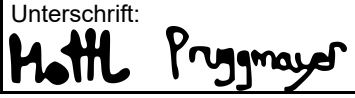
HÖHERE TECHNISCHE BUNDESLEHRANSTALT

HOLLABRUNN

Höhere Abteilung für Elektronik – Technische Informatik

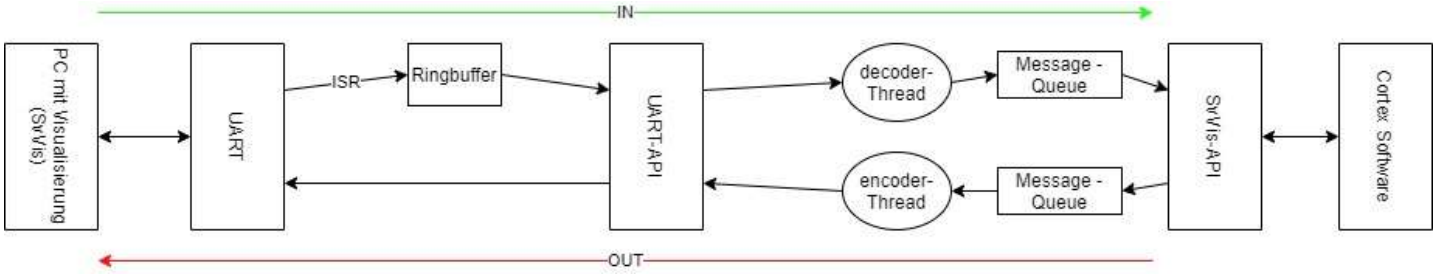
Klasse / Jahrgang: 5BHEL	Gruppe:	Übungsleiter: Reisinger
Übungsnummer: 3	Übungstitel: Demoprogramm Messwertvisualisierung – Erzeuger / Verbraucher Problem	
Datum der Übung:	Teilnehmer: Pruggmayer, Mottl	
Datum der Abgabe: 25.03.2021	Schriftführer: Pruggmayer, Mottl	Unterschrift: 

Inhaltsverzeichnis

1 Aufgabenstellung

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.

1.1 Blockschaltbild



1.2 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.

1.3 ISR (Interrupt Service Routine)

1.3.1 Definition

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

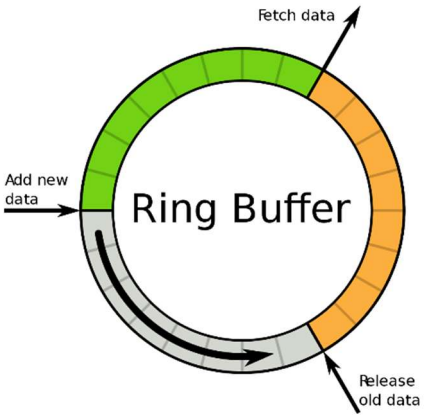
1.3.2 Verwendung

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

1.4 Ringbuffer

1.4.1 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.



1.4.2 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.

1.5 UART-API

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

1.6 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.

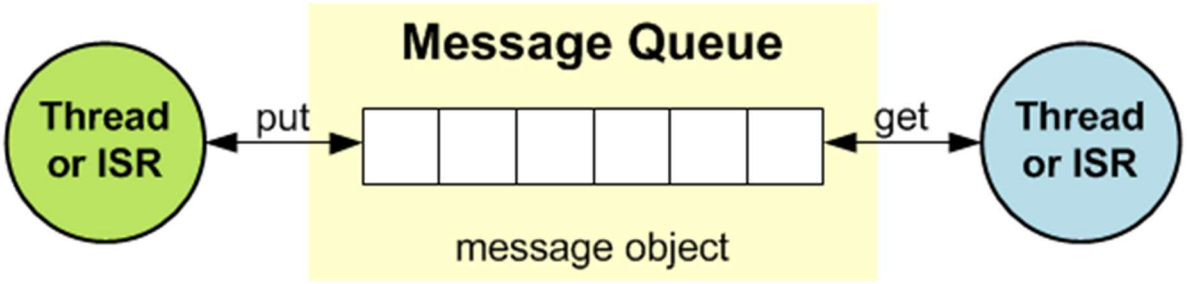
1.7 encoder-Thread

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.

1.8 Message-Queue

1.8.1 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



1.8.2 Verwendung

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

1.9 SvVis-API

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

1.10 Cortex – Software

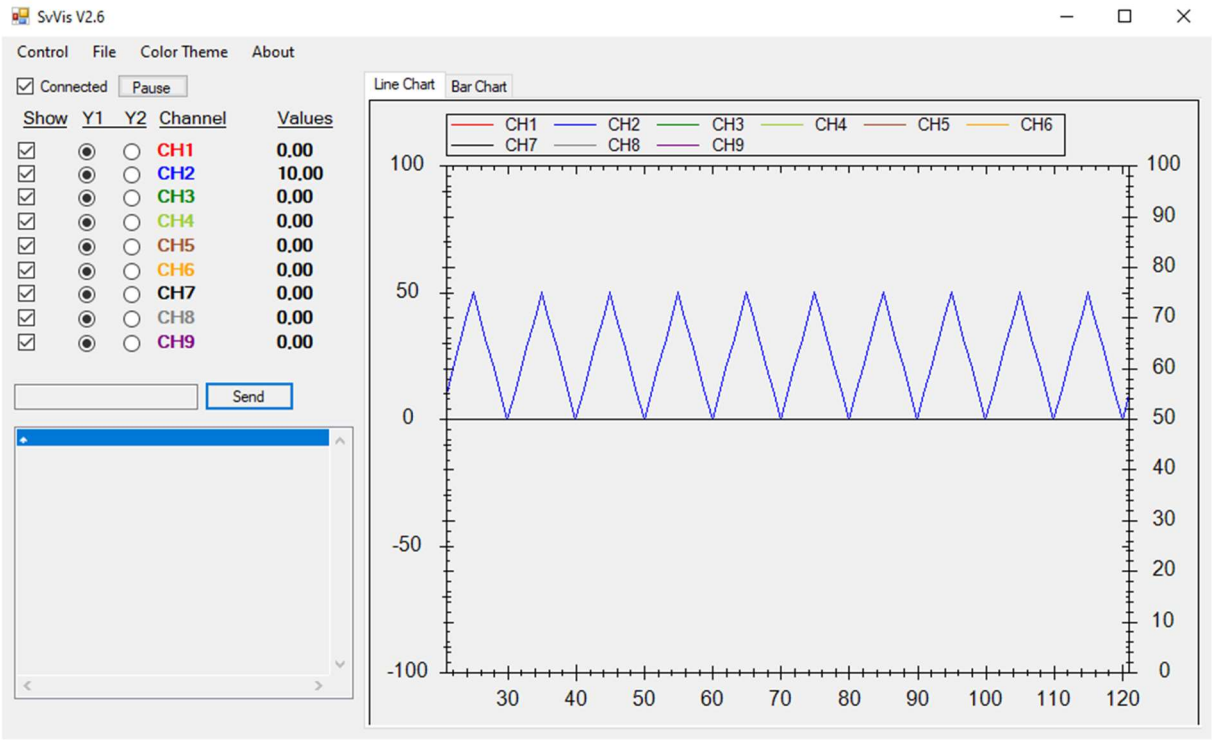
Diese Software simuliert Messdaten indem sie ein Dreieck-Signal generiert.

```
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]);
}
```

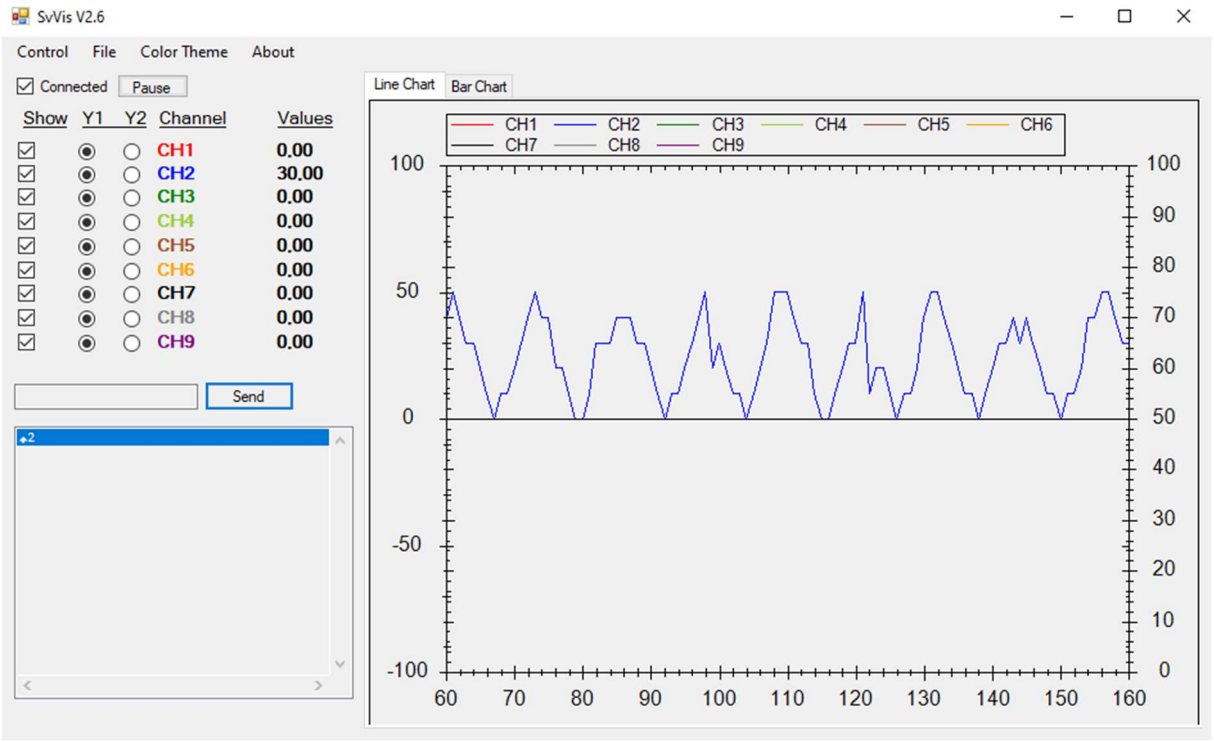
1.11 **Langsamer Erzeuger(Cortex), schneller Verbraucher(PC)**

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.



1.12 **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.



2 Source Code

2.1 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();
}
```

2.2 ring_pipe

2.2.1 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;
};
```

2.2.2 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;
}
```

2.3 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
```

2.4 USART

2.4.1 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
```

2.4.2 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_siz
e)
{
    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_AP
B2Periph_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;
}
```

2.5 SvVis

2.5.1 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 da
ta
        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 = osWaitF
orever);
        osStatus_t send_str(const char *str,  uint32_t timeout = osWaitForever);
        osStatus_t send_i16(::SvVis::channel_t channel, int16_t data, uint32_t tim
eout = osWaitForever);
        osStatus_t send_float(::SvVis::channel_t channel, float data, uint32_t tim
eout = osWaitForever);
        osStatus_t flush();
        size_t available();
        void recv_msg(::SvVis::message_t &msgbuf, uint32_t timeout = osWaitForever
);
    };
```

```
} // namespace SvVis
```

2.5.2 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), nullp
tr);
    this-
>queue_recv = osMessageQueueNew(recv_queue_size, sizeof(::SvVis::message_t), nullp
tr);
    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); // wai
t 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);
}
```

2.5.3 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::flo
at_base + ::SvVis::channel::count)) return sizeof(float);
        if((chid >= ::SvVis::channel::int16_base) && (chid < ::SvVis::channel::int
16_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++] = rec
vbuf;}
            }
```

```
            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, osWaitForeve
r);
        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;*/ }
    }
}
```