

# **Decoding the S-Bus protocol with a Grasshopper**

**J. Lemaire**



# 1 Objectives

It is proposed here to describe the S-Bus protocol and to test it with a FrSky R-XSR receiver. The transmitted frames will be decoded first with an oscilloscope. This decoding will then be confirmed using a serial monitor. All this in view to write decoding programs on a Grasshopper development board microcontroller from Tlera Corp.

## 2 The S-BUS protocol<sup>1</sup>

S-Bus is a protocol invented by Futaba for communication between a radio receiver and servos. But it is now very often used for communications between a radio receiver and a flight controller.

It is an inverted (1  $\Leftrightarrow$  0) TTL serial protocol<sup>2</sup> that operates at 100000 bauds<sup>3</sup> and can transmit 18 channels on a single wire :

- 16 proportional channels, with 2048 possible levels (11-bit coding) ;
- 2 binary channels.

A **S-Bus frame** is a sequence of 25 bytes (200 bits in total then) as follows:

Start = 0x0F	CH1	CH2	...	CH16	Flags	End= 0x00
8 bits	11 bits	11 bits	...	11 bits	8 bits	8 bits

<sup>1</sup> <https://os.mbed.com/users/Digixx/notebook/futaba-s-bus-controlled-by-mbed/> for instance.

<sup>2</sup> The **TTL serial protocol** allows bidirectional and asynchronous digital communication between 2 sources. In addition to the GND (0 V) and VCC (typically +3.3 V or +5 V) reference inputs, each must have at least one RX input to receive data and one TX output to output. They will be connected by crossing them. We can also have pins to control the streams (CTS and RTS), but they will not be used here.

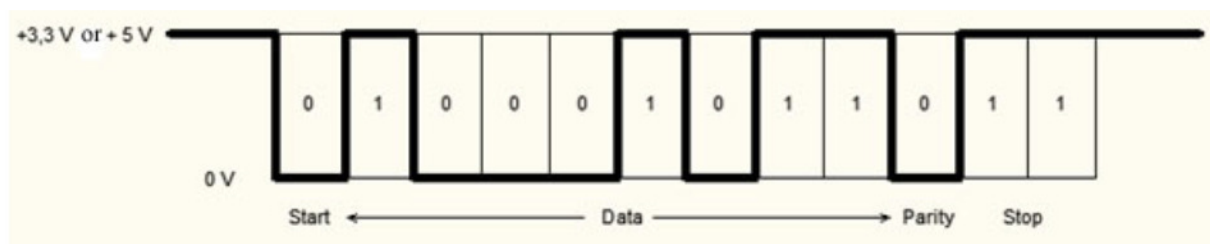
The logical levels are defined as follows:

- 1  $\Leftrightarrow$  HIGH (VCC V).
- 0  $\Leftrightarrow$  LOW (0 V).

At rest, the line is HIGH.

The data frames are composed of :

- a Start bit 0 which can detect the beginning of a frame ;
- the data bits (8 here) ;
- possibly a parity bit to detect an error: for example, in even parity (here), we make sure that the total number of data bits or parity equal to 1 is an even number ;
- 1 to 2 (here) Stop bits 1 to indicate a minimum delay before the start of a new frame.



The data bits provide the inverted binary representation (least significant bits  $\Leftrightarrow$  most significant bits) of an unsigned integer: for example, the previous sequence (10001011) corresponds to the integer 0b11010001 = 209.

The bits are transmitted with a constant speed defined in bauds (N bits / s).

<sup>3</sup> 100000 bits/s  $\Leftrightarrow$  10  $\mu$ s/bit

Flags :

CH17	CH18	Lost frame	Failsafe	Unused
1 bit	1 bit	1 bit	1 bit	4 bits

Thus<sup>4</sup> :

- CH1 uses the 8 bits of the 2nd byte (least significant) and 3 first bits of the 3rd byte (most significant) ;
- CH2 uses the last 5 bits of the 3rd byte (least significant) and the first 6 bits of the 4th byte (most significant) ;
- etc.

Finally, each byte of this S-Bus frame is transmitted as follows using a 12-bits inverted serial frame :

Start bit = 1	Data	Parity bit (odd) <sup>5</sup>	Stop bits = 00
1 bit	8 bits	1 bit	2 bits

Thus, to transmit a 25-bytes S-bus frame, it will be necessary to send 300 bits on the serial port, which will therefore take 3 ms at 100000 bits / s. The sequence of these bits will be called the **S-Bus serie frame** in the following.

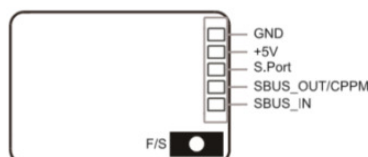
In practice, a S-bus serie frame is sent periodically every T ms with  $7 \leq T \leq 14$  ; this corresponds to a frequency near 100 Hz.

Finally, note that this protocol allows to connect several servos (slaves) on the same bus with a single receiver (master). But this aspect of things will not be treated here where we will limit ourselves to a couple (master, slave).

### 3 FrSky R-XSR receiver<sup>6</sup>



It allows to receive 16 channels and to transmit them on its output S-BUS\_OUT / CPPM in a S-BUS (by default) or CPPM format :



In this test, the R-XSR will be paired with a FrSky Taranis X9D+ radio that transmits values for 16 channels with CH1 = Aileron, CH2 = Elevator, CH3 = Throttle and CH4 = Rudder.

<sup>4</sup> Cf. § 4.1 fore more precisions.

<sup>5</sup> After the inversion, the parity will then be « even ».

<sup>6</sup> <https://www.frsky-rc.com/product/r-xsr/>

**Failsafe** setting :

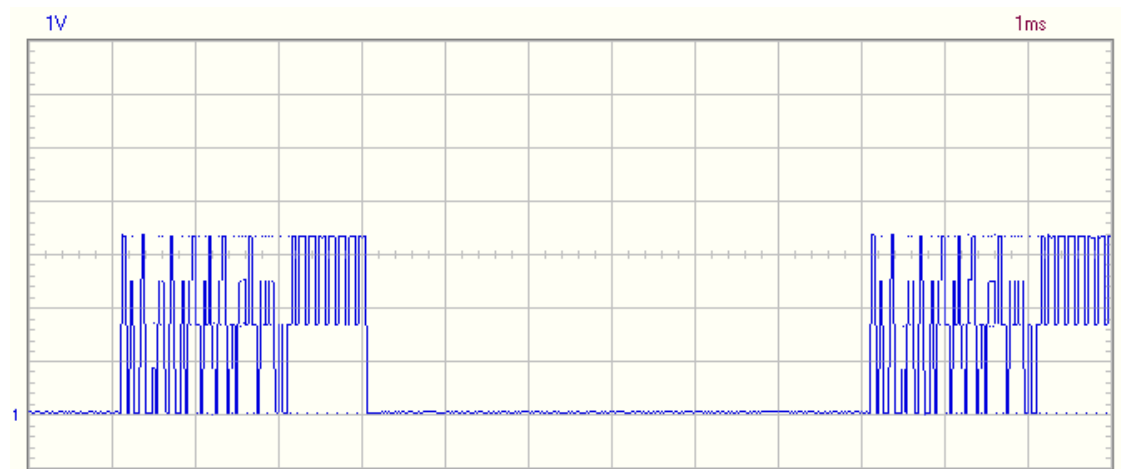
- on the radio, set to "Receiver" in the selected model configuration page ;
- the receiver transmits on the SBUS the values corresponding to CH3 = -100% (throttle min) and all other channels equal to 0%, the flag Failsafe being then equal to 1.

Note also that this radio receiver can be powered by a Lipo 1S battery (3.7V).

## 4 S-Bus frames decoding

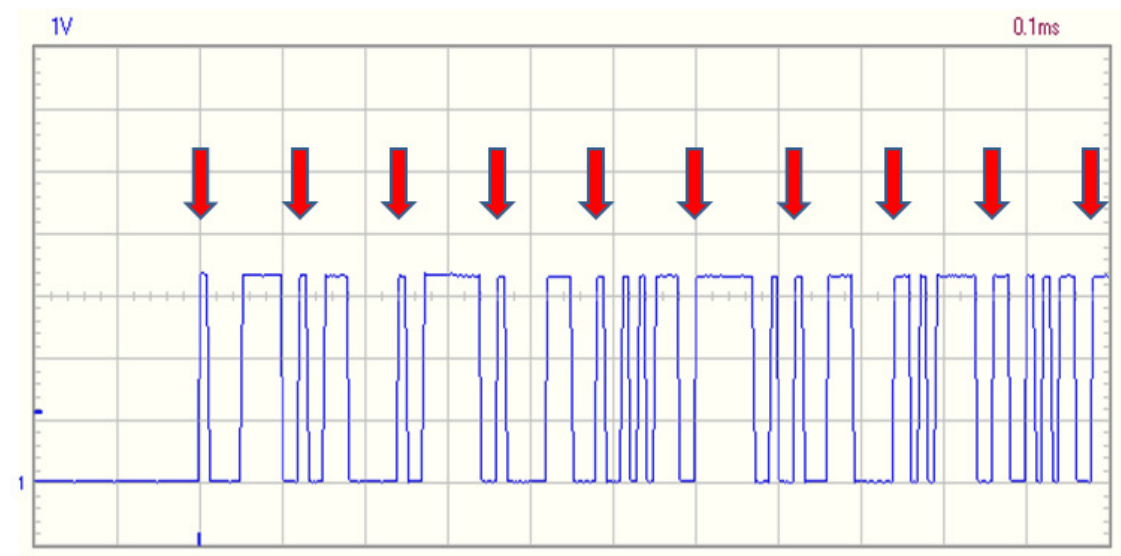
### 4.1 Decoding with an oscilloscope<sup>7</sup>

With **PCLab2000LT**, the R-XSR S-Bus output can be plotted :



The signal level is between 0 V (= LOW) and 3.3 V (= HIGH), each S-Bus serie frame lasts 3 ms and such a frame is sent every 9 ms.

Transmitted data with CH1  $\cong$  0%, CH2  $\cong$  0%, CH3  $\cong$  -100% (min throttle), CH4  $\cong$  0%, CH5  $\cong$  -100%, etc. :



<sup>7</sup> <https://www.vellemanusa.com/products/view/?country=us&lang=enu&id=524708>

S-Bus serie frame (data bits in yellow) :

10000111110010011110000000100111111100100000111000100101011001111111001001000111000000110101111100...

Inverted S-Bus serie frame (1  $\Leftrightarrow$  0) :

011110000011101100011110110000000111011111000111011010100111000000110110110001111001010000011...

Decoded bytes (least significant bits and most significant bits are reversed) :

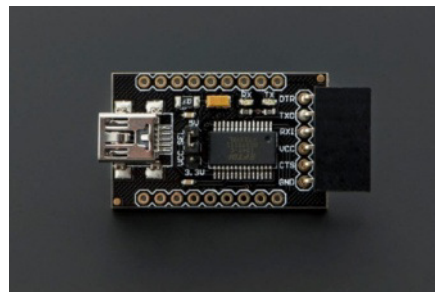
Byte	D0	D1	D2	D3	D4	D5	D6	D7	...
Binary	00001111	11100011	00000011	00011111	00101011	11000000	11000111	00001010	...
Hexa	0F	E3	03	1F	2B	C0	C7	0A	...

Corresponding S-Bus frame :

- Début = D0 = 0x0F
- CH1 = (D1 | D2<<8) & 0x07FFF = 0b011111100011 = 995
- CH2 = (D2>>3 | D3<<5) & 0x07FFF = 0b011111100000 = 992
- CH3 = (D3>>6 | D4<<2 | D5<<10) & 0x07FFF = 0b000101011100 = 172
- CH4 = (D5>>1 | D6<<7) & 0x07FFF = 0b011111100000 = 992
- CH5 = (D6>>4 | D7<<4) & 0x07FFF = 0b000101011100 = 172
- ...

## 4.2 Decoding with a serial monitor on a PC

Here, the **Serial TTL  $\Leftrightarrow$  USB** converter from **DFRobot**<sup>8</sup> is used :



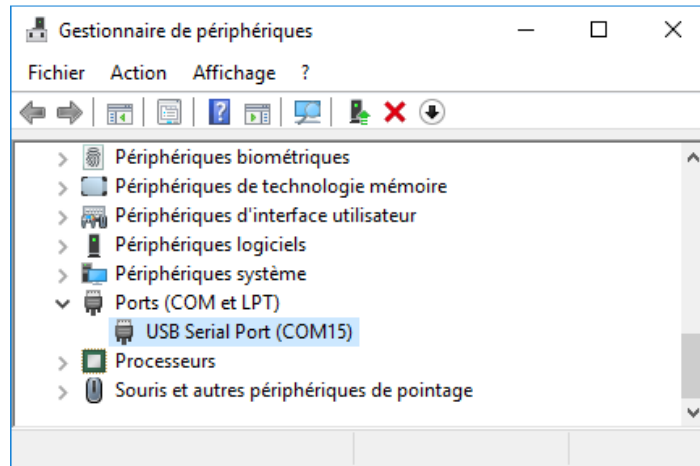
Connections with the R-XSR :

R-XSR	Converter <sup>9</sup>
S-BUS	TXO (=RX)
+5V	VCC
GND	GND

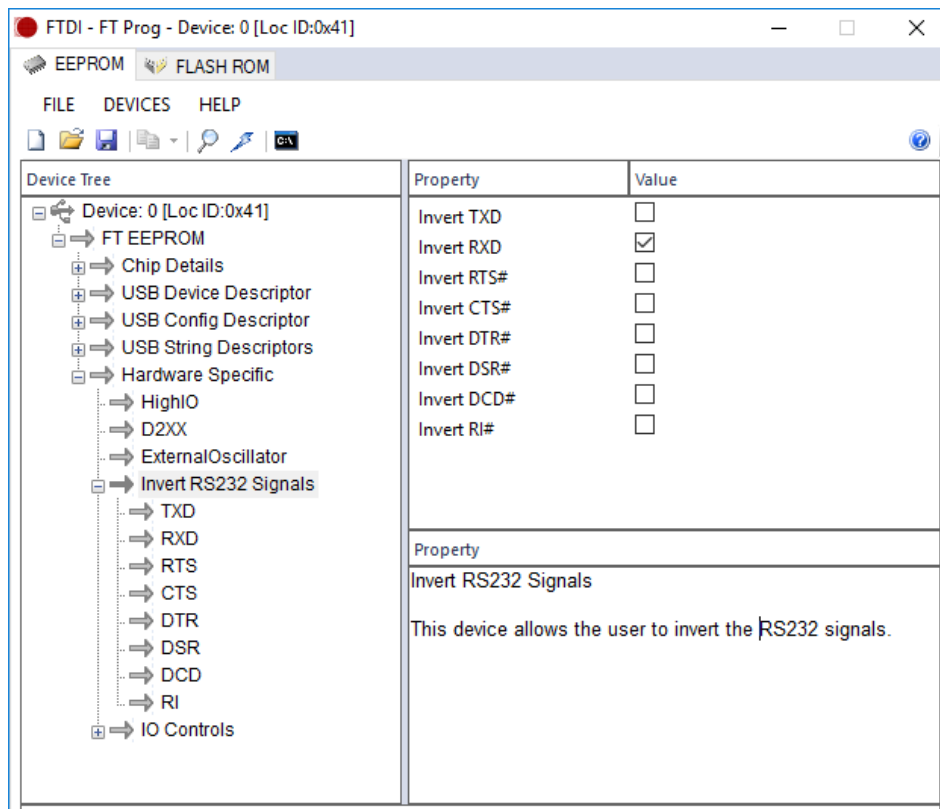
As soon as it is connected to the PC via its USB port, the necessary FTDI drivers are automatically installed and a virtual serial port is added : COMx, whose number (15 here) should be noted with the command **Parameters / Devices / Device Manager / Ports (COM and LPT)** :

<sup>8</sup> [http://www.dfrobot.com/index.php?route=product/product&filter\\_name=dfrobot065&product\\_id=147#.URDbMR38KrE](http://www.dfrobot.com/index.php?route=product/product&filter_name=dfrobot065&product_id=147#.URDbMR38KrE)

<sup>9</sup> On this converter the indications of pins RXI and TXO mention the name of the pins to which it must be connected on a microcontroller.



This converter was chosen because it has a genuine programmable FTDI component<sup>10</sup>. This feature is important here because it will allow to invert the bits (1↔0) of the received signal, before its decoding. This can be done with the program **FT\_Prog**<sup>11</sup>:



The procedure is as follows :

- start **FT\_Prog** and identify the FTDI component with the **DEVICES / Scan and Parse** command ;
- check the **Invert RXD** box in the **FT EEPROM / Hardware Specific / Invert RS232 Signals** parameter page ;
- save this change in the component with the **DEVICES / Program** command.

Then, on the PC, it is necessary to use a serial monitor, **RealTerm**<sup>12</sup> for example, connected to the virtual serial port (COM15 here thus) created by the USB connection, with the following parameters :

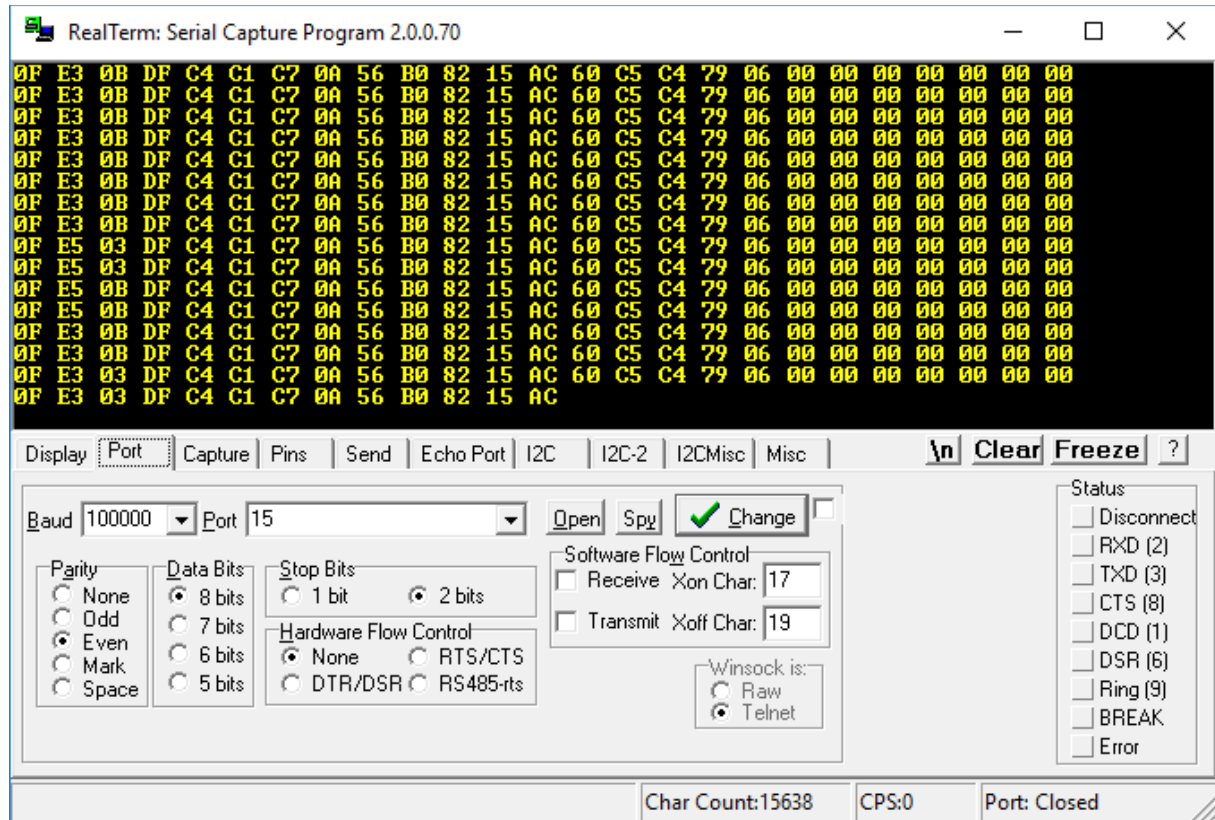
<sup>10</sup> This is not always the case with other converters using FTDI clones...

<sup>11</sup> [http://www.ftdichip.com/Support/Utilities.htm#FT\\_PROG](http://www.ftdichip.com/Support/Utilities.htm#FT_PROG)





Decoded bytes after changing CH3  $\cong$  100% :



Decoded S-Bus frame sample :

Byted	D0	D1	D2	D3	D4	D5	D6	D7	...
Hexa	0F	E3	0B	DF	C4	C1	C7	0A	...
Binary	00001111	11100011	00001011	11011111	11000100	11000001	11000111	00001010	...

- Start = D0 = 0x0F
- CH1 = (D1 | D2<<8) & 0x07FFF = 0b011111100011 = 995
- CH2 = (D2>>3 | D3<<5) & 0x07FFF = 0b011111100001 = 993
- CH3 = (D3>>6 | D4<<2 | D5<<10) & 0x07FFF = 0b11100010011 = **1811**
- CH4 = (D5>>1 | D6<<7) & 0x07FFF = 0b011111100000 = 992
- CH5 = (D6>>4 | D7<<4) & 0x07FFF = 0b00010101100 = 172
- ...

Remarks

- It is observed here that the values -100% and + 100% on a channel of the radio produce the values 172 and 1811 on the SBUS, the value 0% producing a value close to 992. This leads Simon Levy to propose the following normalization :

$$\widehat{SBUS} = \frac{SBUS - 172}{1811 - 172} \times 2 - 1 \cong SBUS \times 0,00122 - 1,209884$$

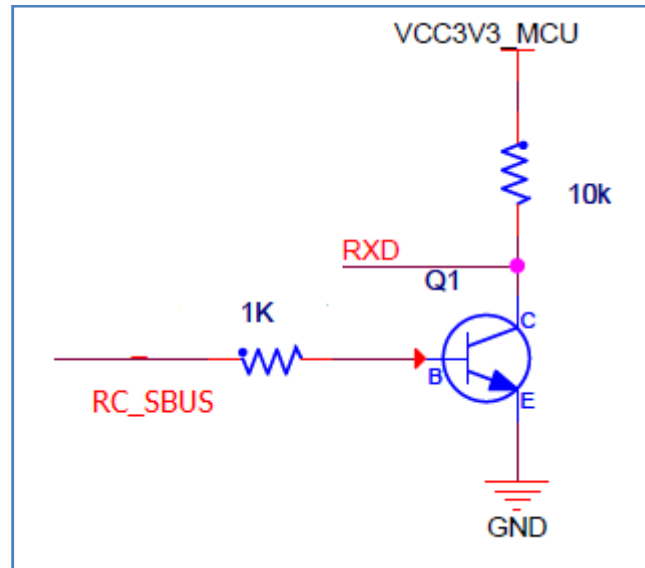
to obtain values between -1 and +1<sup>13</sup>.

Nevertheless, it should be noted that these extreme values can change if the trims of the radio are modified. In this situation, it would be better to calibrate the radio channels as

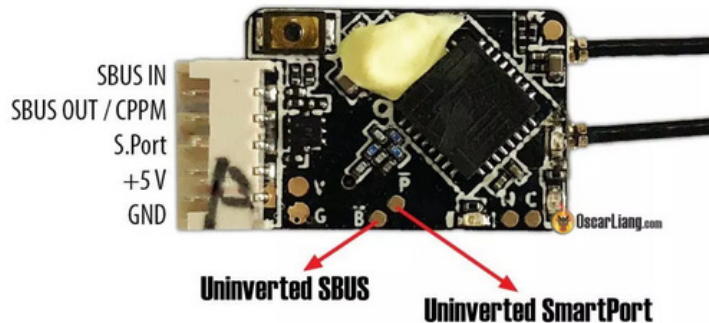
<sup>13</sup> <https://github.com/simondlevy/SBUSRX>

proposed in the ground station softwares (Mission Planner<sup>14</sup> for example), to determine the extreme values produced on the S-Bus.

- It is essential here to be able to set the speed of the port equal to 100000 bauds, which is not traditional ! On the other hand, if input signal bits cannot be inverted ( $1 \Leftrightarrow 0$ ) by programming the FTDI component, there are hardware solutions with a NPN transistor (S9014-TO92 for example) used in switching mode<sup>15</sup>



But on the R-XSR, we can also connect directly to a pin that does not reverse the signal<sup>16</sup> :



## 4.3 Decoding with a Grasshopper

### 4.3.1 The Grasshopper board from Tlera Corp<sup>17</sup>

Designed by Kris Winer and his team, this development board is based on Murata's **CMWX1ZZABZ-078** component<sup>18</sup>, which brings together in a single chip:

- a STMicroelectronics **STM32L082KZ**<sup>19</sup> microcontroller ;
- a Semtech **SX1276**<sup>20</sup> transceiver ;

<sup>14</sup> <http://ardupilot.org/planner/>

<sup>15</sup> [https://dev.px4.io/en/tutorials/linux\\_sbus.html](https://dev.px4.io/en/tutorials/linux_sbus.html)

<sup>16</sup> <https://oscarliang.com/uninverted-sbus-smart-port-frsky-receivers/>

<sup>17</sup> <https://www.tindie.com/products/TleraCorp/grasshopper-loralorawan-development-board/>

<sup>18</sup> <https://wireless.murata.com/eng/products/rf-modules-1/lpwa/type-abz.html>

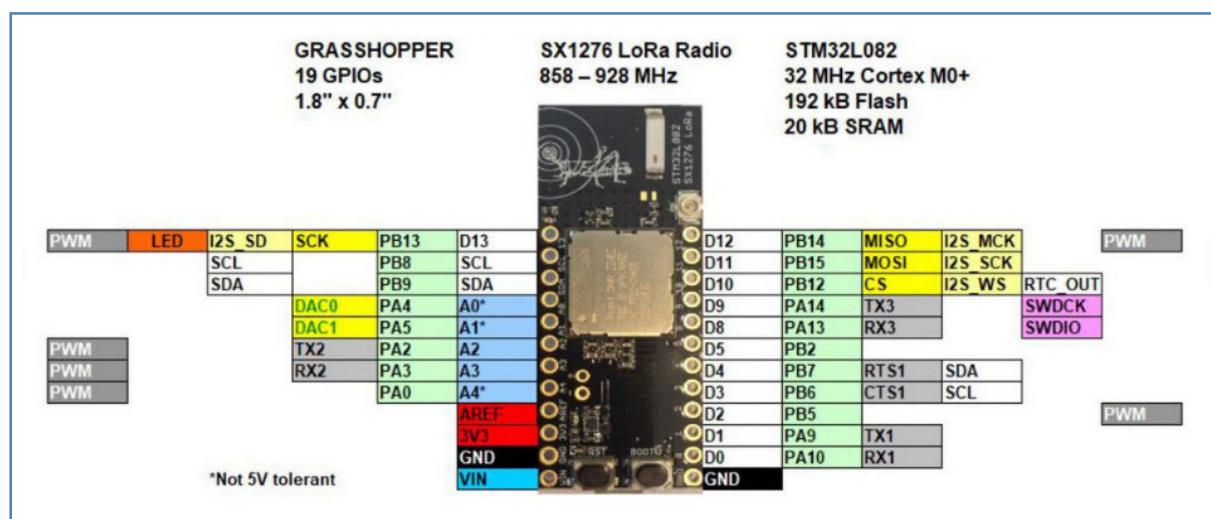
<sup>19</sup> [https://www.st.com/content/st\\_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-ultra-low-power-mcus/stm32l0-series/stm32l0x2/stm32l082kz.html](https://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-ultra-low-power-mcus/stm32l0-series/stm32l0x2/stm32l082kz.html)

whose essential characteristics are :

- 32 bits ARM Cortex M0 CPU ;
- clocked at 32MHz ;
- 192KB flash memory and 20KB RAM ;
- very low consumption ;
- 3.3V logic level with any 5V tolerant pins ;
- 3x 12-bits ADC ;
- 19 affectable GPIOs (USB, Serial, I2C, SPI, analog comparators...) ;
- LPWA 860-930MHz module ;
- internal antenna tuned to this frequency band + IPX connector for external antenna ;
- FSK, OOK and LoRa modulation.

With the format of a Teensy 3.2 (excluding internal antenna), this card has an USB connector, a 3.3V-150mA On Semiconductor **NCV8170**<sup>21</sup> regulator allowing a 3.7V / 5V power supply via the VIN pin, the plug in battery or the USB connector (possible simultaneously) and a 3.3V output.

Standard pin affectation :



Thanks to Thomas Roell's basic Arduino library for STM32L0<sup>22</sup> and an integrated bootloader<sup>23</sup>, the Grasshopper can be programmed easily with the Arduino IDE and via its USB connector. Several examples of programs are provided<sup>24</sup> and Kris Winer responds very quickly by email to any technical question !

### 4.3.2 The Grasshopper serial ports

In addition to the serial port associated with its USB port, the Grasshopper card has 3 other serial ports that use by default the following pins :

- RX1 => D0 and TX1 => D1
- RX2 => A3 and TX2 => A2

<sup>20</sup> <https://www.semtech.com/products/wireless-rf/lor-transceivers/sx1276>

<sup>21</sup> <https://www.onsemi.com/PowerSolutions/product.do?id=NCV8170>

<sup>22</sup> <https://github.com/GrumpyOldPizza/ArduinoCore-stm32l0>

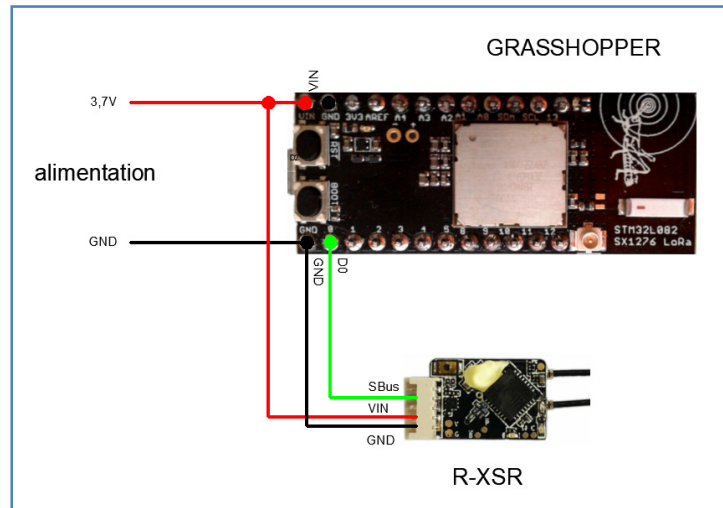
<sup>23</sup> Sometimes a faulty program may prevent the integration of the standard serial USB interface into the Arduino IDE. In this case, connect the Grasshopper and press the RESET button (left) while holding down the BOOT button (right), then download a correct program (Blink for example) so that the USB serial port is recreated.

<sup>24</sup> <https://github.com/kriswiner/CMWX1ZZABZ/tree/master/Grasshopper>

- RX3 => D8 and TX3 => D9

They can be accessed via the **Serial1**, **Serial2**, **Serial3** objects.

Here we will use **Serial1**, the pin **RX1** (D0) being connected to the output **S-BUS\_OUT / CPPM** of the receiver R-XSR :



The port parameters will be defined using the **begin (baud, format)** method. For example, to choose 100000 baud, to invert the bits of the signal (0 ⇔ 1) and to have 8 bits of data with an even parity bit<sup>25</sup> as well as 2 Stop bits, one will call the method :

**begin (100000, SERIAL\_SBUS)**

the constant **SERIAL\_SBUS** being defined in the **HardwareSerial.h** file of the Arduino Core library by:

```
#define SERIAL_SBUS (STM32L0_UART_OPTION_DATA_SIZE_8 | STM32L0_UART_OPTION_PARITY_EVEN |  
STM32L0_UART_OPTION_STOP_2 | STM32L0_UART_OPTION_RX_INVERT | STM32L0_UART_OPTION_TX_INVERT)
```

Therefore, a FIFO buffer with 64 bytes length<sup>26</sup> is automatically filled with the decoded bytes in the series S-Bus frames; we test the presence of available data with the **available()** method and get them while emptying the buffer with the **read()** method. The **end()** method stops decoding. To decode the S-Bus frames, it is therefore sufficient to regularly look at this buffer.

To decode this S-Bus data, we are led to define 2 procedures :

- **parseSBusFrame** that will fill an S-Bus frame by exploiting the previous FIFO buffer as often as possible. The simplest way is to do this by implementing **serialEvent1** :

```
void serialEvent1(void)
{
  ...
}
```

because it will be called automatically at the end of the program loop if a serial event has occurred (here receiving a character on RX1), as can be seen in the **main.cpp** program of the Arduino Core library :

```
for (;;)
{
  loop();
```

<sup>25</sup> After inversion of the bits.

<sup>26</sup> Defined in **HardwareSerial.h** de the Arduino Core.

```

    if (g_serialEventRun) (*g_serialEventRun)();
}

```

Contrary to what its name might suggest, it is not an event procedure run in interrupt mode. But if we take care not to introduce waiting times in the **loop()** procedure of the **.ino** program, we will be able to process efficiently and very simply (avoiding the pitfalls of interrupt processing) the flow of data arriving on the serial port (1 byte of useful data every 0.12ms at 100000 bauds, which allows to transmit the 25 bytes of an S-Bus frame in 3ms at least), without losing too much (as soon as the buffer of 64 bytes reaches saturation).

- **decodeSBusFrame**, which will decode the last previously filled S-Bus frame, extracting from it the values of the 18 channels, failsafe and the lost frame indicator.

That's exactly what Simon Levy offers in his **SBUSRX** library<sup>27</sup> inspired by Borderflight's **SBUS** Library<sup>28</sup>.

Inspired by a code found in Ordinoscope<sup>29</sup>, we propose here to slightly simplify the procedure of updating S-Bus frames, while making a small improvement to take into account the following problem : since the size of the FIFO buffer is 64 bytes by default, it can contain up to 2 complete S-Bus frames and the start of the next S-Bus frame. So, if we are sure that there was no overflow of the buffer (we'll see how), it is better to fill 2 tables while emptying this buffer :

- **\_SBusFrame** which will contain the last complete S-Bus frame ;
- **\_SBusFramePartial** which will contain the beginning of the following S-Bus frame.

If a call to **decodeSBusFrame** comes just after, we will use **\_SBusFrame**. And to avoid useless copying when we exchange the roles of these 2 tables, only their addresses will be swapped.

To test that there has been no overflow of the buffer, it is enough to check that the time elapsed since the last call of **parseSBusFrame** is less than 15ms because, at 100000 bauds, it takes at least 15.68ms for fill a 64-byte buffer ; of course, this is only valid if the buffer is emptied in **parseSBusFrame**.

A last detail : for simplicity, we did not here standardized the values of the channels as proposes Simon Levy. If we want to do this by resuming its approach, we note here that strings such as "% + 2.2f" to format float type numbers are not operational. To solve this problem, it will suffice to use the **dtostrf** function, including the folder where it is defined in the Arduino Core library : "**avr / dtostrf.h**".

### 4.3.3 Programs

```

// DecodeSBus.ino

/*
  Decoding Futaba S-Bus protocol
  Libraries to be used :
  - DecodeSBus
  - ElapsedMillis from Paul Stoffregen (https://www.arduino-libraries.info/libraries/elapsed-millis)
  The decoded channel values are sent on the USB port with a format that can be read by Serial
  Oscilloscope
  (http://x-io.co.uk/serial-oscilloscope/)
*/

```

<sup>27</sup> <https://github.com/simondlevy/SBUSRX>

<sup>28</sup> <https://github.com/bolderflight/SBUS>

<sup>29</sup> <https://www.ordinoscope.net/index.php/Electronique/Protocoles/SBUS>

```

#include "DecodeSBus.h"      // DecodeS-Bus library
#include "elapsedMillis.h"   // EllapsedMillis library

#define PERIOD_DISPLAY 200  // Period for displaying the results (ms)

uint16_t channels[18];      // Channels
bool failsafe;              // Failsafe
bool lostFrame;             // Lost frame
elapsedMillis timerDisplay;  // Timer for displaying the results (ms)
DecodeSBus dSBus(Serial1);  // S-Bus decoder object using Serial1

void serialEvent1(void)
{
    dSBus.parseSBusFrame();
}

void setup ()
{
    // USB output init
    Serial.begin(115200);

    // S-Bus object init
    dSBus.begin();
}

void loop ()
{
    if (timerDisplay >= PERIOD_DISPLAY)
    {
        // Timer reset
        timerDisplay = 0;

        // S-Bus decoding and results display
        if (dSBus.SBusFrameOk())
        {
            dSBus.decodeSBusFrame(channels, failsafe, lostFrame);
            char tmp[24];
            sprintf(tmp, "%i,%i,%i,%i,%i,%i,%i\r", channels[0], channels[1], channels[2], channels[3],
                failsafe, lostFrame);
            Serial.print(tmp);
        }
    }
}

// DecodeSBus.h

/*
Decoding Futaba S-Bus protocol library
Adapted from
- https://www.ordinoscope.net/index.php/Electronique/Protocoles/SBUS
- https://github.com/simondlevy/SBUSRX
Tested on a Grasshopper development board (Tlera Corp)
(https://www.tindie.com/products/TleraCorp/grasshopper-loralorawan-development-board/)
with the Arduino Core for STM32L0 from Thomas Roell (https://github.com/GrumpyOldPizza/ArduinoCore-stm32l0)
*/

#pragma once

#include "Arduino.h"

class DecodeSBus
{
public:
    DecodeSBus(HardwareSerial &serial);
    void begin();
    void end();
    bool SBusFrameOk();
    void parseSBusFrame();
    void decodeSBusFrame(uint16_t channels[], bool &failsafe, bool &lostFrame);
private:
    static const uint16_t TIMEOUT = 15000; // Buffer overloaded risk
    HardwareSerial* _serial;               // Serial object (Serial1, Serial2 or Serial3)
    uint8_t *_SBusFrame;                   // Last complete S-Bus frame
    uint8_t *_SBusFramePartial;            // Partial S-Bus frame
    uint8_t _tFrames[50];                  // Array for these 2 frames
    uint8_t _idx;                          // Index of the last byte to update in _SBusFramePartial

```

```

        bool _SBusFrameOk;           // True if _SBusFrame can be decoded, false if not
        uint32_t _dateOld;           // Date (us) of the previous parseSBusFrame call
};

// DecodeSBus.cpp

#include "DecodeSBus.h"
#include <STM32L0_UART.h>

DecodeSBus::DecodeSBus(HardwareSerial &serial)
{
    _serial = &serial;
    _idx = 0;
    _SBusFrameOk = false;
    _SBusFrame = &_tFrames[0];
    _SBusFramePartial = &_tFrames[25];
    _dateOld = 0;
}

void DecodeSBus::begin()
{
    _serial->begin(100000, SERIAL_SBUS);
}

void DecodeSBus::end()
{
    _serial->end();
}

bool DecodeSBus::SBusFrameOk()
{
    return _SBusFrameOk;
}

void DecodeSBus::parseSBusFrame()
{
    // Buffer overloaded test
    uint32_t dateNow = micros();
    if ((dateNow - _dateOld) > TIMEOUT)
    {
        _idx = 0;
        _dateOld = dateNow;
    }

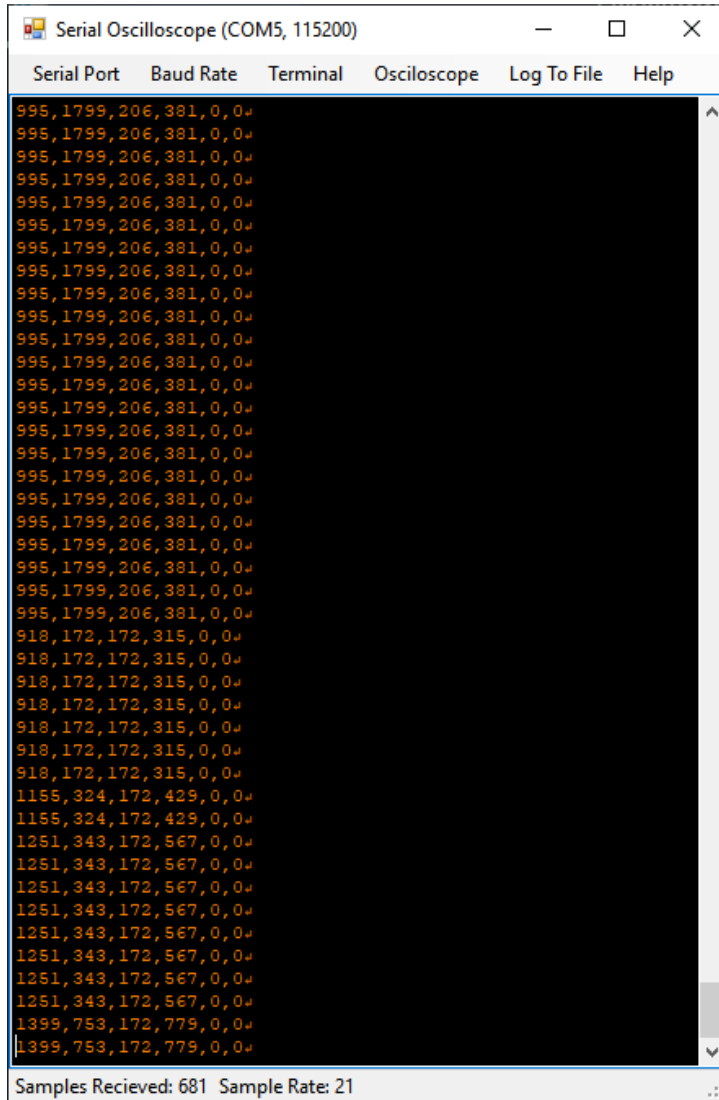
    // Parsing the buffer
    uint8_t b;
    while (_serial->available ())
    {
        b = _serial->read ();
        if (_idx == 0 && b != 0x0F) {}
        else { _SBusFramePartial[_idx++] = b; }
        if (_idx == 25)
        {
            _idx = 0;
            if (_SBusFramePartial[24] != 0x00) {}
            else // new complete frame found
            {
                _SBusFrameOk = true;
                uint8_t *sav = _SBusFrame;
                _SBusFrame = _SBusFramePartial;
                _SBusFramePartial = sav;
            }
        }
    }
}

void DecodeSBus::decodeSBusFrame(uint16_t channels[], bool &failsafe, bool &lostFrame)
{
    if (_SBusFrameOk)
    {
        channels[0] = ((_SBusFrame[1] | _SBusFrame[2]<<8) & 0x07FF);
        channels[1] = ((_SBusFrame[2]>>3 | _SBusFrame[3]<<5) & 0x07FF);
        channels[2] = ((_SBusFrame[3]>>6 | _SBusFrame[4]<<2 | _SBusFrame[5]<<10) & 0x07FF);
        channels[3] = ((_SBusFrame[5]>>1 | _SBusFrame[6]<<7) & 0x07FF);
        channels[4] = ((_SBusFrame[6]>>4 | _SBusFrame[7]<<4) & 0x07FF);
        channels[5] = ((_SBusFrame[7]>>7 | _SBusFrame[8]<<1 | _SBusFrame[9]<<9) & 0x07FF);
        channels[6] = ((_SBusFrame[9]>>2 | _SBusFrame[10]<<6) & 0x07FF);
        channels[7] = ((_SBusFrame[10]>>5 | _SBusFrame[11]<<3) & 0x07FF);
    }
}

```

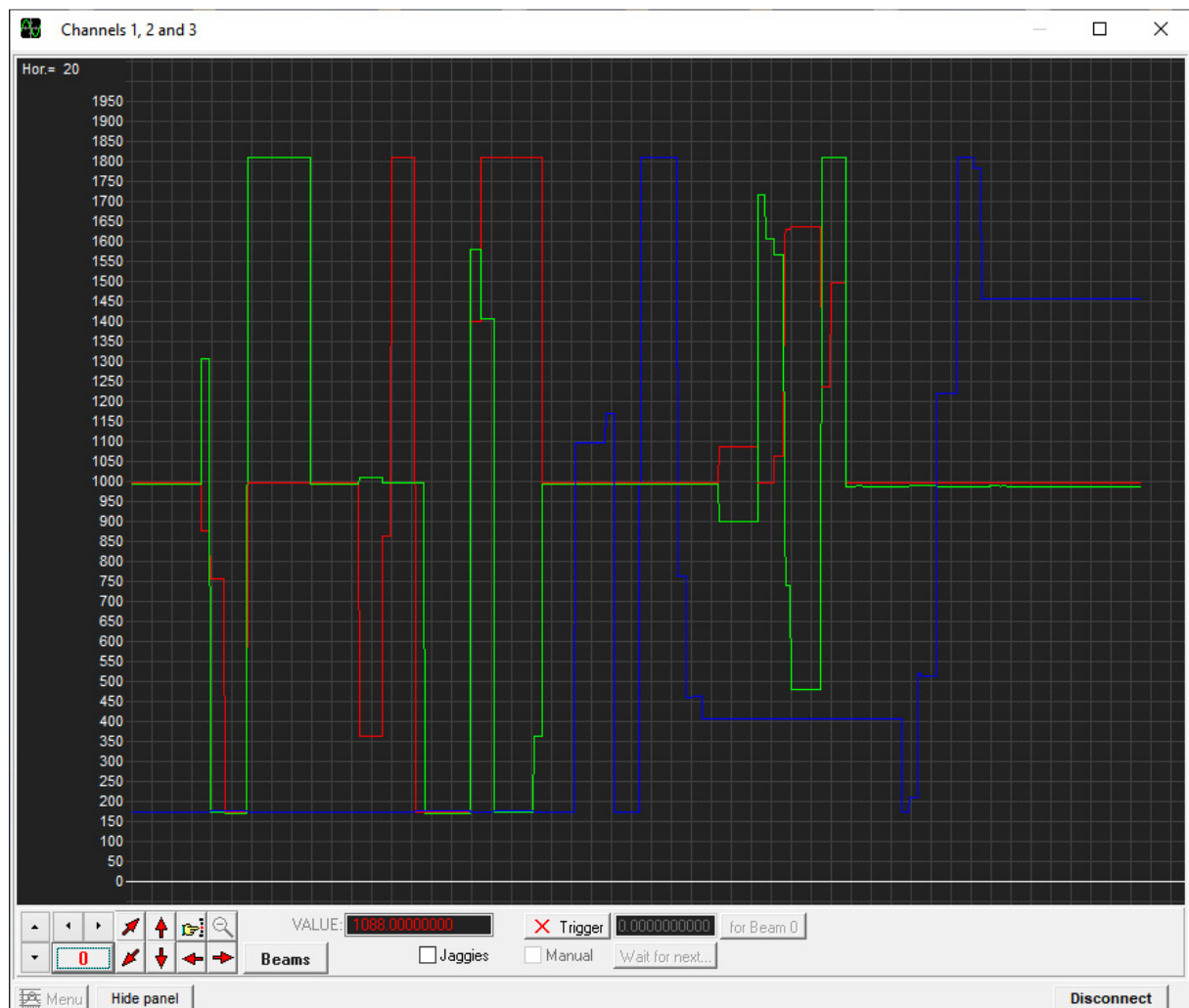
$$\left. \begin{array}{l} \{ \\ \} \end{array} \right\}$$

Displays with **Serial Oscilloscope**<sup>30</sup> :



<sup>30</sup> <http://x-io.co.uk/serial-oscilloscope/>





Failsafe test when the radio is switched off :

