Simple Reliable Serial Communication protocol (SRSC)

Table of contents

1 Introduction

SRSC is a simple communication protocol that is built on top of serial communication and extends it with data integrity checking, message format definition and buffer overflow protection.

It is inspired by UDP and Robust Arduino Serial protocol.

2 Principle

SRSC starts communication with simple connection. It tels both actors that the other one is listening and what is the size of opposite ones serial buffer. When the connection is established, both actors can start sending packets. Those have defined types that determine which action should be performed on the receiver side, what data is transferred, and the size of the data. Structure and types of packets are described in chapter *3* and *4*.

Several error states can occur during communication - these are addressed by the protocol. The first of them is data corruption or complete loss of packet content. Each packet therefore contains a checksum that allows the receiver to determine the validity of the packet. If the packet is corrupted, the receiver discards it. This principle implies a potential problem for the protocol, which is described at the end of this chapter.

The second problem is buffer overflow. Some platforms have a very limited buffer for storing serial communication messages (e.g. standard Arduino has 64 bytes). Therefore, SRSC implements a semaphore and packet delivery acknowledgement method to guarantee that if the buffer is full on the receiver side, no data will be sent from the sender until the buffer is freed. The semaphore size is calculated according to the buffer size (`semaphore_size = opponents_buffer_size / max_packet_size`, rounded down). The principle of semaphore operation is as follows: when the sender sends a packet, the semaphore value is decreased (on the sender side), then when the ACCEPTACK packet is received, it is increased again. In case the semaphore value drops to 0, the sending of messages is suspended until it is increased again.

The protocol is primarily designed for fast data transfer in streams (e.g. measured values or continuous series of commands with small changes). For this reason, there is no implementation of resending corrupted packets. If an application requires sending packets where delivery guarantee is critical, appropriate safeguards must be taken. One possibility is to use so-called critical packets, which contain an ID and are sent multiple times (5x). This greatly reduces the possibility of data loss. When multiple packets with the same ID arrive in sequence, the first uncorrupted packet is accepted, the others are discarded.

3 Packet structure

Byte	**0**	**1**	**- / 2**	**(2 - 5) / (3 - 6)**
Information	Packet type	Checksum	Packet ID	Payload

3.1 Packet type

- *size*: 1 byte
- *position*: 0

Packet type represents action or type of data in the packet payload.

3.1.1 Packet type groups

1. *Control packets*
   - packet that represents an action (does not transfer data)
   - e.g. ACCEPTACK
2. *Data packets*
   - packets used to transfer data

3.1.2 Packet type ranges

- *0 - 63* = (64) control packets
  - *0 - 15* = (16) reserved for future versions of the protocol
- *64 - 254* = (191) data packets
  - *64 - 95* = (32) reserved for future versions of the protocol
- *255* = restricted (it is used by serial communication to indicate empty buffer)

3.2 Checksum

- *size*: 1 byte
- *position*: 1

Sum used for error detection. It uses *1's Complement* algorithm.

It detects following kinds of error:

- `n` number bit flip
- complete data loss

3.2.1 Calculation process

1. all bytes of the packet are summed together and anything exceeding 8 bits is discarded
2. the counted byte is then bitwise inverted and inserted to the packet
3. on the receiver side all bytes of the packet (including the checksum) are added together
4. the result is compared with the number 255 (`0b11111111`) - if they are equal, the packet arrived uncorrupted

Error Masking Probability paper

3.3 Packet ID

- *size*: 0 | 1 byte
- *position*: 2

Number used to identify series of critical packets. It's NOT totally unique identifier - it cycles over 256 values.

Non-critical packets do not contain an ID byte.

3.4 Payload

- *size*: 0 | 1 | 2 | 4 byte(s)
- *position*:
  - 2 - 5 (if ID byte is not present)
  - 3 - 6 (if ID byte is present)

Numerical information transmitted by the packet. Size of payload depends on the packet type.

4 Reserved packet types

4.1 Control packets

4.1.1 CONNECT

- *code*: `0x00`

An attempt to establish connection. It Transmits the size of the sender's buffer (if the buffer is unlimited, the value is 0). If the packet is corrupted, the receiver discards it and the sender resends it after a certain period of time.

4.1.1.1 Packet structure

Byte	**0**	**1**	**2 - 5**
Information	`0x00`	Checksum	Size of the buffer

4.1.2 CONNACK

- *code*: `0x01`

Response to CONNECT packet. It is sent if the sender has received a valid connection attempt. It serves as a CONNECT packet for the sender. The sender uses it to send its buffer size. If it is successfully received by the receiver, the connection is successfully established.

4.1.2.1 Packet structure

Byte	**0**	**1**	**2 - 5**
Information	`0x01`	Checksum	Size of the buffer

4.1.3 ACCEPTACK

- *code*: `0x02`

A notification that tells an transmitter that the packet has arrived correctly to the receiver and the transmitter can increase the semaphore value.

4.1.3.1 Packet structure

Byte	**0**
Information	`0x02`

4.2 Restricted

4.2.1 Type 255

Value 255 (0xFF) is used by serial communication libraries to indicate empty packet buffer, so the packet type will be indistinguishable.

5 Communication process flow

5.1 Actors

- **A** - communication participant 1
- **B** - communication participant 2
- **DP** - example data packet with type `0x40` and payload `0x05` (one byte)

5.2 Process

- *A*: builds CONNECT packet (with its buffer size)
- *A*: sends CONNECT packet
- *B*: accepts *A* CONNECT packet
- *B*: saves *A* buffer size
- *B*: builds CONNACK packet (with its buffer size)
- *B*: sends CONNACK packet to *A*
- *A*: accepts *B* CONNACK packet => connection is established
- *A*: saves *B* buffer size
- ...
- *A*: builds *DP* with given data
- *A*: counts checksum for *DP*
- *A*: sends *DP* to *B*
- *A*: decreases semaphore value
- *B*: receives *DP*
- *B*: validates integrity of *DP* => DP is ok
- *B*: gives *DP* to responsible handler
- *B*: sends ACCEPTACK packet to *A*
- *A*: accepts ACCEPTACK packet
- *A*: increases semaphore value

6 Implementation

- Arduino (C++)
- Java

7 Contribution

Contributions are highly welcomed, both for the protocol definition and for its implementations.

8 License

The protocol specification is licensed under CC-BY 4.0 license.