BRIDGEPORT INSTRUMENTS, LLC	Bridgeport Instruments, LLC 6448 E HWY 290, STE D-103 www.BridgeportInstruments.com

Instrument Software For Morpho-Based MCAs

1 Operating the instrument

The Morpho instruments are programmed for autonomous data acquisition, receiving occasional commands from a remote controller. The controller communicates its requests via macro-commands that carry enough information to let the instrument proceed with the data acquisition and send data to the controller, or a data aggregator, when required. To eliminate the need for polling, the instrument is able to send data to the controller when they become available, without the controller having to issue a read request. Data sent upstreams are preceded with a data header that tells the controller exactly what type of data are being sent, what their format is, and which instrument they originate from.

1.1 Data exchange

To facilitate communication between different computer systems, multi-byte datums have to have their bytes arranged in proper order for the target computer to make sense. Internet traffic uses big-endian ordering and two-byte, four-byte and eight-byte datums have to be treated separately when converting from or to little-endian environments. Hence, some care is required when sending mixed-data type arrays.

Instruments based on ARM-9 processors will send and receive Ethernet data in big-endian convention. MSP430-based instruments are little-endian machines with limited computing resources. To save time, bulk data are sent and received in little-endian convention (LSB first), leaving the required conversions to the upstreams computer.

Commands sent from the controller to the instrument consist of a fixed-format header and command-specific data. Data sent by the instrument, consist of a fixed-format header with identification and formatting information for the data block that follows. Note that the instrument may send data to the controller either in response to a read command or autonomously in periodic intervals or in response to an event.

The command and data structures in use are described in the Command and Data Interface document.

Details of the communication with the embedded MCA are described in its respective API document: The eMorpho API documentation covers all single-channel eMorpho's. The API for the 4-channel qMorpho is covered in the qMorpho API document.

This document focuses on the code running on the instrument's MCU and it's interface to a remote controller.

2 Architecture

The instrument control software executes a never-ending main loop, which checks for the arrival of new commands, executes active commands and periodically, or on an alert, sends data to the remote controller (or possibly a separate data aggregator). The majority of the code consists of the main loop, the command parser and the Morpho API. This part is system independent and common to all instruments. In contrast, the functions moving data across the physical interfaces (from the MCU to the MCA and from the MCU to the controller) have specifically been written for the hardware they are implemented on.

Each instrument has instrument-specific resources, such as non-volatile RAM, a real-time clock, an SPI interface, etc. Access to these functions is clearly very system-dependent.

2.1 Main loop

The main loop repeatedly executes three sections: a command parser, periodic data acquisition and bulk data acquisition.

The command parser checks for newly arriving commands. In most cases the work only involves converting the command data into the native byte order and copying the data into a dedicated array. If the request is for the beginning of periodic data acquisition, the parser will also start the data acquisition.

The first periodic data acquisition implemented is a radiation scan. The instrument measures count rates and acquires histograms and periodically sends these data to the controller. The period for sending count rate data can be different from the period for sending histogram data.

Bulk data acquisition is supported for acquiring a programmable number of list mode buffers or traces and sending the data to the controller.

2.2 Interfaces

Logically, the interface between the local processor (MCU) and the Morpho MCA is through the Morpho API. This application programmer's interface is documented separately. The only instrument-specific aspect is its single function that actually moves the data from and to the MCA.

For the ARM-9 based systems the connection to the controller is established via Ethernet. For the small, ultra-low power systems based on MSP430 MCUs the interface to the controller is either via USB or an 8-bit asynchronous hand shake bus to the controllers MCU (ARM-9, for instance).

2.3 Initializations

There are a number of system specific initializations concerning clock speeds, power consumption, etc. which will be described in this chapter.

3 Command Processing

3.1 *General*

The receiving routine copies the command header and data into a data array called command_buffer. Its size is defined as `INSTRUMENT_COMMAND_SIZE` words. When a command has been received the receiving function returns a CMD_flag. When the flag is raised, the main loop inspects `command_buffer[CH_CMD_GROUP]` and in a switch statement selects a function to process commands from this group. Currently only the `CMD_DAQ` group is populated. Before the processing function returns it sets `command_buffer[CH_CMD_GROUP] = CMD_DONE` to indicate that no command processing is pending.

Below we identify the pair of a command group and the command itself with a colon notation as in `CMD_DAQ:DAQ_CMD_SETUP`.

3.1.1 Process command header

```
uint16 process_cmd_header(uint16 *command_buffer)
```

Type	Format	Name	Description
inout	uint16 *	command_buffer	The buffer contains the command header and the data that follow.

The function converts the 5 16-bit words command_buffer[1] to command_buffer[5] into native byte order. The first byte of the command header encodes the byte order of the remainder of the command header and of the command data. The function does not parse the rest of the command header and therefore does not know the format of the data that follow. Hence it makes no attempt at changing the data byte ordering.

Returns 0 if no errors are encountered.

3.1.2 Recognize command groups

Commands are organized into command groups and a switch statement operating on `command_buffer[CH_CMD_GROUP]` selects the appropriate command processing routine. This way software development on different aspects of the instrument, such as data acquisition and access to peripheral devices via SPI can be pursued independently without code or command-numbering overlap.

3.2 *DAQ commands*

The biggest group of commands is associated with data acquisition from the embedded MCA. The top-level function is `int DAQ_command(uint16 *command_buffer)` : It is nothing more than a switch on `command_buffer[CH_CMD_COMMAND]` to select a function to process the actual command.

3.2.1 MCA Setup

The purpose of this function is to provide setup data (i.e. control register contents) for the embedded MCA. It is called in response to `CMD_DAQ:DAQ_CMD_SETUP`.

```
uint16 daq_cmd_setup(uint16 *command_buffer)
```

Type	Format	Name	Description
inout	uint16 *	command_buffer	The buffer contains the command header and the data that follow.

The function converts the various sections after the command header into native byte order using the formatting information from previous sections. The details of this command are described in the Command and data Interface document.
In the current version of the software the command supports loading or computing any on of 7 predefined settings into the MCA, or copying a set of control registers from the command's data to the MCA.

3.2.2 Radiation Scan

This function prepares the instrument to measure count rates and acquire histograms. It will start the data acquisition and the main loop will then periodically read the data from MCA and send them on to the controller. There are two periods, one for the count rate measurements (stored in rad_scan[0]) and one for the histogram acquisition (stored in rad_scan[1]).

```
daq_cmd_scan(uint16 *command_buffer)
```

Type	Format	Name	Description
inout	uint16 *	command_buffer	The buffer contains the command header and the data that follow.

Global data array affected:

Type	Format	Name	Description
out	float32	rad_scan[0]	Count rates update period, in seconds
out	float32	rad_scan[1]	Histogram update period, in seconds
out	float32	rad_scan[2]	16-bit channel pattern specifies participating channels (for multi-device/multi-channel systems only).

The data reporting to the controller is inhibited if the auto-reporting bits in the MODE field of the command were cleared. Data reading from MCA is inhibited if the corresponding period is set to zero. Data reading and reporting occur in different blocks so programmers can insert code for analysis in between.

3.2.3 Trace acquisition

This function prepares the instrument to acquire a given number of pulse waveforms (traces) and send these to the controller.

```
daq_cmd_trace(uint16 *command_buffer)
```

Type	Format	Name	Description
inout	uint16 *	command_buffer	The buffer contains the command header and the data that follow.

Main loop data array affected:

Type	Format	Name	Description
out	uint16	trace_cmd[0]	Type of trace acquisition
out	uint16	trace_cmd[1]	Number of traces requested
out	uint16	trace_cmd[2]	Reserved to indicate special actions to be performed before the first trace acquisition (e.g. open a file).
out	uint16	trace_cmd[3]	Reserved to indicate special actions to be performed after each trace acquisition (e.g. analysis).
out	uint16	trace_cmd[4]	Reserved to indicate special actions to be performed after the last trace acquisition (e.g. close file).
out	uint16	trace_cmd[5]	Bit-field: Pattern of participating channels. For multi-device / multi-channel systems only; ignored otherwise.

The function will prepare the trace_cmd buffer such that the main loop will attempt to acquire the requested number of traces and send them to the controller. Resending this command with a zero requested traces will end an ongoing bulk data acquisition.

There are three types of traces:

1) Untriggered: This records a waveform at the time of the request, similar to the AUTO trigger on an oscilloscope.
2) Triggered: This records a waveform in response to an input pulse exceeding the energy trigger threshold.
3) Validated: This records a waveform at the time when the pulse energy has been measured and the pulse has been validated as acceptable (usually an integration time after the leading-edge trigger).

If a validated trace acquisition is requested there are multiple possible sources for the validation pulse, depending on the instrument.

On a single-channel instrument the only validation is local to the channel and consists of the pulse not being out of range or found to member to a pile up. In this case, no validation source need be given, and the corresponding bit field is ignored.

For multichannel instruments (qMorpho in a qDAQ instrument) the validation may be derived from an external trigger or from the coincidence logic running in the C&C FPGA. The validation bit field has to be set accordingly.

For a multichannel instrument the exact implementation of the trace acquisition function will depend on the requirements of the particular application. In this sample code for a multi-channel instrument we support validated trace acquisition with the validation pulse being common to all channels, e.g. distributed by the coincidence logic of the C&C FPGA.

This forces a common stop to each trace acquisition, but the user still has the power to allow any set of hit patterns, from the all-inclusive 0xFFFE (any one or more channels firing) to the most restrictive 0x8000 (all four channels in coincidence).

3.2.4 List mode acquisition

This function prepares the instrument to acquire a given number of pulse waveforms (traces) and send these to the controller.

```
daq_cmd_listmode(uint16 *command_buffer)
```

Type	Format	Name	Description
inout	uint16 *	command_buffer	The buffer contains the command header and the data that follow.

Main loop data array affected:

Type	Format	Name	Description
out	uint16	listmode_cmd[0]	Type of listmode acquisition
out	uint16	listmode_cmd[1]	Number of list mode buffers requested
out	uint16	listmode_cmd[2]	Reserved to indicate special actions to be performed before the first list mode acquisition (e.g. open a file).
out	uint16	listmode_cmd[3]	Reserved to indicate special actions to be performed after each list mode acquisition (e.g. analysis).
out	uint16	listmode_cmd[4]	Reserved to indicate special actions to be performed after the last list mode acquisition (e.g. close file).
out	uint16	listmode_cmd[5]	Bit-field: Pattern of participating channels. For multi-device / multi-channel systems only; ignored otherwise.

The function will prepare the listmode_cmd buffer such that the main loop will attempt to acquire the requested number of list mode buffers and send them to the controller. Resending this command with a zero requested traces will end an ongoing bulk data acquisition.

For multi-device / multi-channel instruments the implementation assumes that all list mode buffers fill at the same rate. When the first one fills up, as indicated in the Morpho status word, the main loop will read the list mode buffers from all participating channels and send the data to the controller.

4 Periodic Tasks

4.1 Measuring time

To support the periodic reporting of count rates, histograms and other quantities the instrument's processor needs to be able to measure elapsed time. While the actual time measurement is naturally a very processor-specific task, we attempt to encapsulate the task into a form common to all Morpho instruments.

```
refresh_times(float64 *wall_clock, float32 *elapsed_time, uint16 mode)
```

Type	Format	Name	Description
out	float64	wall_clock	Time since booting the processor, or resetting the wall clock
out	float32	elapsed_time	Array of eight values, to guide up to eight periodic tasks.
in	uint16	mode	Bit 0: 0 → Zero the elapsed_time array and the wall_clock, 1 → Update wall_clock and elapsed_time array

When called with bit 0 of mode set to zero, the wall_clock and elapsed_time array is cleared. When called with bit 0 set to 1, the clocks are updated by the amount of time that has passed since the last call to refresh_times. Units are in seconds, and on an ARM-9 the granularity is 1ms independent of the selected external clock speed. On an MSP430 running at 1MHz, the granularity is 125 ns but will be inversely proportional to the operating clock frequency.

When managing periodic tasks, the main loop will zero the respective elapsed_time entry whenever the associated periodic task has been executed. It will then wait until until the elapsed time exceeds the period defined in rad_scan[0] or rad_scan[1] before triggering the periodic action again.

5 Ethernet interface

5.1 IP-address discovery and initialization

This section relates to instruments with an embedded ARM-9 processor or where an ARM-9 is used to control an RDR (with MSP430 MCU) and connects via Ethernet to a host computer.

Reliable Ethernet communication (TCP/IP) follows a connection-oriented protocol which requires both sides to know each others IP-address. The software includes a discovery mechanism for the instruments' IP addresses if the instruments are located on the same local subnet as the controller.

On boot-up it calls udp_listener, which on its first call opens a UDP listening socket bound to any internet address. udp_listener is called every time the main loop repeats. On every call the udp_listener looks for a UDP message from a controller with a specific content.

When that message arrives the udp_listener stores the controller's IP-address in a global `struct sockaddr_in si_controller`. The udp_listener has also discovered the local MAC address, and stored its own IP-address as assigned by a DHCP router. It now passes this information to the tcp_talker, which sends the instrument's MAC address and IP address to the controller using the connection information stored in the structure si_controller.

On the other side, the controller, when starting up, created a listening socket bound to its own IP-address so it would be ready to receive data from the tcp_talker of any attached instrument on port 9931.. In a second step the controller calls a UDP broadcast function to send the discovery message to all units on the same subnet, sending to port 9930. The controller then uses its ENET_rcv function to collect the MAC and IP address data from all attached instruments.

Each instrument keeps a record of a successful uplink to the controller in a main loop variable ControllerHasSpoken. Once this variable is >0 two-way communication has been established.

On the instrument side two functions are involved:

```
int udp_listener (void):
```

No input parameters. Returns 1 if a UDP broadcast with the correct message has been received, 0 otherwise. The udp_listener listens on port 9930, as defined in ip_announce.h. It fills the following global variables:

1) `unsigned char Instrument_MAC_Address[8];`
2) `unsigned long local_ip_address_word;`
3) `struct sockaddr_in si_instrument, si_controller;`

```
void tcp_talker(void)
```

No input parameters. The function will send the instrument's data to the controller. The data packet includes the 6-byte MAC address followed by the 4-byte IP-address in network byte order (big endian). The `tcp_talker` sends its data to port #9931.

When an Ethernet-connected instrument boots up it opens a listening socket bound to its own IP-address as assigned by a DHCP router. The function that performs this task is tcp_init and by default it listens on port 9877.

```
SOCKET tcp_init(int port)
```

Type	Format	Name	Description
input	int	port	The controller will be sending commands to this port number.

Returns a socket a socket number. This listening socket will be kept live until the instruments reboots.

5.2 Receiving commands

The instrument receives commands on the listening socket opened by `tcp_init()`. The instrument uses `tcp_select_poll(..)` to receive commands from the controller.

```
tcp_select_poll (SOCKET ListenSocket, uint8 *command_buffer, uint16 *bytesRcv,
struct sockaddr_in *ControllerAddress)
```

Type	Format	Name	Description
input	SOCKET	ListenSocket	The socket returned by tcp_init(port)
output	uint8 *	command_buffer	Address of the command buffer array to fill
output	uint16 *	bytesRcv	Address for a variable storing the number of bytes received
output	sockaddr_in	controllerAddress	Ethernet connection data pertaining to the controller; filled by the accept function.

The function tcp_select_poll uses a non-blocking call to select to determine if the controller has sent a new command. If so, it accepts the connection and receives the command into the global array command_buffer.

Returns 1 if a command has been received, 0 if there was no activity and <0 if an error occurred.

5.3 Sending data

This is the low-level function that sends data upstreams to the controller.

```
send2Controller(struct sockaddr_in *si_controller, uint16 port, uint8 *buffer,
uint16 numBytes, uint16 *numBytesSent)
```

Type	Format	Name	Description
input	sockaddr_in *	si_controller	Internet address of the controller
input	uint16	port	Port to send the data to, 9932 by default.
input	uint8 *	buffer	Address of data buffer. The contents of buffer will be sent to the controller.
input	uint16	numBytes	Number of bytes the instruments wants to send.
output	uint16 *	numBytesSent	Address of a variable to receive the number of bytes that have been sent to the controller.

The function calls connect to create a connect socket and then uses send to send the data to the controller. On completion it shuts down and closes the connect socket.

6 Sending data to the controller

These are the top level functions for sending data back to the controller. They do call the hardware dependent send2Controller function but are otherwise identical on all instruments. The individual functions differ from each other mainly in terms of the information they load into the data header that precedes the data and the data formats they expect as an input. The functions will report the data byte order in the header but will not change the byte order prior to sending the data. The instrument ID, device and channel number are included in the data header so that the controller can tell from the data what their exact origin is.

Each time the instrument calls one of the send-data functions it connects to the controller, sends the data and closes the outbound socket. This way, the individual data messages will be well separated in the receiving queue of the controller.

6.1 Sending count rates

This function sends the count rates data to the controller.

```
 int send_rates(uint16 devNum, uint16 chNum, uint16 numEntries, uint16 format,
uint16 byteOrder)
```

Type	Format	Name	Description
input	uint16	devNum	Device number, 0 if there is only one Morpho
input	uint16	chNum	Channel number, 0 if there is only one Morpho
input	uint16	numEntries	Number of entries (9 for a single set of counters, 16+9 for a dual set of counters)
input	uint16	format	Use either DH_FMT_FLOAT32 or DH_FMT_UINT32
input	uint16	byteOrder	Byte order of the rates data

This function expects 4-byte data, either in uint32 or float32 format. Depending on the given format, the function will send the data in the array `rates_f[chNum]` or, for uint32) will send data from the `rates_i[chNum]` array. For qDAQ systems the data arrays are `rates_f[devNum][chNum]` and `rates_i[devNun][chNum]`, respectively.

Returns number of bytes sent if there is no error.

6.2 Sending a histogram

This function sends a histogram to the controller.

```
int send_histogram(uint16 devNum,  uint16 chNum,  uint16 offset,  uint16 numBins,
uint16 byteOrder)
```

Type	Format	Name	Description
input	uint16	devNum	Device number, 0 if there is only one Morpho
input	uint16	chNum	Channel number, 0 if there is only one Morpho
input	uint16	offset	Reserved
input	uint16	numBins	Number of bins to be sent
input	uint16	byteOrder	Byte order of the histogram data

The function will send the data of the array `histogram[chNum]` (`histogram[devNum][chNum]` for the qDAQ instruments). On the RDR the data come from the `bulk_data` array.
Returns number of bytes sent if there is no error.

6.3 Sending a trace

This function sends an acquired waveform to the controller.

```
int send_trace(uint16 devNum, uint16 chNum, uint16 numPoints, uint16 byteOrder)
```

Type	Format	Name	Description
input	uint16	devNum	Device number, 0 if there is only one Morpho
input	uint16	chNum	Channel number, 0 if there is only one Morpho
input	uint16	numPoints	Number of ADC samples to be sent
input	uint16	byteOrder	Byte order of the trace data

This function sends data out of the `pulse[chNum]` array or the `pulse[devNum][chNum]` array (qDAQ). The ADC samples are uint16 data, scaled such that the value 32767 corresponds to the full scale range of the ADC. On the RDR the data source is the `bulk_data` array.

Returns number of bytes sent if there is no error.

6.4 Sending list mode data

This function sends a list mode data buffer to the controller.

```
int send_listmode(uint16 devNum, uint16 chNum, uint16 numWords, uint16 byteOrder)
```

Type	Format	Name	Description
input	uint16	devNum	Device number, 0 if there is only one Morpho
input	uint16	chNum	Channel number, 0 if there is only one Morpho
input	uint16	numWords	Number of words to be sent
input	uint16	byteOrder	Byte order of the trace data

This function sends data from the `listmode[chNum]` array (`listmode[devNum][chNum]` for qDAQ). On the RDR the data are pulled from the `bulk_data` array.

Returns number of bytes sent if there is no error.

7 Data Space Organization

During data acquisition the main loop functions write data into predefined global data arrays from where the analysis and data sending functions pick up the data. All data arrays have a 12-byte space reserved at the beginning for the data header, which contains information on how the data were created and which instrument, device and channel is the origin of the data.

In the tables below `MD` is short for `MAX_DEVICES`, `MC` is short for `MAX_CHANNELS` and `DHL` is short for `DATA_HEADER_LENGTH`. All three are defined in the instrument's main header file.

In the MCA bases the main global data arrays are:

Array	Description
`float32 rates_f[MC][32+DHL/4]`	Count rates in floating point format; there is room for 32 entries.
`uint32 rates_i[MC][32+DHL/4]`	Count rates in uint32 format; there is room for 32 entries.
`uint32 histogram[MC][4096+DHL/4]`	Histogram as 4-byte bins; On ARM-9 processors there is room for 4096 bins; on MSP430 processors there is rooms for 1024 bins.
`unit16 pulse[MC][1024+DHL/2]`	Trace acquisition memory; there is room for 1024 ADC samples.
`unit16 listmode[MC][1024+DHL/2]`	Trace acquisition memory; there is room for a single 1024-word buffer

In the qDAQ systems the main global data arrays are:

Array	Description
`float32 rates_f[MD][MC][32+DHL/4]`	Count rates in floating point format; there is room for 32 entries.
`uint32 rates_i[MD][MC][32+DHL/4]`	Count rates in uint32 format; there is room for 32 entries.
`uint32 histogram[MD][MC][4096+DHL/4]`	Histogram as 4-byte bins; On ARM-9 processors there is room for 4096 bins; on MSP430 processors there is rooms for 1024 bins.
`unit16 pulse[MD][MC][1024+DHL/2]`	Trace acquisition memory; there is room for 1024 ADC samples.
`unit16 listmode[MD][MC][1024+DHL/2]`	Trace acquisition memory; there is room for a single 1024-word buffer

For qDAQ-4 units MD=1, and MC=4; for qDAQ-8 units we have MD=1, and MC=4.

In the portable RDR systems the main global data arrays are:

Array	Description
`float32 rates_f[MC][32+DHL/4]`	Count rates in floating point format; there is room for 32 entries.
`uint32 rates_i[MC][32+DHL/4]`	Count rates in uint32 format; there is room for 32 entries.
`uint32 bulk_data[1024+DHL/4]`	bulkdata as 4-byte bins; there is rooms for 1024 4-byte entries.

Note that MC=1 and DHL=12.
Due to memory limitations on the MSP430 the bulk data array receives either histogram, list mode or trace data. The sending functions for this kind of data will always use the bulk data array as their data source.