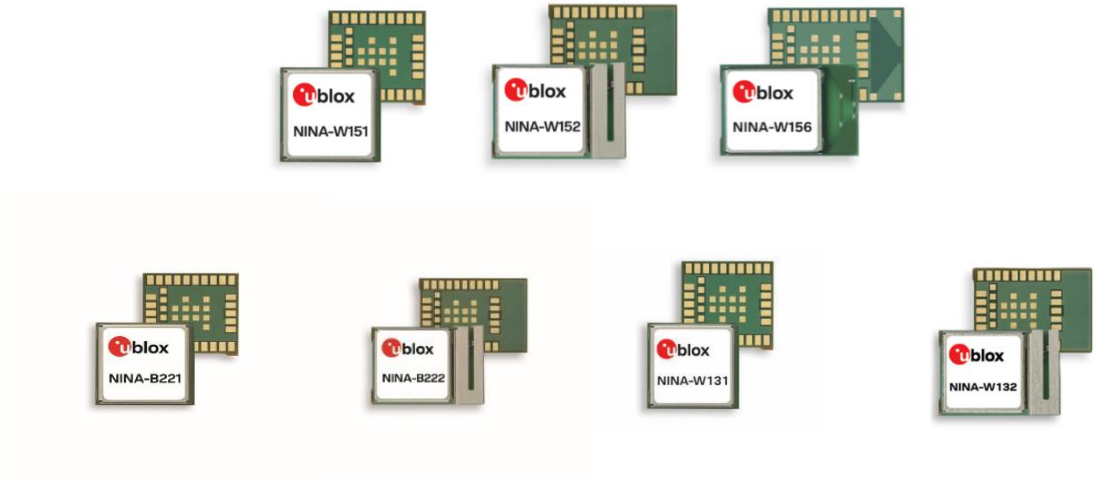
S

u-connectXpress

SPI peripheral

Protocol specification



Abstract

This document describes how to interact over an SPI bus with a u-blox module running the u-connectXpress software. The optional protocol on top of the SPI bus that handles the data flow and several additions to the hardware interface are also described.

Document information

Title	**u-connectXpress**	
Subtitle	SPI peripheral	
Document type	Protocol specification	
Document number	UBX-20028725	
Revision and date	R03	20-Jun-2022
Disclosure restriction	C1 - Public	

This document applies to the following products:

Product name	Firmware version
NINA-B221	3.0.0 or later
NINA-B222	3.0.0 or later
NINA-W131	3.0.0 or later
NINA-W132	3.0.0 or later
NINA-W151	3.0.0 or later
NINA-W152	3.0.0 or later
NINA-W156	3.1.0 or later

Contents

1 Introduction

This document describes how to interact over Serial Peripheral Interface (SPI) with a u-blox module running the u-connectXpress software.

There is a simple protocol, “u-connectXpress SPI control protocol”, hereafter called “control protocol”, defined to provide stable SPI communication between a u-blox module and an external host processor.

This SPI bus can be configured to optionally use the control protocol and can be configured to use different SPI modes. Also, the pins used for the different SPI signals are configurable. For information describing the pin considerations when SPI is the only interface, see also Activating SPI without using AT commands.

This document also describes the startup procedure in case the module only has SPI as the single interface for interacting with the host.

The SPI bus data can be sent and received seamlessly over other interfaces (Wi-Fi, Bluetooth, UART, and so on) using the unique u-blox stream concept. For more information about the u-blox stream concept, see also the u-connectXpress user guide [7].

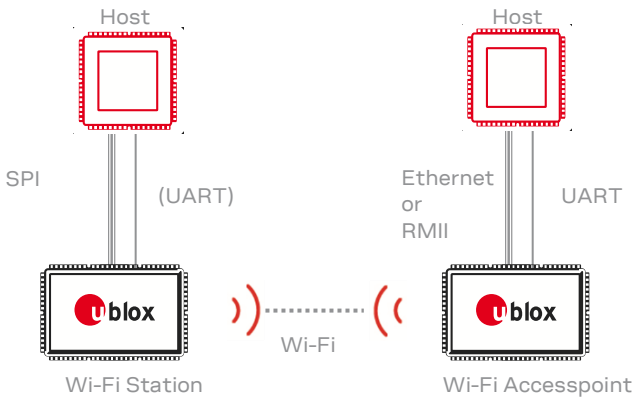


Figure 1 Setup example showing a bridge configuration used to relay data from SPI over Wi-Fi

2 Hardware interface

The u-blox modules introduce some extra signals compared to the standard SPI bus, as seen in Figure 2.

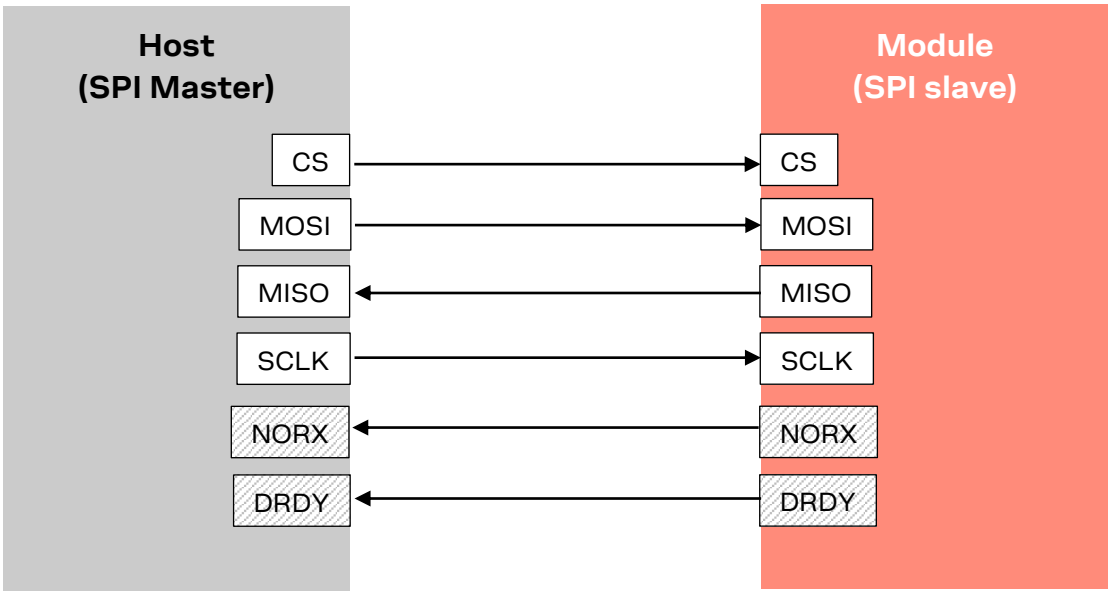


Figure 2 SPI signals. Optional signals dashed

The extra hardware signals are used as follows:

- **DRDY** (optional): Module output, signals that module has data available. If the control protocol (see u-connectXpress SPI control protocol) is enabled from module to host, the host can poll the module as an alternative to using this HW signal.
- **NORX** (optional): Module output, signals that the module cannot receive data right now. If the control protocol is enabled from module to host, the host can read the NORX status from packets received from the module. This provides an alternative to using this hardware signal. This signal has no pre-defined pin and can only be used when the SPI is enabled using an AT-command. See also u-connectXpress SPI control protocol.

For further information about how the signals can be configured, see the u-connectXpress AT commands manual [2].

3 u-connectXpress SPI control protocol

3.1 Introduction

The issues addressed by this protocol are:

- SPI is full duplex. If the clock is ticking, data is always sent in both directions. On either side, there is no way of knowing whether some or all received data is valid or should be ignored.
- In simplex mode, the slave can assume that the master will not send more data than necessary. However, if the master uses full duplex, and it wants to read more than it needs to send, it must send filler bytes at the end of the transaction to keep the clock running.

3.2 Protocol overview

A packet consists of a packet header of four extra bytes in the beginning followed by the payload.

The first two bytes of the packet header serves as a 16-bit preamble. The last two bytes of the header is packet information.

A Chip Select (CS) assertion is considered the start of a “packet”.

The packet format is slightly different depending on which direction data is sent, as described in Host to module and Module to host.

3.3 Host to module

Header 0	Header 1	Header 2	Header 3	Payload 0	Payload 1		Payload N-1
		Bit 7-0	Bits 7-0				
0xBA	0x15	Length Hi	Length Lo	*Data*	*Data*	...	*Data*

Table 1: SPI control protocol packet in direction host to module

The packet format is in the host to module direction similar to the Module to Host protocol, but without the **NORX** bit, as shown in Table 1.

☞ The preamble is the bytes 0xBA and 0x15 in the headers two first bytes. Any packet with an invalid header will be discarded by the module.

☞ If the full transaction is less than four bytes, the module ignores the packet.

☞ If the length is zero, the module ignores the packet.

☞ If the header specifies a length bigger than given maximum SPI transaction, the module ignores the packet. Maximum SPI transaction size is set when the SPI stream is configured, see command **AT+UDCP** in the u-connectXpress AT commands manual [2] and Appendix E.

☞ If the header specifies a length bigger than the actual transaction, but below maximum SPI transaction length, all following payload is interpreted as valid data.

3.4 Module to host

Header 0	Header 1	Header 2		Header 3	Payload 0	Payload 1		Payload N-1
		Bit 7	Bits 6-0	Bits 7-0				
0xBA	0x15	NORX	Length Hi	Length Lo	*Data*	*Data*	...	*Data*

Table 2: SPI control protocol packet in direction module to host

The packet format in the direction module to host is described in Table 2.

- The preamble is the bytes 0xBA and 0x15 in the headers two first bytes. Any packet with invalid header must be discarded.
- In this case the whole SPI transaction is invalidated, so in a full duplex communication also the host to module packet is discarded and should be resent.
- **NORX** is set if the module currently cannot accept more data from the host.
- The length fields indicate number off payload bytes sent from module.
- The host should ignore any data in a packet where the length fields are 0 but can assume that the **NORX** bit is valid.

3.5 SPI master polling module for data

As described in Hardware interface, pins **DRDY** and **NORX** can be replaced by polling the module – even though this approach is not optimal. Because of the packet lag, two consecutive packets with same state must be read before any conclusion can be made.

- Replacing **DRDY** pin: to make sure slave has no data available, the master must read two packets with zero length directly after each other. Only then is it safe to assume the slave has no data to send.
- Replacing **NORX** pin: to make sure slave can accept data, the master must read two packets with NORX=0 directly after each other. Only then is it safe to assume the slave can receive data.

4 Activating SPI without using AT commands

When the host is connected to the module over the SPI, only the module detects this during the startup procedure and is then ready to accept AT commands over the SPI interface.

The procedure that needs to be followed by the host to indicate to the module that it should enable AT commands over SPI is as follows:

1. During startup, the module toggles **DRDY** to indicate to the host that it is awake and polling the SPI interface.
2. Having detected toggling on the **DRDY** signal, the host asserts the SPI **CS** and enables the SPI clock signal, **SCLK**.
3. The module detects the SPI **SCLK** and enables the SPI interface. When the host has received `+STARTUP` from the module, it is then ready to accept AT commands over the SPI bus.

AT Commands can now be sent over the SPI interface to set up the connections and configure the SPI bus.

☞ The startup mode for the SPI bus is SPI mode 3 with the control protocol enabled. MTU size is set to 768 bytes.

☞ In case SPI is the only interface it is not possible to configure the pins used. In these instances, the NORX pin cannot be used. See the appropriate data sheet [3] [5] [6] for information about which pins to use.

☞ Once an AT command has been received over UART, it is no longer possible to activate the SPI using this method. In that case, SPI must be activated using AT commands over the UART.

☞ If SPI is activated via the hardware pin, the module does not accept AT commands sent over the UART interface (if present).

Appendix

A Glossary

Abbreviation	Definition
CPU	Central Processing Unit
CS	Chip Select
MISO	Mast Input Slave Output
MOSI	Master Output Slave Input
SCLK	Serial Clock Signal
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter

Table 3: Explanation of the abbreviations and terms used

B Example AT command

```
AT+UDCP="spi://spi0/?cs=32&sclk=31&miso=36&mosi=35&mode=3&drdy=25&size=720&proto=3
```

This AT command will set up the HW pins used for the SPI interface and configure an SPI stream

- SPI PDU size of 720 bytes.
- Enable the use of the control protocol in both directions (host -> module, module -> host).

For more information about the command parameters, see the u-connectXpress AT commands manual [2]. To find out how to use this stream in a bridge configuration, see also the u-connectXpress user guide [7]. For pin information, see the respective data sheet [3] [5] [6] .

C Control protocol packet examples

If the module has 260 bytes available to the host, but the host only clocks 10 bytes before deasserting **CS**, the packet from module to host will look like that shown in Table 4.

Header[0]	Header[1]	Header[2]	Header[3]	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
0xBA	*0x15*	*0*	*0x01*	*0x04*	0x12	0x34	0x56	0x78	0x9A	0xBC

Table 4: 6-byte data packet

When the master starts the next transaction, the data received by the master will look like that shown in Table 5 – with the length field reduced by the 6 bytes already transmitted.

Header[0]	Header[1]	Header[2]	Header[3]	Data[0]	Data[1]	...	Data[253]
0xBA	*0x15*	*0*	*0x00*	*0xFE*	0xDE	0xF0	0x...	0xAC

Table 5: Data packet with remaining data

D Example code

D.1 SPI header

An example of how a host could handle the SPI header in a transaction from the module is given below.

```
bool handle_spi_transaction(uint8_t *buf, uint16_t transaction_len) {
   if (transaction_len < 4) {
      return false; // not even header data, ignore
   }

   if (buf[0] != 0xba || buf[1] != 0x15) {
      return false; // typically no slave attention, ignore
   }
   uint16_t header = (buf[2] << 8) | buf[3];
   if (header & 0x8000) {
      no_rx = true; // module cannot accept more data
   }
   else {
      no_rx = false;
   }

   uint16_t packet_len = header & 0x7fff;
   if (packet_len == 0) {
      return; // no payload, ignore
   }
   if (packet_len > spi_mtu - 4) {
      // only accept up to maximum transaction length
      // minus header length
      packet_len = spi_mtu - 4;
   }
   if (packet_len > transaction_len - 4) {
      // there's more data available on module than what
      // is available in this transaction
      packet_len = transaction_len - 4;
   }
   handle_module_payload(&buf[4], packet_len);
   return true;
}
```

D.2 Polling module for data

If the **DRDY** and **NORX** signals are not used the host can poll the module as described in SPI master polling module for data. The recommended way of doing this efficiently is outlined in the following pseudo code algorithm.

☞ Note that the **NORX** signal is read but ignored in this example.

```
THREAD(module_data_poller) {
   uint8_t rx_data[max_transaction_length];
   uint8_t zero_data_before = 0;
   while (1) {
      size_t size_received = read_packet_from_module(rx_data, max_transaction_length);
      if (size_received < 4 || rx_data[0] != 0xba || rx_data[1] != 0x15) {
         // module not responsive, wait a bit and retry
         sleep(snooze_period);
         continue;
      }
      uint16_t avail_data = ((rx_data[2] << 8) | rx_data[3]) & 0x7fff;
      // NORX bit set?
      uint8_t norx = rx_data[2] & 0x80 ? TRUE : FALSE;
      if (avail_data == 0) {
```

```
            if (!zero_data_before) {
               zero_data_before = 1;
               continue; // first zero packet, try again directly
            }
            // had two consecutive zero readings, so safe to sleep
            sleep(poll_sleep_period); // poll on a later occasion
         }
         else {
            zero_data_before = 0;
            uint16_t data_length;
            if (avail_data > max_transaction_length - 4) {
               // have more data on module side than the transaction size
               data_length = max_transaction_length - 4;
            }
            else {
               data_length = avail_data;
            }
            handle_data_from_module(&rx_data[4], data_length);
         }
      }
   }
```

E Special considerations for modules based on ESP32 chipsets

☞ NINA-W15x, NINA-W13x and NINA-B2 modules are based on ESP32 chipsets from Espressif, and there are some considerations to consider for these modules:

- Always send at least 8 bytes.
- Number of transmitted bytes must be divisible by 4.
- Maximum number of transmitted bytes in one transaction is 4096.
- The last four sent bytes always become corrupt on the slave side, so the master must always transmit four extra dummy bytes.

Related documents

[1] NINA-W1 series, system integration manual, UBX-17005730
[2] u-connectXpress AT commands manual, UBX-14044127
[3] NINA-W15 data sheet, UBX-18006647
[4] NINA-B2 system integration manual, UBX-18011096
[5] NINA-B2 data sheet, UBX-18006649
[6] NINA-W13 series, data sheet, UBX-17006694
[7] u-connectXpress user guide, UBX-16024251
[8] EVK NINA-W1/EVK NINA-B2 user guide, UBX-17011007

☞ For product change notifications and regular updates of u-blox documentation, register on our website, www.u-blox.com.

Revision history

Revision	Date	Name	Comments
R01	03-Jul-2020	mape	Initial release.
R02	04-Feb-2021	flun	Clarified NORX. NINA-W156 requires u-connectXpress 3.1.0 onwards. Renamed document.
R03	20-Jun-2022	mape	Corrected number of clock cycles need for SPI detection in Activating SPI without using AT commands. Updated contact information. Minor editorial changes.

Contact

For further support and contact information, visit us at www.u-blox.com/support.