RODOS

Serial Protocol / encoding (S3P)

Version: 2.0
Date: Sept. 2019
Author: S. Montenegro

This document explains the S3P message encoding for serial transfers on byte streams (which are not message oriented). This encoding protocol is independent from endianity, but we shall mention, RODOS uses big-endian encoding for all message transfers and our S3P Commands are encoded in big-endian too. All RODOS Messages are protected with checksums or CRCs. This protection will not be handled here. S3P is only a framing for messages and encoding of special commands.

1. Message Framing and Commands

S3P contains special commands for example for time synchronization, and data flow. These features are optional and seldom implemented. In this specification they are in gray font. These special commands may be transmitted at any time, eg. outside of messages or inserted in a data transfer without corrupting data.

S3P Commands are dual bytes beginning with a mark byte (0xFE) followed by a control code. S3P uses byte stuffing (^) for the cases where user data contains a value which corresponds to the mark (0xFE). If you are wondering why not 0xFF: if is because 0xFF occurs very often in user data. Definition of S3P Commands:

Command	Symbol	Code	Meaning
BOM	[	0xFE 0x02	Begin of Message
EOM	]	0xFE 0x03	End of Message
Stuffing	^	0xFE 0x7E	A single bye of user data: 0xFE
Stop	⏸	0xFE 0x13	Stop (for data flow control)
Continue	⏵	0XFE 0x11	Continue (for data flow control)
Sync	●	0xFE 0x16	Time-Sync (like one-pulse-per-second)

A message is encapsulated between BOM (0xFE 0x02) and EOM (0xFE 0x03). It may contain an arbitrary number of bytes. If 0xFE (Mark) occurs in the user data then it sends a Stuffing double byte (0xFE 0x7E) instead, which the receiver will translate to a single byte 0xFE

Stop (0xFE 0X13) and Continue (0xFE 0x11) are used for data flow like traditional UART ^s and ^q. When the receiver can not store more bytes it shall send a Stop command, when it is ready to continue receiving it shall send a Continue command. These commands may be inserted in an ongoing message transfer.

Time-Sync is used as “One Pulse Per Second” (1PPS) to synchronize clocks and can be send at any time, inside and outside of messages.

2. Encoding Examples

The message

0x1	0x2	0x3	0xFE	0x4					

will be encoded as

[	0x1	0x2	0x3	^	0x4	]			

Binary:

0xFE	0x2	0x1	0x2	0x3	0xFE	0x7E	0x4	0xFE	0x3

These Messages
'H' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd'
0x11, 0x22, 0x33, 0x44,
0x11, 0x22, 0x33, 0XFE, 0x44

Will be encoded as
['H' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' **][**0x11 0x22 0x33 0x44**][**0x11 0x22 0x33 **^** 0x44**]**

For example, the last message will be transmited as:
0xFE 0x02 0x11 0x22 0x33 0xFE 0x7E 0x44 0xFE 0x03

And the receiver application will get (the same which the sender application sends)
0x11 0x22 0x33 0xFE 0x44

We can insert other commands in the transmissions, for example
[0x11 ⏸ 0x22 0x33 **^** 0x44 0x55 ⏺ 0x66 0x77 ⏵ 0x88 0x99 **]**

The receiver application will not get ⏸ ● ⏵ and instead of ^ (0xFE 0x7E) just 0x7F.

Please remember, S3P is only a message framing, which was designed to be robust against transmission errors. The specification includes no header formats or CRC check sums. This shall be provided/implemented in the next layer. S3P avoids to encode the end of the message as a function of a length field. This method (using length) is too susceptible to transmission errors.

Examples of message interpretation

Incoming data	***Internal registered data***
`[aaa][bbb]`	`Message1: aaa` `Message2: bbb`

`xxxx[aaa]yyy[bbb]zzzz[ccc]`	`Message1: aaa`
	`Message2: bbb`
	`Message3: ccc`

`aaaa][bbb]xxx[cc[dddd][eee]`	`Message1: bbb`
	`Message2: ddd`
	`Message3: eee`

`[aa⏸a]xxx⏵[b● bb]●`	`Message1: aaa` `Message2: bbb`

3. Implementation examples

This is a minimalistic example which does not implement control flow and time sync. You can find a complete implementation in the support lib class S3pEncoder. Please note: The functionality is the same tbut he implementation differs.

Common definitions

```
#include "stdint.h"

extern void    myPutChar(uint8_t c);
extern uint8_t myGetChar();

static const uint16_t  BOM     =  0xFE02; // independent of endinaty we send msf
static const uint16_t  EOM     =  0xFE03;
static const uint16_t  STUFF   =  0xFE7E;
static const uint16_t  COMMAND =  0xFE00;
static const uint8_t   MARK    =  0xFE;
#define IS_COMMAND(_c)   (((_c) >> 8) == MARK)
```

To send messages

Sends S3P commands as dual bytes and normal characters as single bytes using a low level/Hardare-Driver “myPutByte()” function.

```
void putDualByte   (uint16_t c) { myPutChar(c >> 8); myPutChar(c & 0xff); }

void putByteEncoded(uint8_t c)  {
    if(c == MARK) putDualByte(STUFF);
    else myPutByte(c);
}

void sendMsg(int len, uint8_t* s) {
    putDualChar(BOM);
    for(int i = 0; i < len; i++) putByteEncoded(*s++);
    putDualChar(EOM);
}
```

To receive messages by polling

Receives single bytes using a low level/Hardware Driver function “myGetChar()”, then group them in a dual byte and search for S3P commands. It ignores all characters until it finds a BOM. Then it copies normal bytes to the user buffer until the maximal length or until EOM is found. If a (new) BOM is found before the expected EOM, we assume the last message was corrupted, it will be ignored ad we begin with a new message.

```
uint16_t getDualByte() { // Only the first time it will deliver an unexpected 0
    static uint16_t econdded = 0;
    return (econdded = (econdded << 8 ) | myGetChar());
}


int getMsg(int maxlen, uint8_t* s) {
    int len = -1;
    while(1) {
        if(len >= maxlen) return maxlen;
        uint16_t c = getDualByte(); // Warning: it suspends until data arrives
        if(c == BOM)      { len = 0;          continue;  } //start collecting bytes
        if(len < 0)       {                   continue;  } //Ignore all before BOM
        if(c == STUFF)    { s[len++] = MARK; continue;  }
        if(c == EOM)      {                   return len;}
        if(IS_COMMAND(c)) {                   continue;  }//Ignore all other cmds
        if((c & 0xff) != MARK) s[len++] = c & 0xff;
    }
}
```

To receive messages by up-calls, eg. as interrupt server

```
// The last buffer you gave is ready! Please give us a new buffer!
extern char* upcallMsgReady(int len);

void upcallInputChar(char inputChar) {

    static int16_t len      =   -1;
    static char*   msg      =    0;
    static int16_t maxLen   = 1300; // just as example
    static int16_t dualChar =    0;
    if(msg == 0)   msg      = upcallMsgReady(0); // only the first time

    dualChar  = (dualChar  << 8 ) | inputChar;

    if(len >= maxLen)     { msg = upcallMsgReady(len); len = -1; }
    if(dualChar == BOM)   { len = 0;           return; } //start collecting
    if(len < 0)           {                    return; } //Ignore all before BOM
    if(dualChar == STUFF) { msg[len++] = MARK; return; }
    if(dualChar == EOM)   { msg = upcallMsgReady(len); len = -1; return; }
    if(IS_COMMAND(dualChar)) {                 return; }//Ignore all other cmds
    if(inputChar != MARK ) msg[len++] =  inputChar;
}
```