By Michael Momayezi
Phone 512-553-9933
Fax 512-553-9934
email momayezi@bridgeportinstruments.com

Bridgeport Instruments, LLC, 11740 Jollyville Rd., Ste 500, Austin, TX 78759,
www.BridgeportInstruments.com

Common USART Interface

May, 2020

Summary

All BPI product that use an embedded ARM processor share a common serial interface.

- Small and embedded computers enjoy easy access to the MCA without complexity and power consumption of a USB interface.
- Using commercial interface hardware, the user can cover long distances using RS485.

1. Brief description

We implement a half-duplex serial interface with software flow control. The implementation is geared towards the most restrictive case of a micro-controller that does not use interrupts. Each transmission is preceded by sending a single ping-byte to make sure the receiving side is ready to receive a command or data. Finally, large data transmissions are cut into 256-byte chunks so that the host processor does not need to implement a big data buffer.

2. Detailed description

2.1 Hardware

In most cases the RX and TX pins are connected directly to the ARM I/O pins. Hence the drive strength is limited to 1mA and they are sensitive to electrostatic discharge. As a result, cables connecting directly to the MCA should be interior and shielded cables. Long cable runs require proper decoupling and conversion to RS485 or other robust standards.

2.2 Speed

In the current implementation, the default baud rate is 115200 Bd. Users can change the device settings to program a different baud rate into non-volatile memory. The safest method to do this is to use the USB interface of the MCA, although it can be achieved via the serial port as well. The maximum speed supported by the embedded ARM is 3MBd.

The standard communication setting is 1 start bit, 8 data bits, 1 stop bit, no parity.

2.3 Protocol

The high level communications protocol is exactly the same as when using the USB interface. When using the serial interface on a small computer such as a Raspberry Pi or an Arduino, the user would launch the data server inside the mds_serial folder rather than the mds_v3 folder. All client software remains unchanged

When the host computer is a micro-controller, the generic C-code examples in the examples folder will prove helpful.

The communication protocol requires that the host first sends a command of 64 bytes, telling the MCA what the next action is going to be. That second action is to either send data to the MCA or demand data from the MCA.

2.4 Protocol

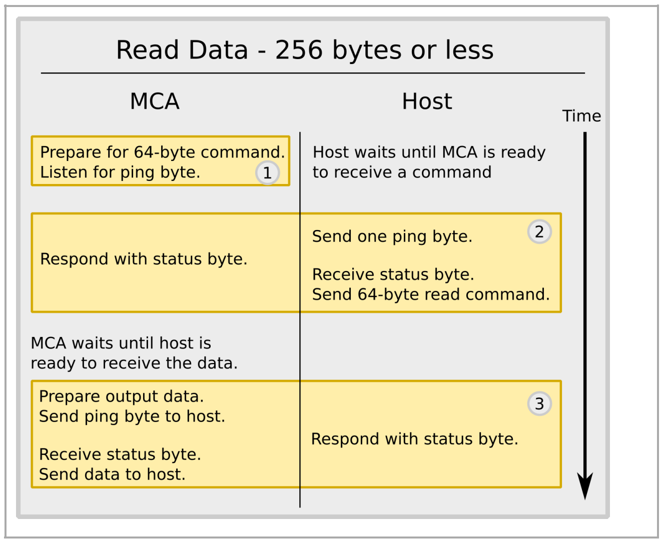
Each command or data transmission is prepared by a single-byte ping. Embedded micro-controllers typically have a 1-byte receive buffer and can raise a flag if a byte has been received. On receipt of a ping-byte, the controller gets ready for the next step, and then sends an answer byte. In this case the next step may be to send a command, or to send or receive data.

The two flow diagrams below show the sequence for a write data and a read data command. Whenever the amount of data to be transferred is larger than 256 bytes, the software has to cut that into chunks of 256 bytes, with the last chunk be shorter, if needed.

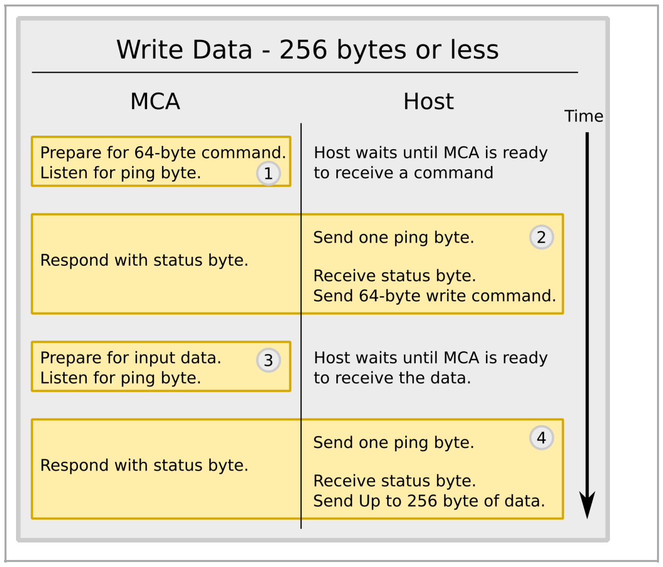
When reading more than 256 bytes of data, box no. 3 will be repeated as often as is necessary. When writing more than 256 bytes of data, box no. 3 and 4 will be repeated as often as is necessary.

2.5 Timeout

Successful data transmission requires that the entire expected sequence of pings, commands and data transmissions is executed. If the host abandons a sequence or if an unrecoverable error occurs, the MCA will reset the USART interface to its idle state, box (1), 5 seconds after receiving the first ping.


Read Data - 256 bytes or less
MCA
Host
Time
Prepare for 64-byte command. Listen for ping byte. 1
Host waits until MCA is ready to receive a command
Respond with status byte.
Send one ping byte. 2
Receive status byte. Send 64-byte read command.
MCA waits until host is ready to receive the data.
Prepare output data. Send ping byte to host.
Receive status byte. Send data to host.
3
Respond with status byte.

Low level protocol for reading up to 256 byte of data from the MCA.


Write Data - 256 bytes or less
MCA
Host
Time
Prepare for 64-byte command. Listen for ping byte. 1
Host waits until MCA is ready to receive a command
Respond with status byte.
Send one ping byte. 2
Receive status byte. Send 64-byte write command.
Prepare for input data. Listen for ping byte. 3
Host waits until MCA is ready to receive the data.
Respond with status byte.
Send one ping byte. 4
Receive status byte. Send Up to 256 byte of data.

Low level protocol for writing up to 256 byte of data from the MCA.

3. Software description

The mds_serial/embedded_c folder contains a very simple API that is suitable for embedded 32-bit micro-controllers. For test and debugging purposes, the serial interface is implemented in a POSIX compliant manner, and it was tested on a Raspberry Pi 3. The software stack consists of three layers:

- mca1k_api.c implements the high-level read, write, and read-modify-write functions, plus a number of convenience functions for easy access to all data in the MCA.
- bpi_device.c implements the driver, including the flow-control protocol.
- Pi3_serial.c has the hardware-dependent serial I/O code.

3.1 mca1k_api.c

This level implements the three main functions a user would use to communicate with the MCA: read, write, and read-modify-write. The user specifies the MCA data of interest, such as arm_ctrl or arm_histo and then calls the appropriate function.

Beyond the three core communications functions, this level provides a number of convenience functions that cover all relevant cases of reading, writing or modifying MCA data arrays.

There is a single data structure, mca_command, that controls the communication. It is defined in mca1k_api.h

Given the limited data stack size in a micro-controller environment, all functions are being passed a pointer an mca_command structure. The functions are written such that user code needs to only provide memory for one such structure.

Finally, the *mca_command* structure does not contain any data arrays. Instead, it only contains pointers to all data arrays. This ensures that users can align the actual data arrays on 4-byte to 16-byte boundaries, or place them in certain memory segments, as the hardware may require; eg for direct memory access.

In all data exchanges, the LSB is transmitted first. Communication between two little-endian processors will always be correct for all data formats, such as uint16_t, uint32_t or float.

3.1.1 mca_write(struct mca_command *mca_cmd)

Write data to a specific MCA data array.

3.1.2 mca_read(struct mca_command *mca_cmd)

Read data from a specific MCA data array.

3.1.3 mca_rmw(struct mca_command *mca_cmd)

Read-modify-write data from a specific MCA data array. First, the function reads the MCA data into either mca_cmd.u_data (for uint32_t) or mca_cmd.f_data (for float), depending on the requested MCA data type. Replacement data are stored in mca_cmd.replace_u_data or mca_cmd.replace_f_data, again depending on whether the MCA data array consists of uint32_t or float data. The global array replace_idx contains the array index in u_data or f_data indicating which original data should be overwritten by the replacement data. After updating the data array, it is written back to the MCA.

Here is an example to change the operating voltage to 34.5V. In the arm_ctrl array, the requested operating voltage is stored at the offset AC_CAL_OV. Hence, replace_idx[0] = AC_CAL_OV; and replace_data[0]=34.5; Then call mca_rmw(mca_cmd).

3.2 bpi_device.c

This intermediate layer implements the communications protocol in just two functions: one to write a block and one to read a block. It implements the flow control, which requires that any transmission of block data, in either direction, must be preceded by a ping.

The sending side first sends a single-byte ping and waits for the receiver to answer back with a single byte. Once the answer has been received, the sending side can be sure that the receiver is ready to receive the expected number of bytes.

3.2.1 bpi_write_buffer(uint32_t *pSrc, uint32_t num_bytes)

Write data, from pSrc to the MCA. If more than 256 bytes need to be transmitted, this functions will send the data in chunks of 256 bytes. For every chunk it obeys the flow control protocol and uses the ping-mechanism. The last chunk may be smaller than 256 bytes.

3.2.2 bpi_read_buffer(uint32_t *pDst, uint32_t num_bytes)

Read data from the MCA and copy into the memory pDST. If more than 256 bytes need to be transmitted, this functions will receive the data in chunks of 256 bytes. For every chunk it obeys the flow control protocol and uses the ping-mechanism. The last chunk may be smaller than 256 bytes.

3.3 Pi3_serial.c

This is the hardware interface to the serial port. It consists of 5 functions to control and operate the serial interface. The provided functions are POSIX compliant and will run a typical Linux computer, with the appropriate choice for the serial port, such as "/dev/serial0". On a micro-controller, these functions have to be rewritten for the interface in use.

3.3.1 bpi_serial_init()

It programs the serial interface to be non-canonic, ie to be used for binary data transfer. It uses 1 start bit, 8 data bits, 1 stop bit, no parity.

By default the baud rate is set to 115200, as encoded by the constant B115200. Possible low-speed baud rate constants are B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400. The high-speed constants are:

B57600, B115200, B230400, B460800, B500000, B576000, B921600, B1000000, B1152000, B1500000, B2000000, B2500000, B3000000, B3500000, B4000000.

MCAs with an Atmel SAML21 can support baud rates between 19200 and 3000000 (3MBd). The baud rate is not negotiable. It is either fixed in the ARM firmware or can be set via the AC_BAUD register (or the "baud" field) in the arm_ctrl data array. Users wanting to use a baud rate different from the 115200Bd default, are advised to use the USB interface to write the baud rate field into non-volatile memory. The USB interface is not affected by this setting, so it is easy to check that the write was successful.

If the baud rate is changed by the host, it will take effect only after a reboot of the MCA. This avoids loss of communication should a regular update of the ARM control registers (arm_ctrl) fail.

3.3.2 bpi_serial_write(uint8_t *p8_Src, uint32_t num_bytes)

This functions takes a pointer to the data buffer and the number of bytes to transmit as its input parameters. The function sends data without delay or hardware/software flow control.

3.3.3 bpi_serial_read(uint8_t *p8_Src, uint32_t num_bytes)

This functions takes a pointer to the data buffer and the number of bytes to read as its input parameters. The serial interface is set up to perform non-blocking reads. Hence the function polls until the requested number of bytes have been received. In its most simple implementation, it polls until all data have arrived, and thus implements a blocking read.

3.3.4 bpi_serial_reset_input_buffer()

This functions empties the input buffer to avoid reading unwanted data. On an embedded micro-controller, it may not be necessary.

3.3.5 bpi_serial_reset_output_buffer()

This functions empties the output buffer to avoid sending unwanted data. On an embedded micro-controller, it may not be necessary.