

    }

}

bool usart::usart::init(USART\_TypeDef \*usartn, uint32\_t baud, size\_t recv\_pipe\_size)

{

    GPIO\_InitTypeDef RX, TX;

    USART\_InitTypeDef usart;

    USART\_ClockInitTypeDef usart\_clock;

    NVIC\_InitTypeDef nvic;

    SystemCoreClockUpdate();

    TX.GPIO\_Mode =  GPIO\_Mode\_AF\_PP;

    RX.GPIO\_Mode = GPIO\_Mode\_IN\_FLOATING;

    RX.GPIO\_Speed = TX.GPIO\_Speed = GPIO\_Speed\_50MHz;

    nvic.NVIC\_IRQChannelPreemptionPriority = 0;

    nvic.NVIC\_IRQChannelSubPriority = 0;

    if(usartn == USART1)

    {

        if(::usart::usart1::handler != nullptr) { return false; }

        // init USART1 RX(PA10) and TX(PA9)

        RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA | RCC\_APB2Periph\_AFIO | RCC\_APB2Periph\_USART1, ENABLE);

        RX.GPIO\_Pin = GPIO\_Pin\_10;

        GPIO\_Init(GPIOA, &RX);

        TX.GPIO\_Pin = GPIO\_Pin\_9;

        GPIO\_Init(GPIOA, &TX);

        this->recv\_pipe = &::usart::usart1::pipe; // set recv pipe

        ::usart::usart1::handler = this;    // set handler

        nvic.NVIC\_IRQChannel = USART1\_IRQn;

        nvic.NVIC\_IRQChannelCmd = ENABLE;

    }

    else if(usartn == USART2)

    {

        if(::usart::usart2::handler != nullptr) { return false; }

        // init USART2 RX(PA3) and TX(PA2)

        RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART2, ENABLE);

        RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA | RCC\_APB2Periph\_AFIO, ENABLE);

        RX.GPIO\_Pin = GPIO\_Pin\_3;

        GPIO\_Init(GPIOA, &RX);

        TX.GPIO\_Pin = GPIO\_Pin\_2;

        GPIO\_Init(GPIOA, &TX);

        this->recv\_pipe = &::usart::usart2::pipe; // set recv pipe

        ::usart::usart2::handler = this;    // set handler

        nvic.NVIC\_IRQChannel = USART2\_IRQn;

        nvic.NVIC\_IRQChannelCmd = ENABLE;

    }

    else if(usartn == USART3)

    {

        if(::usart::usart3::handler != nullptr) { return false; }

        // init USART3 RX(PB11) and TX(PB10)

        RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_USART3, ENABLE);

        RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOB | RCC\_APB2Periph\_AFIO, ENABLE);

        RX.GPIO\_Pin = GPIO\_Pin\_11;

        GPIO\_Init(GPIOB, &RX);

        TX.GPIO\_Pin = GPIO\_Pin\_10;

        GPIO\_Init(GPIOB, &TX);

        this->recv\_pipe = &::usart::usart3::pipe; // set recv pipe

        ::usart::usart3::handler = this;    // set handler

        nvic.NVIC\_IRQChannel = USART3\_IRQn;

        nvic.NVIC\_IRQChannelCmd = ENABLE;

    }

    else

    {

        // no usart found, dun't run the code

        return false;

    }

    USART\_DeInit(usartn);

    usart\_clock.USART\_Clock = USART\_Clock\_Disable;

    usart\_clock.USART\_CPOL = USART\_CPOL\_Low;

    usart\_clock.USART\_CPHA = USART\_CPHA\_2Edge;

    usart\_clock.USART\_LastBit = USART\_LastBit\_Disable;

    USART\_ClockInit(usartn, &usart\_clock);

    usart.USART\_BaudRate = baud;

    usart.USART\_WordLength = USART\_WordLength\_8b;

    usart.USART\_StopBits = USART\_StopBits\_1;

    usart.USART\_Parity = USART\_Parity\_No;

    usart.USART\_Mode = USART\_Mode\_Rx | USART\_Mode\_Tx;

    usart.USART\_HardwareFlowControl = USART\_HardwareFlowControl\_None;

    USART\_Init(usartn, &usart);

    NVIC\_Init(&nvic);

    USART\_ITConfig(usartn, USART\_IT\_RXNE, ENABLE);

    this->recv\_pipe->init(recv\_pipe\_size);

    this->usartn = usartn;

    this->done\_sending = true;

    USART\_Cmd(usartn, ENABLE);

    return true;

}

## SvVis

### Header

#pragma once

#include <stdint.h>

#include <string.h>

#include "serial\_interface.hpp"

namespace SvVis

{

    const size\_t data\_max\_len = 32;

    typedef uint8\_t channel\_t;

    namespace channel

    {

        const size\_t             count      = 9;

        const ::SvVis::channel\_t string     = 10;

        const ::SvVis::channel\_t int16\_base = 11;

        const ::SvVis::channel\_t float\_base = 21;

    } // namespace channel

    namespace priority

    {

        const size\_t none = 0;

        const size\_t max = 0xFFFFFFFF;

    } // namespace priority

    namespace flags

    {

        const uint32\_t done\_sending = 0x00000001;

        const uint32\_t n\_flushing   = 0x00000002;

        const uint32\_t aq\_on        = 0x00000004;

    } // namespace flags

    struct message\_t

    {

        ::SvVis::channel\_t channel;

        int8\_t len;

        union data

        {

            int16\_t i16;

            float   f;

            char    raw[::SvVis::data\_max\_len];

        } data;

        bool is\_string();

    };

    class SvVis

    {

    private:

        serial::interface \*interface;

        osMessageQueueId\_t queue\_send, queue\_recv;

        osThreadId\_t thread\_send, thread\_recv;

        osEventFlagsId\_t event\_flags;

        //bool done\_sending, flushing;

        static \_\_NO\_RETURN void func\_send(void \*this\_void);

        static \_\_NO\_RETURN void func\_recv(void \*this\_void);

    public:

        SvVis() : interface(nullptr)/\*, done\_sending(true), flushing(false)\*/ {} // constructor because uvision c++ compiler cant automatically initialise member data

        bool init(serial::interface &interface, size\_t send\_queue\_size = 4, size\_t recv\_queue\_size = 4);

        size\_t     max\_priority();

        osStatus\_t send\_msg(::SvVis::message\_t &msgbuf, uint32\_t timeout = osWaitForever);

        osStatus\_t send\_str(const char \*str,  uint32\_t timeout = osWaitForever);

        osStatus\_t send\_i16(::SvVis::channel\_t channel, int16\_t data, uint32\_t timeout = osWaitForever);

        osStatus\_t send\_float(::SvVis::channel\_t channel, float data, uint32\_t timeout = osWaitForever);

        osStatus\_t flush();

        size\_t available();

        void recv\_msg(::SvVis::message\_t &msgbuf, uint32\_t timeout = osWaitForever);

    };

} // namespace SvVis

### CPP

#include "SvVis.hpp"

bool SvVis::SvVis::init(serial::interface &interface, size\_t send\_queue\_size, size\_t recv\_queue\_size)

{

    if(this->interface != nullptr) { return false; }

    this->interface = &interface;

    this->queue\_send = osMessageQueueNew(send\_queue\_size, sizeof(::SvVis::message\_t), nullptr);

    this->queue\_recv = osMessageQueueNew(recv\_queue\_size, sizeof(::SvVis::message\_t), nullptr);

    this->thread\_send = osThreadNew(::SvVis::SvVis::func\_send, this, nullptr);

    this->thread\_recv = osThreadNew(::SvVis::SvVis::func\_recv, this, nullptr);

    this->event\_flags = osEventFlagsNew(nullptr);

    // setting event flag default

    osEventFlagsSet  (this->event\_flags, ::SvVis::flags::done\_sending); // default: no message, done sending

    osEventFlagsSet  (this->event\_flags, ::SvVis::flags::n\_flushing); // defualt: not flushing

    osEventFlagsClear(this->event\_flags, ::SvVis::flags::aq\_on); // default: aq off

    return true;

}

size\_t SvVis::SvVis::max\_priority()

{

    return osMessageQueueGetCapacity(this->queue\_send);

}

osStatus\_t SvVis::SvVis::send\_msg(::SvVis::message\_t &msgbuf, uint32\_t timeout)

{

    //if(this->flushing == true) { return osErrorResource; }

    if(osEventFlagsWait(this->event\_flags, ::SvVis::flags::n\_flushing, osFlagsWaitAny | osFlagsNoClear, timeout) == (uint32\_t)osErrorTimeout) { return osErrorTimeout; }

    //this->done\_sending = false;

    osEventFlagsClear(this->event\_flags, ::SvVis::flags::done\_sending);

    return osMessageQueuePut(this->queue\_send, &msgbuf, 0, timeout); // put message into queue

}

osStatus\_t SvVis::SvVis::send\_str(const char \*str, uint32\_t timeout)

{

    size\_t len = strlen(str)+1;

    if(len >= ::SvVis::data\_max\_len) {len = ::SvVis::data\_max\_len;}

    ::SvVis::message\_t msg;

    msg.channel = ::SvVis::channel::string;

    msg.len = len;

    memcpy(msg.data.raw, str, len);

    msg.data.raw[::SvVis::data\_max\_len-1] = '\0';

    return this->send\_msg(msg, timeout);

}

osStatus\_t SvVis::SvVis::send\_i16(::SvVis::channel\_t channel, int16\_t data, uint32\_t timeout)

{

    size\_t len = sizeof(data);

    ::SvVis::message\_t msg;

    msg.channel = ::SvVis::channel::int16\_base + channel;

    msg.len = len;

    msg.data.i16 = data;

    return this->send\_msg(msg, timeout);

}

osStatus\_t SvVis::SvVis::send\_float(::SvVis::channel\_t channel, float data, uint32\_t timeout)

{

    size\_t len = sizeof(data);

    ::SvVis::message\_t msg;

    msg.channel = ::SvVis::channel::float\_base + channel;

    msg.len = len;

    msg.data.f = data;

    return this->send\_msg(msg, timeout);

}

osStatus\_t SvVis::SvVis::flush()

{

    //this->flushing = true;

    osEventFlagsClear(this->event\_flags, ::SvVis::flags::n\_flushing);

    osEventFlagsWait(this->event\_flags, ::SvVis::flags::done\_sending, osFlagsWaitAny, osWaitForever); // wait for send queue to be empty

    //while ( this->done\_sending  == false ) { osThreadYield(); }

    this->interface->flush(); // wait for interface to be empty

    //this->flushing = false;

    osEventFlagsSet(this->event\_flags, ::SvVis::flags::n\_flushing);

    return osOK;

}

size\_t SvVis::SvVis::available()

{

    return osMessageQueueGetCount(this->queue\_recv);

}

void SvVis::SvVis::recv\_msg(::SvVis::message\_t &msgbuf, uint32\_t timeout)

{

    osMessageQueueGet(this->queue\_recv, &msgbuf, nullptr, timeout);

}

### SvVis Threads

#include "SvVis.hpp"

namespace SvVis

{

    size\_t chid2len(uint8\_t chid)

    {

        if(chid == ::SvVis::channel::string) return ::SvVis::data\_max\_len;

        if((chid >= ::SvVis::channel::float\_base) && (chid < ::SvVis::channel::float\_base + ::SvVis::channel::count)) return sizeof(float);

        if((chid >= ::SvVis::channel::int16\_base) && (chid < ::SvVis::channel::int16\_base + ::SvVis::channel::count)) return sizeof(int16\_t);

        return 0;

    }

} // namespace SvVis

\_\_NO\_RETURN void SvVis::SvVis::func\_recv(void \*this\_void)

{

    ::SvVis::SvVis \*tar = (::SvVis::SvVis\*)this\_void;

    ::SvVis::message\_t msg;

    uint8\_t recvbuf;

    size\_t maxlen;

    for(;;)

    {

        // init length

        msg.len = 0;

        tar->interface->pop(recvbuf, osWaitForever);

        msg.channel = recvbuf;

        maxlen = ::SvVis::chid2len(msg.channel);

        memset(&msg.data, 0, sizeof(msg.data));

        if(msg.channel != ::SvVis::channel::string)

        {

            // handle non-string messages

            while (msg.len < maxlen)

            {

                tar->interface->pop(recvbuf, osWaitForever);

                if(msg.len < ::SvVis::data\_max\_len) {msg.data.raw[msg.len++] = recvbuf;}

            }

            osMessageQueuePut(tar->queue\_recv, &msg, 0, osWaitForever);

        }

        else

        {

            // handle string message

            while (recvbuf != '\0')

            {

                tar->interface->pop(recvbuf, osWaitForever); // the terminating '\0' will be put into the buffer

                if(msg.len < ::SvVis::data\_max\_len-1) {msg.data.raw[msg.len++] = recvbuf;}

            }

            msg.data.raw[::SvVis::data\_max\_len-1] = '\0'; // security cut at the end of the string

            if(msg.data.i16 == 0)

            {   // aq off

                osEventFlagsClear(tar->event\_flags, ::SvVis::flags::aq\_on);

            }

            else if(msg.data.i16 == 1)

            {   // aq on

                osEventFlagsSet(tar->event\_flags, ::SvVis::flags::aq\_on);

            }

            else

            {   // string message

                osMessageQueuePut(tar->queue\_recv, &msg, 0, osWaitForever);

            }

        }

    }

}

\_\_NO\_RETURN void SvVis::SvVis::func\_send(void \*this\_void)

{

    ::SvVis::SvVis \*tar = (::SvVis::SvVis\*)this\_void;

    ::SvVis::message\_t msg;

    for(;;)

    {

        osMessageQueueGet(tar->queue\_send, &msg, nullptr, osWaitForever);

        osEventFlagsWait(tar->event\_flags, ::SvVis::flags::aq\_on, osFlagsWaitAny | osFlagsNoClear, osWaitForever);

        tar->interface->put(msg.channel, osWaitForever);

        tar->interface->put\_blocking(msg.data.raw, msg.len);

        if(osMessageQueueGetCount(tar->queue\_send) == 0) { osEventFlagsSet(tar->event\_flags, ::SvVis::flags::done\_sending); /\*tar->done\_sending = true;\*/ }

    }

}