SPI Interface
SPI interface and implementation in u-blox wireless module
Application Note

Abstract
Description of the modified SPI (Serial Peripheral Interface) protocol, used for high speed connection between LISA-U modules and an application processor.
<table>
<thead>
<tr>
<th><strong>Document Information</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title</strong></td>
<td>SPI Interface</td>
</tr>
<tr>
<td><strong>Subtitle</strong></td>
<td>SPI interface and implementation in u-blox wireless module</td>
</tr>
<tr>
<td><strong>Document type</strong></td>
<td>Application Note</td>
</tr>
<tr>
<td><strong>Document number</strong></td>
<td>UBX-13001919</td>
</tr>
<tr>
<td><strong>Document revision</strong></td>
<td>A</td>
</tr>
<tr>
<td><strong>Document status</strong></td>
<td>Preliminary</td>
</tr>
</tbody>
</table>

This document and the use of any information contained therein, is subject to the acceptance of the u-blox terms and conditions. They can be downloaded from www.u-blox.com. 

u-blox makes no warranties based on the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. 

u-blox reserves all rights to this document and the information contained herein. Reproduction, use or disclosure to third parties without express permission is strictly prohibited. Copyright © 2013, u-blox AG.
Contents

Contents.......................................................................................................................3

1 Introduction ..............................................................................................................5

2 HW Interface ..........................................................................................................6

3 Protocol format .......................................................................................................7
  3.1 General ..................................................................................................................7
  3.2 SPI header ............................................................................................................7
  3.3 Byte ordering ..................................................................................................... 8
  3.4 Bit ordering ......................................................................................................... 9
  3.5 SPI frame sizes ................................................................................................... 9
  3.6 Invalidating SPI packets .....................................................................................10
  3.7 Usage of RTS, CTS and MORE ..........................................................................10
  3.8 State machines ..................................................................................................12
    3.8.1 Slave state machine ....................................................................................12
    3.8.2 Master state machine ................................................................................13

4 Communication protocol .......................................................................................14
  4.1 General ................................................................................................................14
  4.2 Slave initiated transfer to the Master ..................................................................14
  4.3 Master initiates transfer with sleeping Slave .....................................................15
  4.4 Frame end ..........................................................................................................15

5 Timing ......................................................................................................................17

6 Troubleshooting .....................................................................................................18
  6.1 Fast change of RDY lines could not be detected ................................................18
  6.2 Slave reboot outside SPI transfer ......................................................................18
  6.3 Slave reboot during SPI transfer ......................................................................18
  6.4 Master reboot outside SPI transfer ....................................................................18
  6.5 Master reboot during SPI transfer ....................................................................18
  6.6 Recovery mechanism for slave ........................................................................18

7 Protocol examples ..................................................................................................20
  7.1 Communication example ..................................................................................20
    7.1.1 How to fill the header and data buffer - Example ......................................20
    7.1.2 Multi-frame transfer example ....................................................................21
  7.2 Flow control ......................................................................................................22
    7.2.1 Data transmission stopped ........................................................................22
    7.2.2 Data transmission continues .....................................................................23
    7.2.3 Both sides signal flow control ..................................................................24
  7.3 MORE condition ..............................................................................................25
8  Implementation comments.................................................................26
  8.1  SPI clock rates on LISA-U1/LISA-U2 series..........................................26
  8.2  Estimation of available bandwidth ..........................................................26
      8.2.1  General considerations .................................................................26
      8.2.2  Dependency on different parts of SPI transmission .........................26
  8.3  Settings for LISA-U1/LISA-U2 series driver ............................................28

9  Additional notes.......................................................................................29
  9.1  State machines for implementation ..........................................................29
      9.1.1  Master (Application Processor) .........................................................29
      9.1.2  Slave (LISA-U1/LISA-U2 series module) ............................................30
  9.2  Pseudo code ..........................................................................................31

10 SPI debug command +USPITRACE ............................................................32
  10.1  Defined Values .....................................................................................32

Related documents....................................................................................33

Revision history.........................................................................................33

Contact.......................................................................................................34
1 Introduction

This document is a guideline describing how to connect a LISA-U1 / LISA-U2 series module to an application processor via the Serial Peripheral Interface (SPI). In LISA-U1/LISA-U2 series, two additional signal lines are used to communicate the state of readiness of the two processors. It is assumed that the communication over the SPI is error free.

The SPI interface allows high-speed communication in both directions simultaneously. Since most application processors do not support a high-speed asynchronous interface, a synchronous protocol is used.

The application processor sees LISA-U1/LISA-U2 series modules connected via the SPI interface as any other serial device. All the control lines should be used. The transmission and reception of data is similar to an asynchronous device.

SPI is a master-slave protocol. LISA-U1/LISA-U2 series modules implement the slave side.

In the following sections the processor with the master communication role is called master and the processor with the slave communication role is called slave.

The following symbols are used to highlight important information within the document:

- An index finger points out key information pertaining to integration and performance.
- A warning symbol indicates actions that could negatively impact performance or damage the device.

This document applies to the following products:

- LISA-U1 series
- LISA-U2 series
2 HW Interface

The HW interface uses five wires plus the ground connection. Table 1 lists the signals.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_MISO</td>
<td>SPI Data Line. Master Input, Slave Output</td>
<td>Module Output. Idle high. Data is transferred from Slave to Master</td>
</tr>
<tr>
<td>SPI_MOSI</td>
<td>SPI Data Line. Master Output, Slave Input</td>
<td>Module Input. Idle high. Internal active pull-up to V_INT (1.8 V) enabled Data is transferred from Master to Slave</td>
</tr>
<tr>
<td>SPI_SCLK</td>
<td>SPI Serial Clock. Master Output, Slave Input</td>
<td>Module Input. Idle low. Internal active pull-down to GND enabled Data clock (generated by master)</td>
</tr>
<tr>
<td>SPI_MRDY</td>
<td>SPI Master Ready to transfer data control line. Master Output, Slave Input</td>
<td>Module Input. Idle low. Internal active pull-down to GND enabled Master active and ready to transfer data. Similar to Select Slave (SS) on SPI</td>
</tr>
<tr>
<td>SPI SRDY</td>
<td>SPI Slave Ready to transfer data control line. Master Input, Slave Output</td>
<td>Module Output. Idle low. Slave active and ready to transfer data</td>
</tr>
</tbody>
</table>

Table 1: SPI interface signals on LISA-U1/LISA-U2 series

The defined HW interface differs slightly from the standard SPI protocol.

SPI_MOSI and SPI_MISO are active low (see section 8.3).

Figure 1 shows the line usage:

- The frame-size is known by both sides before a packet-transfer of each packet. The same amount of data is exchanged in both directions simultaneously.
- Both sides set their SPI_MRDY and SPI_SRDY lines independently when they are ready to transfer data. The other side must wait for the activating interrupt to allow the transfer of the other side.
- The master starts the clock shortly after SPI_MRDY and SPI_SRDY are set to active. The **number of clock cycles depends on the SPI frame size** (see section 3.5), **to be transferred**
- The SPI_SRDY line will be set down after the end of the clock.
- Usually the SPI_MRDY line is also set to inactive with the end of the clock, but when there is a large transfer containing multiple packets, the SPI_MRDY line stays active. This is described in detail later in this document.

![Figure 1: Line usage of the modified SPI Protocol](image-url)
In the diagrams in the following sections DATA EXCHG means communication over all three SPI signals; the SPI_SCLK, SPI_MOSI, SPI_MISO signals are not shown. The timing can vary when SPI_SRDY and SPI_MRDY are lowered at the end of a transfer. There is no fixed relation here, so SPI_SRDY can be lowered before SPI_MRDY goes low or vice versa, or both will be lowered at the same time. This depends on the timing of both chips.

3 Protocol format

3.1 General

A serial protocol with a fixed header size and payload controls the communication.

<table>
<thead>
<tr>
<th>Header (4 bytes)</th>
<th>Payload (multiple of 4 bytes)</th>
</tr>
</thead>
</table>

Figure 2: SPI-frame

The size of the header is always 4 bytes. The size of the payload (bytes) must be a multiple of 4.

After the power-up, the payload-size is set to a fixed value of DEF_BUF_SIZE, to prevent the master and slave from losing synchronization if a silent reset of the master or slave occurs.

The maximum payload-length MAX_BUF_SIZE and the default payload-length DEF_BUF_SIZE are fixed values defined in the firmware, depending on the usage and the DMA-size of both sides. Section 8.3 describes the settings for LISA-U1/LISA-U2 series modules.

In the LISA-U1/LISA-U2 series the maximum for DEF_BUF_SIZE is MAX_BUF_SIZE. In this case the system works with a fixed buffer size.

The values of DEF_BUF_SIZE and MAX_BUF_SIZE concern the payload and do not include the header.

A partial (CTS and RTS flags) software emulation of the RS232 interface is realized via the SPI Header. The control information about the RS232 status lines can be exchanged between the master and slave by setting the relevant flags in the SPI Header.

Throughout this document the term payload refers to the SPI payload if not otherwise stated.

3.2 SPI header

The SPI header size is 4 bytes. It has the following structure:

<table>
<thead>
<tr>
<th>Bit n.</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>24</th>
<th>20</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DSR</td>
<td>DTR</td>
<td>CTS</td>
<td>RTS</td>
<td>DCD</td>
<td>RI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit n.</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RES=0</td>
<td>RES=0</td>
<td>MORE</td>
<td></td>
<td></td>
<td>Current data size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: SPI header (DSR/CTS bits sent by module, DTR/RTS bits sent by application processor)
These flags are written by both master and slave:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| MORE | The flag is set, if at the time of writing this packet, more data is available  
- 0 nothing more to send  
- 1 more data to send |
| Current data size | The amount of valid data in this payload. (Since it is not possible to send a partial payload, the master and the slave must fill the 2044 bytes long payload with padding bytes after sending the valid data.) |
| Next Data Size | The size of the payload required for the next-transfer on multiple transfers.  
It is always set to DEF_BUF_SIZE (2044, binary 011111111100) due to fixed frame size handling.  
Every new transfer (SPI_MRDY has gone low or last packet was sent with MORE flag removed) will start with a fixed payload size of DEF_BUF_SIZE bytes. |

“Next Data Size” and “Current data size” refer only to the payload data, not to the header.

These flags are written by the slave (module):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| CTS | CTS Flow-Control information of the module (same flag as RTS)  
- 0 the module is able to receive data  
- 1 the module is not able to receive data |
| RI | Ring indicator. Set to 1 when a ring on an actual phone would be needed, e.g. in case of an incoming call or SMS |
| DCD | Data Carrier Detect  
- 1 the terminal is in command mode  
- 2 the terminal is in data mode (data connect, SMS text mode, file download) |

These flags are written by the master (terminal):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| RTS | Flow-Control information of the terminal (same flag as CTS)  
- 0 the terminal is able to receive data  
- 1 terminal is not able to receive data |
| RI | Ring indicator. Set to 1 when a ring on an actual phone would be needed, e.g. in case of an incoming call or SMS |

Typically - but not necessarily - the module is the slave and the terminal the master. Most of the RS232-flags are set to 0 if everything is OK. This means the slave can just send a 0 header if nothing is going on, and does not need to start a transfer if the HW is set to automatically send 0 (see section 3.6).

RES is reserved for future use.

DSR, DCD, RI, DTR, RES are currently not supported. Any flag which is not supported shall be ignored if set.

### 3.3 Byte ordering

The byte ordering of the header is LSB (Least Significant Byte is transmitted first) and cannot be changed during runtime. For details refer to section 8.3. The bit ordering is described in the section 3.4.
The header is read out starting from byte 0. See Figure 4 as an example.

<table>
<thead>
<tr>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>(bits 24-31)</td>
<td>(bits 16-23)</td>
<td>(bits 8-15)</td>
<td>(bits 0-7)</td>
<td></td>
</tr>
</tbody>
</table>

Header and payload order according to section 3.2

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>(bits 0-7)</td>
<td>(bits 8-15)</td>
<td>(bits 16-23)</td>
<td>(bits 24-31)</td>
<td></td>
</tr>
</tbody>
</table>

Header and payload order as they are sent/received over the SPI lines

**Figure 4: SPI-frame over the SPI lines with data format MSB (Most Significant Byte) first**

### 3.4 Bit ordering

The bit ordering of the header and payload bytes is msb (most significant bit is transmitted first) and cannot be changed during runtime. For details refer to section 8.3.

Example: the ASCII character ‘a’ (decimal 97, hexadecimal 0x61, binary 0110.0001) is transmitted as the sequence 0,1,1,0,0,0,0,1. See Figure 6 as an example.

```plaintext
... 0 1 1 0 0 0 0 1 ...
```

**Figure 5: SPI-frame over the SPI data lines with data format msb first**

### 3.5 SPI frame sizes

The SPI protocol is designed to work either with fixed or variable payload length. Variable frame sizes have been used for smaller throughput. For HSPA, fixed frame sizes with 16 byte alignment (including header) provides optimal performance, see section 8.3.

Consequently each SPI packet has a fixed frame size regardless of the amount of valid data to transmit.

<table>
<thead>
<tr>
<th>Header (4 bytes)</th>
<th>Payload (DEF_BUF_SIZE)</th>
</tr>
</thead>
</table>

**Figure 6: SPI fixed frame size**

When the amount of data to be transferred is less than `DEF_BUF_SIZE`, the remaining bytes can have any values.

**Important SPI Header content:**
- “Current data size” field: Amount of valid data within the SPI payload (value must be DEF_BUF_SIZE or less)
- “Next data size” field: always `DEF_BUF_SIZE`

Be aware that the SPI payload length is not aligned with the payload content. This means that if, for example, a Layer 2 protocol like MUX (3GPP 27.010) is transferred, then a MUX frame might be split into two SPI packets if it does not fit into one SPI packet. It can also happen that a SPI packet contains more than one MUX frame.

It is the responsibility of the upper layers (Layer 2, etc.) to extract and reassemble their frames out of the SPI packets. This is beyond the scope of this document and will not be covered here.
3.6 Invalidating SPI packets

Since the SPI protocol forces sending empty data, it is mandatory to clearly mark such invalid SPI packets.

To invalidate an outgoing SPI packet, set the header either to 0xffffffff or to 0x00000000. The receiving party will act as follows:

- When receiving a header set to 0x00000000, the “Current data size” is 0: no data is received
- When receiving a header set to 0xffffffff, the “Current data size” as well as the “more data” bit are forced to 0; all the other fields keep the values of the previously received valid frame

As an example, it can be used by a system keeping the RX line active or inactive during the data transfer without really sending data.

<table>
<thead>
<tr>
<th>Field</th>
<th>0x00000000</th>
<th>0xffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS/CTS</td>
<td>0</td>
<td>State of previous frame</td>
</tr>
<tr>
<td>DTR/DSR</td>
<td>0</td>
<td>State of previous frame</td>
</tr>
<tr>
<td>DCD</td>
<td>0</td>
<td>State of previous frame</td>
</tr>
<tr>
<td>RI</td>
<td>0</td>
<td>State of previous frame</td>
</tr>
<tr>
<td>MORE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CURR_SIZE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Next_SIZE</td>
<td>DEF_BUF_SIZE</td>
<td>DEF_BUF_SIZE</td>
</tr>
</tbody>
</table>

Table 2: Behavior when receiving invalid SPI frames

3.7 Usage of RTS, CTS and MORE

The MORE, RTS and CTS flags define whether and how communication needs to be continued.

**RTS** / **CTS** are not used as HW flow control; they indicate the status of the higher layer buffer (the ability to receive valid data). The functionality is therefore not exactly the same as RTS / CTS on a UART. RTS / CTS will not stop the physical communication, they will only ensure that no valid data is sent from the master to the slave, or vice versa, depending on whether RTS or CTS is signaled.

For example, if the slave signals CTS during data transfer, the master will stop sending valid data. It can still send empty frames, but the slave will still be able to transmit data to the master.

At the beginning of each packet-transfer, master and slave check the state of their respective flags. The next transfer starts directly if the Equation 1 returns 1:

\[
\begin{align*}
(Master\_able\_to\_receive \&\& \text{Slave}\_has\_MORE) \\

|| (\text{Slave}\_is\_able\_to\_receive \&\& \text{Master}\_has\_MORE)
\end{align*}
\]

- **Master**\_able\_to\_receive: RTS = 0 in master header
- **Master**\_has\_MORE: MORE = 1 in master header
- **Slave**\_is\_able\_to\_receive: CTS = 0 in slave header
- **Slave**\_has\_MORE: MORE = 1 in slave header

**Equation 1: check of multiple packet transfer**

SPI only allows a transfer in both directions at the same time.

- If Equation 1 returns 0 the transfer will be stopped as described in section 4.4.
- If one or both are not able to exchange data due to RTS/CTS, the next packet will be handled as described in Table 3:
### Table 3: Table of the MORE and flow control usage

If one side raises the flow control then the other side is not allowed to transmit data until the flow control is removed. Removing flow control is done by sending an SPI packet with the relevant flag reset. The side which must obey the flow control may only set the MORE-flag if it has data to transmit. Only after it has received a packet indicating flow control removal it can continue sending data afterwards.

> When the flow control is signaled, it is the responsibility of the side exercising the flow control to signal its removal. Polling the flow controlling side to check the status is allowed, but not necessary. As a consequence SPI packets might be exchanged which contain only a valid header but no valid payload.

<table>
<thead>
<tr>
<th>Master RTS</th>
<th>Slave MORE</th>
<th>Slave CTS</th>
<th>Master MORE</th>
<th>Next Size(^1)</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DEF</td>
<td>No transfer</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>DEF</td>
<td>Transfer</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>DEF</td>
<td>No transfer; then slave initiates the transfer procedure if CTS=1(^2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>DEF</td>
<td>Wait for CTS = 0; then slave initiates the transfer procedure(^3)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>DEF</td>
<td>Transfer</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>DEF</td>
<td>Transfer</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DEF</td>
<td>Transfer with no data towards slave</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>DEF</td>
<td>Transfer with no data towards slave; the master must delay sending data until CTS=0; both sides can initiate the next transfer to signal RTS or CTS =0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DEF</td>
<td>No transfer; then the master initiates the transfer procedure if RTS=1(^2)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>DEF</td>
<td>Transfer with no data towards the master</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>DEF</td>
<td>No transfer</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>DEF</td>
<td>Wait for CTS=0; then the slave initiates the transfer procedure(^3)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>DEF</td>
<td>Wait for RTS=0; then the master initiates the transfer procedure(^3)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>DEF</td>
<td>Transfer with no data towards the master; the slave must delay sending data until RTS=0; both sides initiate the next transfer to signal RTS or CTS =0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>DEF</td>
<td>Wait for RTS=0; then the master initiates the transfer procedure(^2)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>DEF</td>
<td>As soon as the master or slave are able to receive, they will initiate the transfer procedure according to the rules above</td>
</tr>
</tbody>
</table>

\(^1\) DEF=DEF_BUF_SIZE  
\(^2\) This does not mean that the other side is not allowed to start a transfer, but it cannot send any data until the other side resets the RTS/CTS flag. If one side has set the RTS or CTS line it is possible to continue the transfer in the other direction
3.8 State machines

3.8.1 Slave state machine

The slave starts up in inactive mode with a data size of `DEF_BUF_SIZE`. The `MRDY_FLAG` will be set to 1 with the `SPI_MRDY` interrupt. It will be set to low before the transfer starts.

As soon as the slave gets an `SPI_MRDY` active interrupt or has data to transmit, it writes the HW-registers for the next transfer. This means if the master starts its clock it will automatically send the data. The slave waits for the end of the transfer signaled internally by the RX-interrupt. Now the slave checks the flow-control and more flags. If transfers are true, the next transfer will be started, otherwise it returns to the start state.

![State Machine Diagram](image-url)

**Figure 7**: State Machine of the Slave
3.8.2 Master state machine

The master starts up in inactive mode with a data size of DEF_BUF_SIZE. As soon as it receives an SPI_SRDY interrupt it will set the SRDY_FLAG = 1. If the SRDY_FLAG is 1 or the master has data to transmit, it will set the SPI_MRDY line to active. If the SRDY_FLAG is 0, it will wait for the slave to be ready to transmit data, causing a change of the SRDY_FLAG by the SPI_SRDY active interrupt. If the SRDY_FLAG is set, the master programs its registers and sets the SRDY_FLAG = 0 before starting the transfer. The RX-interrupt internally signals the end of the transfer. Now the master checks the flow-control and more flags. If the Equation 1 is true the next transfer will be started, otherwise it will go back to the start state. The master must reset the SPI_MRDY line at the end of the transfer.

Figure 8: State Machine of the Master
4 Communication protocol

This section describes three scenarios:

- Slave initiated transfer
- Master initiated transfer with a sleeping slave
- Frame end

Section 5 describes the error scenarios and the recovery mechanisms.

4.1 General

The master controls the SPI_SCLK and since data is shifted out only when the clock is applied, it is not necessary to lower the SPI_MRDY signal after each frame. The master can hold this line high during the complete transfer (multi-frame transfer); it must only ensure correct SPI_SRDY transition detection to activate the SPI_SCLK.

It is allowed for the master to lower SPI_MRDY after each SPI frame, but not recommended. Ensure that the SPI_SCLK is deactivated after one SPI frame; this is not shown in the diagrams.

The SPI_SCLK must be active the whole time during one SPI frame size. The SPI frame size is known and fixed. The master and the slave must be able to receive at least one SPI frame. It is not allowed to deactivate the SPI_SCLK when a frame is transmitted.

If the master lowers SPI_MRDY during an SPI frame, it must ensure that the ongoing transfer can be correctly terminated. This means the SPI_SCLK should still be running and outputting data. This behavior is not recommended, but does not cause any problems as the slave will not check the SPI_MRDY signal during an ongoing transfer.

The SPI_MRDY signal going low does not correspond to a multi-frame transfer end. A multi-frame transfer end is signaled by the MORE flag. The transfer ends when this flag is not set, meaning that it is the last frame.

4.2 Slave initiated transfer to the Master

Requirements: the interrupt is able to wake up the master on SPI_SRDY and the slave on SPI_MRDY signal

Starting from the state where the master is in sleep mode, the following actions will happen:

- The slave indicates to the master that it is ready send data by activating SPI_SRDY
- When the master is ready to send, it signalizes this by activating SPI_MRDY
- The master will activate the clock and the two processors will exchange the communication header and data

Figure 9: slave initiated data transfer
• If the data has been exchanged, the slave will deactivate SPI_SRDY to process the received information. The master does not need to deactivate SPI_MRDY as it controls the SPI_SCLK. After preparation, the slave again activates SPI_SRDY and waits for SPI_SCLK activation. When the clock is active, all the data will be transferred without intervention. If more data is to transferred (flag set in any of the headers) the process will repeat from step 3.

### 4.3 Master initiates transfer with sleeping Slave

**Requirements:** the interrupt is able to wake the slave from sleep mode on SPI_MRDY.

- The master wakes the slave by setting the SPI_MRDY line active.
- As soon as the slave is awake it will signal it by activating SPI_SRDY.
- The master will activate the clock and the two processors will exchange the communication header and data.
- If the data has been exchanged, the slave will deactivate SPI_SRDY to process the received information. The master does not need to deactivate SPI_MRDY as it controls the SPI_SCLK.
- After the preparation, the slave will reactivate SPI_SRDY and wait for SPI_SCLK activation. When the clock is active, all the data will be transferred without intervention. If there is more data to transfer (flag set in any of the headers), the process will repeat from step 3.

### 4.4 Frame end

- The master wakes the slave by setting the SPI_MRDY line active.
- As soon as the slave is awake it will signal it by activating SPI_SRDY.
- The master will activate the clock and the two processors will exchange the communication header and data.
- If the data has been exchanged, the slave will deactivate SPI_SRDY to process the received information. The master does not need to deactivate SPI_MRDY as it controls the SPI_SCLK.
- After the preparation, the slave will reactivate SPI_SRDY and wait for SPI_SCLK activation. When the clock is active, all the data will be transferred without intervention. If there is more data to transfer (flag set in any of the headers), the process will repeat from step 3.

*Figure 10: Application processor initiated data transfer*

*Figure 11: Transfer Termination*
In case of the last transfer (see protocol), the master will lower its SPI_MRDY. After the data-transfer is finished the line must be low. If the slave has already set its SPI_SRDY line the master must raise its line to initiate the next transfer (slave-waking-procedure).

If the data have been exchanged, the slave will deactivate SPI_SRDY to process the received information. This is the normal behavior.

- After the last data-transfer is finished (detected by the end of the clock) the next transfer can be started by generating an activating interrupt on the SPI_MRDY or the SPI_SRDY line.

![Figure 12: Transfer Termination – Slave restarts transmission](image)

If the data have been exchanged, the slave will deactivate SPI_SRDY to process the received information. This is the normal behavior.

The slave will indicate to the master that it is ready to send data by activating SPI_SRDY.

When the master is ready to send, it signalizes this by activating SPI_MRDY – Optional when SPI_MRDY is low before.

In the example above, the slave indicates immediately after a transfer termination that it is ready to start transmission again. In this case the slave will raise SDRY again, and the SPI_MRDY line of the master can be either high or low at that moment. It must only ensure that the SPI_SRDY change will be detected correctly (usually via interrupt).
5 Timing

The minimum time before starting the next data transfer is defined as shown in Figure 13 for SPI_MRDY \( (t_{m\_trans}) \) and in Figure 14 for SPI_SRDY \( (t_{s\_trans}) \). Table 4 reports the minimum allowed values.

![Figure 13: SPI_MRDY transition](image)

LISA-U1/LISA-U2 series modules are able to set the SPI_SRDY line to active within a maximum time \( (t_{s\_res}) \) once the master sets the SPI_MRDY line to active, as shown in Figure 15. Table 4 reports the maximum values.

![Figure 14: SPI_SRDY transition](image)

![Figure 15: SPI_SRDY response](image)
6 Troubleshooting

The following sections describe the possible error scenarios for an unsuccessful SPI transfer. An additional dedicated error detection mechanism should be added to the system solution, otherwise the user may run into synchronization issues in the higher layer protocols.

6.1 Fast change of RDY lines could not be detected

If the MORE flags are 0, but in the meantime new data arrives at one side, it will raise its RDY (S-/M-)RDY line again. The time between setting the line low and high again might be too short to detect the interrupt; therefore a minimum time for detecting the interrupt must be defined. See the minimum requirements in section 8.3. If the time constraints are not respected, a break in the protocol occurs (the SPI communication no longer works).

6.2 Slave reboot outside SPI transfer

When the slave boots up it will check the SPI_MRDY line and start the next transfer immediately if the SPI_MRDY line is active.

6.3 Slave reboot during SPI transfer

Invalid data received by the master; impact not known; for the next wanted transfers the slave will not react on SPI_MRDY according to protocol and the master can detect a slave error. If the slave boots up it will check the SPI_MRDY line and start the next transfer immediately if the SPI_MRDY line is active.

6.4 Master reboot outside SPI transfer

Depending on the settings, the slave may wait infinitely for the raise of SPI_MRDY or discard the frame that it tries to transfer (see also section 6.6). As there might be some higher protocol running that cannot be easily resumed after a master reboot, it is recommended that the application on the master side reset the slave, so both sides will re-start in a defined state.

6.5 Master reboot during SPI transfer

This will lead to a break of the SPI clock. Depending on the settings, the slave may wait infinitely to finish the SPI transfer or discard this SPI frame, see section 6.6. As typically a higher protocol is running that cannot be easily resumed after a master reboot, it is recommended that the application on the master side reset the slave, so both sides will re-start in a defined state.

6.6 Recovery mechanism for slave

As the transfer is under the full control of the master and is based on negotiated frame sizes, the slave shall be enabled to detect a transmission problem. The slave shall cleanly end the ongoing transfer and resume normal operation afterwards.

Possible problems at the master side for example:

- Master transmits less bytes than negotiated
- Master reboots during the transfer
As a result the SPI_SCLK would stop and the slave will not be able to finish its transfer. The slave will stay in the transfer state for an indefinite time.

Slave state after a premature SPI_SCLK stop:
- SPI_SRDY will stay high as transfer is not finished (this means implicitly an SPI_MRDY interrupt will not be answered with raising SPI_SRDY)
- Tx and Rx data for the broken transfer will not be handled at Slave side

The slave is not able to finish the transfer and will not react to subsequent master initiated transfer attempts. As a consequence, the SPI communication will stop as the handshaking mechanism via SPI_MRDY/SPI_SRDY interrupts no longer works.

With a timeout for SPI_SCLK breaks, the slave is capable of detecting a transmission problem and starting the recovery mechanism. The timeout value shall be chosen according to the master SPI_SCLK timing (a defined value cannot be given here).

**Figure 16: Master transmits less bytes**

**Figure 17: Master reboot during transfer**

**Actions to be performed by the slave**
- Leave transfer state (SPI_SRDY shall be lowered)

For the slave Tx data, the master has the following condition to detect a problem:
- Current Data size is bigger than the actual received bytes
7 Protocol examples

7.1 Communication example
The SPI devices are used as an RS232 device. The terminal is the master and the module is the slave.

7.1.1 How to fill the header and data buffer - Example
After powering up the terminal and the module, the terminal starts the communication, enabling the error code by sending AT+CMEE=2; the module answers OK:
The terminal raises SPI_MRDY. If the module raises SPI_SRDY it sends this header in bits:

```
0 0 0 0 11111111100 (Next data size = 2044)
0 0 0 0 0000001011 (Current data size = 11)
```

And this payload in bytes:
```
attcme=2\r\n2033 following bytes
```

While sending, it receives from the module:
```
0 0 0 0 11111111100 (Next data size = 2044)
0 0 0 0 00000000000 (Current data size = 0)
```

With this payload:
```
2044 following bytes
```

The following bytes can have any values.

The module raises SPI_SRDY. Once the terminal raises SPI_MRDY and it starts providing the SPI clock, the module sends this header in bits:

```
0 0 0 0 11111111100 (Next data size = 2044)
0 0 0 0 00000000110 (Current data size = 6)
```

With the payload
```
\r\nOK\r\n2038 following bytes
```

While sending it receives from the terminal:
```
0 0 0 0 11111111100 (Next data size = 2044)
0 0 0 0 00000000000 (Current data size = 0)
```

With the payload in bytes:
```
2044 following bytes
```

The following bytes can have any values.
7.1.2 Multi-frame transfer example

This section shows how a data transfer could look. It describes the buffer sizes and shows how the setting of the RTS / CTS flags stops the transfer.

The master raises SPI_MRDY line and sends a request to the slave. After the slave wakes up the transferred data looks like this:

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>11</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>0</td>
</tr>
</tbody>
</table>

The data transfer is finished.

The slave gets result from the network (5206 bytes) and raises the SPI_SRDY line to start the first transfer. The transferred data looks like this:

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>0</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>1</td>
<td>2044</td>
<td>2044</td>
</tr>
</tbody>
</table>

The slave raises the SPI_SRDY line to start the next transfer. The transferred data looks like this:

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>1</td>
<td>0</td>
<td>2044</td>
<td>0</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>1</td>
<td>2044</td>
<td>2044</td>
</tr>
</tbody>
</table>

The master has a problem with the buffer size and sets the RTS-flag – this should occur very rarely. This means the transfer is finished until the next transfer. Both are back to Inactive Mode.

The master gets space in its buffers and raises the SPI_MRDY line to start the next transfer (the buffer size of this transfer is the starting buffer-size DEF_BUF_SIZE):

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>0</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>1</td>
<td>2044</td>
<td>0</td>
</tr>
</tbody>
</table>

No data is sent from slave to the master because the master has raised the RTS in 3.

The slave raises the SPI_SRDY line to start the next transfer. In the meantime the master gets 2602 bytes of new data and passes the first 2044 with this transfer to the slave, there are still 558 left over. The slave also gets 16 new bytes to transfer. The transferred data looks like this:

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td>1</td>
<td>2044</td>
<td>2044</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>1134</td>
</tr>
</tbody>
</table>

If both are finished and have nothing more to send, both more-flags are 0. The transfer has stopped and the next data-size will be DEF_BUF_SIZE.

<table>
<thead>
<tr>
<th>Transfer</th>
<th>RTS/CTS</th>
<th>MORE</th>
<th>Next_Size</th>
<th>Current Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>558</td>
</tr>
<tr>
<td>Slave</td>
<td>0</td>
<td>0</td>
<td>2044</td>
<td>0</td>
</tr>
</tbody>
</table>

The transfer is finished.
7.2 Flow control

7.2.1 Data transmission stopped

*M: Flow control can also be removed by sending an empty frame

Figure 18: Flow control – Data transmission stopped
7.2.2 Data transmission continues

Figure 19: Flow control – Data transmission continues
7.2.3 Both sides signal flow control

Figure 20: Flow control – both sides signaling
7.3 MORE (transfer) condition

MORE condition is TRUE when:

- The master and slave have set the MORE flag and no flow control is signaled or only one side has signaled flow control
- The master has set the MORE flag, the slave has not and neither signaled flow control
- The slave has set the MORE flag, the master has not and neither signaled flow control
- The master and/or slave must send an error indication

MORE condition is FALSE when:

- The master and slave have not set the MORE flag
- The master has set the MORE flag, the slave has not and the slave has signaled flow control
- The slave has set the MORE flag, the master has not and the master has signaled flow control
- The master and slave have set the MORE flag and both signaled flow control
8 Implementation comments

The LISA-U1/LISA-U2 series supports the specified protocol in slave mode. It is up to the customer to implement the master side and use it to communicate with LISA-U1/LISA-U2 series.

8.1 SPI clock rates on LISA-U1/LISA-U2 series

The external master defines the baud rate. The achievable baud rates depend strongly on chip, pad and board design. LISA-U1/LISA-U2 series slave mode is running with up to 26 MHz SPI master clock.

8.2 Estimation of available bandwidth

The analysis of bandwidth usage will concentrate on the case when both processors are active. This is the usual situation when, for instance, an HSDPA download happens. This use case puts a high demand on the bandwidth.

8.2.1 General considerations

Interrupt Load while Transferring Data:

The MORE and flow-control-flags (RTS, CTS) inform the receiver whether the next data-transfer is required. This prevents the useless transfer of data.

At each transfer the following interrupts will be received:

- The master receives two interrupts: SPI_SRDY, and the Transfer Finished Interrupt TF (indicating that data has been received and sent)
- The slave receives only the Transfer Finished Interrupt TF

DMA Load:

SPI transfers the data in both directions at the same time. Therefore 2 DMAs are needed.

The DMA can be configured according to the transfer-size of each transfer. It might make sense to store the header to another part of the memory, by using the linked list feature LLI of the ARM-DMA. This causes the DMA configuration to depend on the ARM.

This means the data size must be a multiple of the header size, to get the optimal DMA-usage.

The DMAs for SPI transfer shall get the highest priority in the system to achieve maximum throughput.

Latency:

The bigger the packet size, the higher the latency. The best compromise between achievable bandwidth and latency must be found.

8.2.2 Dependency on different parts of SPI transmission

The net bandwidth of the SPI protocol is not only affected by the SPI clock used, but also strongly dependent on the times needed from start of SPI transmission (SPI_SRDY/SPI_MRDY) until finally applying the SPI clock. Find below the basic times contributing to the overall net bandwidth and a formula which can be used to evaluate the net throughput based on the different inputs.
The net throughput can be calculated as follows:

\[ n[\text{Mbps}] = f \cdot \text{jitter} \cdot \left( p \div (p + h) \right) \cdot \left( ( (p + h) \cdot 8 \div f ) \div ( (p + h) \cdot 8 \div f + t1 + t2 + t3 + t4 ) \right) \]

with

\[ f = \text{SPI clock}[\text{MHz}], p = \text{payload}[\text{Bytes}], h = \text{SPI header}[\text{Bytes}], t1 - t4[\text{us}] \]

Equation 2: Net throughput calculation

Ideally it is assumed that the SPI clock is a continuous function during an SPI packet transfer. However, it is possible that there are small pauses between some clock cycles due to the master needing to setup the SPI HW more than once for an SPI packet. The jitter is a numeric value between 0-1 to account for this.

Further reductions of the user data throughput due to higher layer protocols inside the payload are not considered here. The equation can also be used for slave initiated and multi-packet transfer, it is important to use the same reference points.
8.3 Settings for LISA-U1/LISA-U2 series driver

Table 4 describes the current settings of the SPI implementation for LISA-U1/LISA-U2 series.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role in SPI protocol</td>
<td>Slave</td>
</tr>
<tr>
<td>SPI_SCL frequency ( f_{master_clock} )</td>
<td>26 MHz</td>
</tr>
<tr>
<td>SPI_SCLK idle state</td>
<td>Low ( \text{CPOL}=0 )</td>
</tr>
<tr>
<td>SPI_MRDY idle state</td>
<td>Low</td>
</tr>
<tr>
<td>SPI_SRDY idle state</td>
<td>Low</td>
</tr>
<tr>
<td>SPI_MISO idle state</td>
<td>High</td>
</tr>
<tr>
<td>SPI_MOSI idle state</td>
<td>High</td>
</tr>
<tr>
<td>Shift Data</td>
<td>On rising clock edge ( \text{CPHA}=1 )</td>
</tr>
<tr>
<td>Latch Data</td>
<td>On falling clock edge ( \text{CPHA}=1 )</td>
</tr>
<tr>
<td>Byte Endian-ness</td>
<td>LSB (Least Significant Byte) is transmitted first</td>
</tr>
<tr>
<td>Bit Endian-ness</td>
<td>msb (most significant bit) is transmitted first</td>
</tr>
<tr>
<td>HEADER_SIZE</td>
<td>4</td>
</tr>
<tr>
<td>DEF_BUF_SIZE</td>
<td>2044</td>
</tr>
<tr>
<td>MAX_BUF_SIZE</td>
<td>2044</td>
</tr>
<tr>
<td>SPI_MRDY transition ( t_{	ext{trans}} ): minimum time, lower values are not allowed</td>
<td>80 ns min, if LISA is in active mode (power saving disabled) 62 µs min, if LISA is in idle mode (power saving enabled)</td>
</tr>
<tr>
<td>SPI_SRDY transition ( t_{	ext{trans}} ): minimum time, lower values are not allowed</td>
<td>80 ns min</td>
</tr>
<tr>
<td>SPI_SRDY response ( t_{	ext{res}} )</td>
<td>200 µs max, if LISA is in active mode (power saving disabled) 10 ms max, if LISA is in idle mode (power saving enabled)</td>
</tr>
</tbody>
</table>

For more details on SPI timings, refer to LISA-U1 Series Data Sheet [1] and LISA-U2 Series Data Sheet [2].
9 Additional notes

9.1 State machines for implementation

This section describes the state machines. "Continue" is the corresponding evaluation result whether and how the communication needs to be continued, see section 3.7 for details.

9.1.1 Master (Application Processor)

![State machine diagram]

Figure 21: Master-side state machine
9.1.2 Slave (LISA-U1/LISA-U2 series module)

![State Machine Diagram]

The slave (LISA-U1/LISA-U2 series module) does not know if a master is connected and available for the communication. To prevent the communication getting stuck when the Master is missing, the TX Request (in Figure 22) is generated only after the Master detection, that is after the first transmission initiated by Master.

As a general rule, the Master must send an AT command at the first use of the SPI interface (after boot), to allow subsequent data transmission initiated by the slave (e.g. Unsolicited Result Codes). Otherwise, the data is buffered in the slave (e.g. Unsolicited Result Codes), and it is sent to the master only when the master initiates the first transfer.
9.2 Pseudo code

As an example, here is the pseudo code of a reference master SPI device firmware:

Receive data:
  decode incoming packet (from the SPI hw):

  read the header (4 bytes) from the lower level driver

  if the received header is dummy (ffffffff), then
    set all header to 0
  if outgoing CTS is 0 and incoming MORE is 1, or incoming CTS is 0 and outgoing MORE is 1, then
    set flag that the SPI-Transfer can/should be started

  if the right amount of data (not less than a complete SPI buffer) has been received, then
    set flag that the SPI-Transfer can/should be started

  if the header says the payload is non-null, then
    read it from the lower level driver

  in case:
    - keep counting the lost bytes
    - Mark the copied HW buffer as free for reuse

Send data:
  encode outgoing packet (from the SIO buffer to the SPI hw):

  if previous incoming CTS was 0 then
    read the data to be sent (from upper layer buffer);
  else
    No data to transmit, so do not move pointer inside SIO buffer

  evaluate and set (in the outgoing SPI header) the current data size

  if data to be sent does not fit a single SPI frame, then
    set the "next data size" and the "more" bit in the outgoing SPI header.

  if no data to be sent, then
    set all the outgoing packet to FF, return.
  else
    copy the proper amount of data from the SIO buffer to the outgoing packet;
    clean the unused part of the frame.

  enable sending of the data, properly clearing/resetting any flags/semaphores.
10 SPI debug command +USPITRACE

The command activates the SPI trace (debug) capabilities, that are by default disabled. When active, the SPI trace is part of the Primary Log data [3].

The command is used only for testing / debugging purpose.

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Response</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>AT+USPITRACE=&lt;trace_mode&gt;[,&lt;output_size&gt;]</td>
<td>OK</td>
<td>AT+USPITRACE=1,2048 OK</td>
</tr>
<tr>
<td>Read</td>
<td>AT+USPITRACE?</td>
<td>+USPITRACE: &lt;trace_mode&gt;,&lt;output_size&gt; OK</td>
<td>+USPITRACE: 0,2044 OK</td>
</tr>
<tr>
<td>Test</td>
<td>AT+USPITRACE=?</td>
<td>+USPITRACE=&lt;trace_mode&gt;: (list of supported &lt;trace_mode&gt;), &lt;output-size&gt;: (list of supported &lt;output_size&gt;) OK</td>
<td>+USPITRACE=&lt;trace_mode&gt;: (0-3), &lt;output-size&gt;: (1-2048) OK</td>
</tr>
</tbody>
</table>

10.1 Defined Values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| <trace_mode> | Number | Trace level to set
  - 0: SPI debug off
  - 1: SPI debug on, level 1 (SPI uplink/downlink data dump)
  - 2: SPI debug on, level 2 (SPI uplink/downlink data dump plus SPI flow control trace)
  - 3: SPI debug on, level 3 (only SPI flow control trace) |
| <output_size> | Number | Maximum size of a uplink/downlink data block that can be dumped in a single read/write operation
  - 0-2048: supported size range, 2044 is the default value (when the parameter is omitted) |

The SPI uplink/downlink data dump (<trace_mode>= 1 or 2) is present in the Primary Log data in the form of an SDL message trace for the process sio_glue; the state is GLUE_S_ACT and the events GLUE_E_UL and GLUE_E_DL.

The SPI flow control trace (<trace_mode>= 2 or 3) provides support in the SPI protocol analysis in reference to the flow control event; it is present in the Primary Log data in the form of ASCII strings, starting with the substring: “[SPI FC]: timestamp is: “. The timestamp unit is 1/26 µs.
Related documents

[1] u-blox LISA-U1 series Data Sheet, Docu No UBX-13002048
[2] u-blox LISA-U2 series Data Sheet, Docu No UBX-13001734

All these documents are available on our homepage (http://www.u-blox.com).

For regular updates to u-blox documentation and to receive product change notifications, register on our homepage.

Revision history

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Name</th>
<th>Status / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Apr. 08, 2011</td>
<td>mtom</td>
<td>Initial release</td>
</tr>
<tr>
<td>1</td>
<td>May 26, 2011</td>
<td>mtom</td>
<td>Improved the SPI and Byte ordering description</td>
</tr>
<tr>
<td>2</td>
<td>Jan. 08, 2012</td>
<td>lpah</td>
<td>Extended the document applicability to LISA-U2 series and removed LISA-H1 series</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Added the pseudo code of a reference master SPI device firmware</td>
</tr>
<tr>
<td>3</td>
<td>Jul. 18, 2012</td>
<td>mben</td>
<td>Added +USRITRACE debug command description</td>
</tr>
<tr>
<td>3</td>
<td>Aug. 23, 2012</td>
<td>mtom</td>
<td>Better representation of bits in section 7.1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Last revision with docu number 3G.G2-CS-11000)</td>
</tr>
<tr>
<td>A</td>
<td>Aug 27, 2013</td>
<td>mben / lpah</td>
<td>Improved the section 9.1.2 about master detection</td>
</tr>
</tbody>
</table>
Contact

For complete contact information visit us at www.u-blox.com

u-blox Offices

North, Central and South America
u-blox America, Inc.
Phone: +1 703 483 3180
E-mail: info_us@u-blox.com
Regional Office West Coast:
Phone: +1 408 573 3640
E-mail: info_us@u-blox.com
Technical Support:
Phone: +1 703 483 3185
E-mail: support_us@u-blox.com

Asia, Australia, Pacific
Asia, Australia, Pacific
u-blox Singapore Pte. Ltd.
Phone: +65 6734 3811
E-mail: info_ap@u-blox.com
Support: support_ap@u-blox.com
Regional Office Australia:
Phone: +61 2 8448 2016
E-mail: info_anz@u-blox.com
Support: support_ap@u-blox.com
Regional Office China (Beijing):
Phone: +86 10 68 133 545
E-mail: info_cn@u-blox.com
Support: support_cn@u-blox.com
Regional Office China (Shenzhen):
Phone: +86 755 8627 1083
E-mail: info_cn@u-blox.com
Support: support_cn@u-blox.com
Regional Office India:
Phone: +91 959 1302 450
E-mail: info_in@u-blox.com
Support: support_in@u-blox.com
Regional Office Japan:
Phone: +81 3 5775 3850
E-mail: info_jp@u-blox.com
Support: support_jp@u-blox.com
Regional Office Korea:
Phone: +82 2 542 0861
E-mail: info_kr@u-blox.com
Support: support_kr@u-blox.com
Regional Office Taiwan:
Phone: +886 2 2657 1090
E-mail: info_tw@u-blox.com
Support: support_tw@u-blox.com

Headquarters
Europe, Middle East, Africa
u-blox AG
Phone: +41 44 722 74 44
E-mail: info@u-blox.com
Support: support@u-blox.com