



**A  
P  
I**

## **API USER'S MANUAL**

### **DATA ACQUISITION SYSTEM FOR USE WITH SCINTILLATOR DETECTORS**

- ◆ **ACQUIRE ENERGY HISTOGRAMS**
- ◆ **MEASURE COUNT RATES**
- ◆ **DISPLAY PULSE SHAPES**



**BRIDGEPORT INSTRUMENTS, LLC  
6448 E HWY 290, STE D-100, AUSTIN TX 78723-1040  
WWW.BRIDGEPORTINSTRUMENTS.COM**

## Table of Contents

1 . Overview.....	4
2 . USB interface device driver functions.....	5
2.1 LoadDLL.....	5
3 . Morpho I/O functions.....	5
3.1 Morpho_Scan.....	6
3.2 Morpho_Open and Morpho_Close.....	6
3.3 Morpho_IO.....	7
3.4 Morpho_Factory_Init.....	7
4 . Midlevel functions.....	8
4.1 Morpho_Change_Destination.....	8
4.2 Morpho_Write_Controls.....	9
4.3 Morpho_Write_User.....	9
4.4 Morpho_Read_User.....	9
4.5 Morpho_Read_Statistics.....	10
4.6 Morpho_Read_Statistics2.....	10
4.7 Morpho_Read_Results.....	10
4.8 Morpho_Read_Histo.....	10
4.9 Morpho_Read_Trace.....	11
4.10 Morpho_Read_ListMode.....	11
5 . High-level functions.....	12
5.1 Morpho_Quick_Configure.....	12
5.2 Morpho_Set_Gain.....	13
5.3 Morpho_Get_Calibration.....	14
5.4 Morpho_Program_HV.....	14
5.5 Morpho_Suspend_DAQ.....	15
5.6 Morpho_Resume_DAQ.....	15
5.7 Morpho_End_DAQ.....	16
5.8 Morpho_Start_New_Histogram.....	16
5.9 Morpho_Add_To_Histogram.....	17
5.10 Morpho_Start_Any_Histogram.....	18
5.11 Morpho_Set_Request.....	19
5.12 Morpho_Start_Trace.....	19
5.13 Morpho_Start_ListMode.....	20
5.14 Morpho_Get_Status.....	20
5.15 HV2DAC.....	20
5.16 DAC2HV.....	21
5.17 Morpho_Get_Rates.....	21
5.18 Morpho_Get_Rates2.....	22
6 . Helper functions.....	22
6.1 SetBit, ClearBit, ToggleBit.....	22
7 . Unified Command Interface.....	23

8 . API Flavours.....	28
8.1 Integration with LabView under Windows.....	28
8.2 Working under Linux.....	28
9 . Programming Examples.....	29
9.1 Acquire A Histogram.....	29
9.2 Acquire A Triggered Trace.....	30
10 . Revision History.....	31

# 1 . Overview

The API consists of a hierarchical set of five layers. At the very bottom is the interface to the D2XX DLL provided by Future Technology Devices, Inc. Of this driver we are using the “Classic Interface” rather than the Virtual Com Port interface (BPI\_FT245BM\_API.c).

The second layer consists of functions to communicate with the hardware and a wrapper, called Morpho\_IO through which all communications attempts from higher level functions are routed (BPI\_Morpho\_Low.c). All functions that use Morpho\_IO are written in an operating system independent manner. We do, however, stick to the convention that the least significant byte or word is written and read first.

An intermediate layer provides easy to use read and write functions for direct access to the various back-end modules inside the FPGA (BPI\_Morpho\_Mid.c).

The top layer of the API comprises functions to operate the Morpho detector and convert all the information stored in the data acquisition board into physical quantities (BPI\_Morpho\_High.c).

A unified command interface provides a convenient, but optional, common entry point for all Morpho commands (BPI\_Morpho\_Functions.c). Its single function, MorphoCommand() also loads and manages the USB driver DLL. Users are encouraged to use this interface as it provides the most comprehensively tested method to operate one or more eMorphos.

In a typical application, programmers use only the high-level functions and some of the data read functions from the mid-level.

A few examples at the end of this documentation should help programmers to get started (BPI\_Morpho\_Examples.c).

<b>Functions.C (Command interface)</b> One function to dispatch all commands to High.C
<b>High.C (User interface)</b> Short scripts to serve typical commands
<b>Mid.C (FPGA interface)</b> Read / write access to all firmware modules
<b>Low.C (hardware interface)</b> Morpho: Scan, Open, Close, IO, Init
<b>API.C (dll interface)</b> Open DLL, function declarations

*Table 1: Software hierarchy.*

## 2 . USB interface device driver functions

The function to load the dll library file and all the wrapper functions are located in the file BPI\_FT245BM\_API.c.

### 2.1 LoadDLL

This function is used to load the dll library file (using LoadLibrary) and to create function pointers to all relevant functions within that DLL (using GetProcAddress). For each function we provide a wrapper such that the application programmer does not have to use pointers to a function. These functions are described the FTD2XX Programmer's Guide version 2.01 provided by FTDI.

## 3 . Morpho I/O functions

These are the lowest level functions to communicate with the hardware. They are located in the file BPI\_Morpho\_Low.c.

The API stores information about each Morpho that is connected to the computer in a number of global variables and arrays. Most notably, the serial numbers (stored as strings) and the device handles are stored in global arrays.

The global variables are listed below:

<i>Variable</i>	<i>Description</i>
char SerNums[MAX_DEVICES][SERNUM_LENGTH+1];	The strings containing the serial numbers
char *pSerNums[MAX_DEVICES+1];	Pointers to the above strings
FT_HANDLE MorphoHandles[MAX_DEVICES];	Device handles
char Manu[100];	Manufacturer name
char ManuID[100];	Manufacturer ID
char Descr[100];	Descriptor string
char SerNo[100];	Serial number
unsigned long NumberOfMorphos;	Number of Morpho-DAQ boards found by Morpho_Scan()
unsigned long Controls[MAX_DEVICES][16];	Array of 16-word control register images

### 3.1 Morpho\_Scan

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long *	buffer	First word receives number of eMorphos found

Finds all connected devices and gets their serial numbers. It takes one input parameter: a pointer to a unsigned long buffer to store the number of Morphos found (first position) followed by 7-digit serial numbers (one byte per buffer word). Make sure the buffer has room for at least 1+7\*MAX\_DEVICES bytes.

The function records the number of Morphos in the global variable NumberOfMorphos. It stores the serial number strings in SerNums and nulls the MorphoHandles. Morpho\_Scan limits the number of Morphos to MAX\_DEVICES should there be too many devices attached. The extraneous devices will be disregarded.

Returns 0 if the scan was successful and -1 otherwise.

### 3.2 Morpho\_Open and Morpho\_Close

<i>Input parameter, type and name</i>		<i>Description</i>
long	devNum	Device number

These two functions are used to open and close a particular device. They take one input parameter, namely the device number. Enumeration begins at zero. Morpho\_Open stores the device handle in Morpho\_Handles[devNum], while Morpho\_Close will NULL that entry. We open the devices using the serial number string as the identifier to the DLL.

Both functions operate on a single device if a legal device number is provided. But if devNum is negative or greater or equal MAX\_DEVICES, the functions will operate on all devices found during the last Morpho\_Scan.

Both functions return 0 if successful and -1 or -2 otherwise.

### 3.3 Morpho\_IO

This is the single I/O function to be used by the application programmer to exchange data with the FPGA on the Morpho data acquisition board and the EEPROM that supports the USB interface.

<i>Input parameter, type and name</i>		<i>Description</i>
long	devNum	Device number
unsigned long	direction	Used to distinguish reads from writes and the three possible targets, FPGA, EEPROM-USB and EEPROM user area.  The legal values are listed in the header file BPI_Morpho.h.
unsigned long	BytesPerDatum	The function combines byte-oriented data, which were received from the FPGA, into 2-byte or 4-byte datums. BytesPerDatum may be 2 or 4.
unsigned long*	data	Data buffer for read and write operations.
FT_PROGRAM_DATA*	ProgData	Pointer to a data structure containing USB-related EEPROM data. When not writing data to that part of the EEPROM, ProgData should be set to NULL.
unsigned long	nBytes	Number of bytes to be transferred. This parameter is ignored when reading or writing USB-related EEPROM data.

### 3.4 Morpho\_Factory\_Init

This function is used to program the USB-related part of the EEPROM including the serial number string. It will also write the nominal high-voltage DAC setting into the user area of the EEPROM. This function expects to see only one device on the bus and therefore does not accept a device number.

Returns 0 if successful, and -1 through -4 otherwise.

<i>Input parameter, type and name</i>		<i>Description</i>
char *	SerNo	Pointer to the serial number string.
unsigned long	nominalHVDAC	Nominal high-voltage DAC setting. Used to ensure calibrated signal gain.

## 4 . Midlevel functions

These functions manage the I/O with the functional units inside the FPGA. All data transfer from and to the FPGA follows the Windows/Intel convention of sending the low-order byte first. There are six different destinations for data exchange with the FPGA:

<i>Destination</i>		<i>Description</i>
Controls	Write	16 control registers, 2-byte long.
Statistics	Read	Four 32-bit counters (eight 16-bit registers) for run time and count rate measurements.
Results	Read	Eight 16-bit registers, storing list mode data such as an energy and a time stamp, as well as the temperature and some internal values such as the current DC-offset.
User	Read/Write	2048 byte of user area. This is just isolated memory and does not interact with the functional parts of the FPGA.
Histogram	Read	16384 bytes organized as 4K*4-byte energy histogram memory.
Trace	Read	1024 consecutive ADC samples, 2048 bytes
Listmode	Read	Up to 340 events w/ energy, pulse shape and time stamp, 2048 bytes

To make the interface robust and error tolerant Morpho cards employ the following protocol. All writes to the FPGA are sent in 34-byte data packets. In the first two bytes the host encodes which section of the FPGA is going to be addressed. The next 32 bytes are the data payload. The FPGA employs a time out. If a data packet has not been received in its entirety within 100  $\mu$ s after the start of that data packet, the FPGA resets its interface to be in a defined state for the next communication attempted. Similarly, the FPGA will be ready to receive data no later than 100  $\mu$ s after booting/power up.

To shield the application programmer from these hardware dependent details we have created a set of seven functions to exchange data with the various FPGA memories and registers.

### 4.1 *Morpho\_Change\_Destination*

Use this function to to change destination for the next read or write command. Note: There is no function to partially update the control registers.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
Unsigned long	destination	Destination for the next I/O command
unsigned long *	Controls	Points to 16-bit data stored in an array of type unsigned long. There must 16 entries in the array.

Returns 0 if successful, and -1 if Morpho\_IO returns an error.

Use NULL for the Controls pointer if you do not want to change the contents of the Controls register. In this case the function will use the values previously written to this device.

## 4.2 Morpho\_Write\_Controls

Use this function to update all control registers at once. Note: There is no function to partially update the control registers.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	Controls	Points to 16-bit data stored in an array of type unsigned long. There must be 16 entries in the array.

Returns 0 if successful, and -1 if Morpho\_IO returns an error.

## 4.3 Morpho\_Write\_User

Use this function to update the user memory in the FPGA. Note: There is no function to partially update the user memory.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	User	Points to 16-bit data stored in an array of type unsigned long. There must be 256 entries in the array.

Returns 0 if successful and -1 if Morpho\_IO returns an error.

## 4.4 Morpho\_Read\_User

Use this function to read the user memory in the FPGA. Note: There is no function to partially read this memory.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 16-bit data stored in an array of type unsigned long. There must be room for 256 entries in that array.

Returns 0 if successful and -1 if Morpho\_IO returns an error.

## 4.5 Morpho\_Read\_Statistics

Use this function to read the statistics registers. Note: There is no function to partially read the statistics registers.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 32-bit data stored in an array of type unsigned long. There must be room for 4 entries in that array.

Returns 0 if successful and -1 if Morpho\_IO returns an error.

## 4.6 Morpho\_Read\_Statistics2

Use this function to read the statistics registers in Morphos that have two sets of statistics counters. Note: There is no function to partially read the statistics registers.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 32-bit data stored in an array of type unsigned long. There must be room for 8 entries in that array.

Returns 0 if successful and -1 if Morpho\_IO returns an error.

## 4.7 Morpho\_Read\_Results

Use this function to read the results registers. Note: There is no function to partially read the results registers. Other Morpho documentation refers to these as the calibration registers.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 16-bit data stored in an array of type unsigned long. There must be room for 10 entries in that array.

Returns 0 if successful and -1 if Morpho\_IO returns an error.

## 4.8 Morpho\_Read\_Histo

Use this function to read the energy histogram memory registers. Note: There is no function to partially read the histogram memory.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 32-bit data stored in an array of type unsigned long. There must be room for 4096 entries in that array.

Returns 0 if successful, and -1 if Morpho\_IO returns an error.

## **4.9 Morpho\_Read\_Trace**

Use this function to read 1024 consecutive ADC samples. Note: There is no function to partially read the trace memory. Regardless of the ADC accuracy (10-bit or 12-bit) the data are delivered as 16-bit words with a sign bit in the MSB position and a 15-bit mantissa. Data from a 10-bit ADC are packed into 16-bit words as {0, ADC[9:0], 0,0,0,0,0}. Data from a 12-bit ADC are packed into 16-bit words as {0, ADC[11:0], 0,0,0}.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 16-bit data stored in an array of type unsigned long. There must be room for 1024 entries in that array.

Returns 0 if successful, and -1 if Morpho\_IO returns an error.

## **4.10 Morpho\_Read\_ListMode**

Use this function to read the list mode memory. Note: There is no function to partially read the list mode memory.

<i><b>Input parameter, type and name</b></i>		<i><b>Description</b></i>
unsigned long	devNum	Device number
unsigned long *	data	Points to 16-bit data stored in an array of type unsigned long. There must be room for 1024 entries in that array.

Data will be stored consecutively, 3 long-words per event up to a maximum of 340 events. numEvents = data[1023] contains the number of events in the buffer. Note that the list mode memory in the FPGA is never cleared. Hence you have to use numEvents to find the number of valid entries in the data buffer.

There are two selectable formats for the list mode data: Long time stamp and short time stamp.

In both cases an event uses three 16-bit words. The finest available time stamp granularity available is  $dT = 1 / \text{ADC\_Sampling\_Rate}$

When using the long time stamp, the time stamp is 32-bits long. The event data format is:

<i>Offset</i>	<i>Description</i>
data[0]	16-bit energy, 16 times the value entered into the histogram.
data[1]	16-bit time; lower word. 1 LSB = dT
data[2]	16-bit time; higher word

When using the short time stamp, the event data format is:

<i>Offset</i>	<i>Description</i>
data[0]	16-bit energy, 16 times the value entered into the histogram.
data[1]	16-bit phoswich sum, 16 times the value computed from the original sum and the PIT scaling value.
data[2]	16-bit time in units of $32 * dT$

Returns 0 if successful, and -1 if Morpho\_IO returns an error.

## 5 . High-level functions

This is a collection of functions used to start and stop the data acquisition, to set the controlling parameters and to perform some useful conversion functions.

### 5.1 Morpho\_Quick\_Configure

This function is used to set all the control register values (in the computer's memory) that control the signal processing. The function will set the integration time, the DC-Wait, the PUT value, start delay and the scaling values appropriately.

The legal scintillator types are listed in BPI\_Morpho.h. They include: SC\_PLASTIC, SC\_YAP\_CE, SC\_LA\_CL3, SC\_NA\_TL, SC\_BGO, SC\_CSI\_CO3, SC\_CSI\_NA, SC\_CSI\_TL, SC\_CDWO4.

If you operate a phoswich, use the settings for the slower scintillator. Then adjust the phoswich integration time PIT (CR10) to equal the recommended integration time of the faster scintillator. If you do rise time discrimination, set PIT to equal the longest rise time.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	scintillator	Type of scintillator

Control registers affected:

<i>Reg.</i>	<i>Updated parameter</i>
CR3	DC-Wait
CR4	Integration time
CR5	Start delay (for triggered trace capture)
CR10	Phoswich integration time
CR11	Pile up time
CR12	Energy and phoswich scaling values

Returns 0 if successful, -1 if Morpho\_Write\_Controls fails, and -2 if an unsupported scintillator was selected. In the latter case, the function takes no action.

## 5.2 Morpho\_Set\_Gain

Use this function to switch to one of 16 possible input amplifier gains. The table below shows the gain, expressed in Ohms, as a function of the gainSwitch (GS) value.

In the table below, the preferred settings (GS = 0, 1, 2, 4 and 8) are shown highlighted. Depending on speed of the input pulse, the other settings may cause pulse distortion and ringing. Check the resulting pulse form when using non-preferred gains.

<i>GS</i>	<i>gain</i>	<i>GS</i>	<i>gain</i>	<i>GS</i>	<i>gain</i>	<i>GS</i>	<i>gain</i>
0	100Ω	4	3400Ω	8	10100Ω	12	13400Ω
1	430Ω	5	3730Ω	9	10430Ω	13	13730Ω
2	1100Ω	6	4400Ω	10	11100Ω	14	14400Ω
3	1430Ω	7	4730Ω	11	11430Ω	15	14730Ω

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	gainSwitch	4-bit gain select

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR12	Bits 8, 9, 10 and 11 receive the gainSwitch value.

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

### 5.3 Morpho\_Get\_Calibration

Use this function to receive the calibration coefficients, namely the conversion from raw ADC numbers into meaningful quantities such as PMT anode current and charge per histogram bin.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	Controls	Control register array for this device
unsigned long *	data	Array in which the coefficients will be reported. There must be room for 9 entries.

Control registers affected: none.

The content of the data array will be:

<i>Offset</i>	<i>Description</i>
data[0]	PMT anode current per ADC step in units of 1.0e-12 A
data[1]	Incremental charge integral per ADC step and time step in units of 1.0e-21 C
data[2]	Charge per histogram bin in units of 1.0e-18 C
data[3]	Maximum amplitude in ADC steps
data[4]	Maximum PMT anode current in 1.0e-9 A.
data[5]	DC-offset in units of ADC_Full_Range / 4096
data[6]	PC-board temperature in units of 1/64 <sup>th</sup> degree C.
data[7]	Transimpedance, in Ohms, of the analog gain stage.
data[8]	Average input current in units of 1.0e-12 A.

Returns 0 if successful, and -1 or -2 if one of the data array pointers was NULL.

### 5.4 Morpho\_Program\_HV

This functions accepts a new value for the high-voltage controlling DAC and calls a task in the FPGA to reprogram the DAC.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	HVDAC_value	New value for the high-voltage controlling DAC.

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR7	Receives new DAC-value.
CR15	Set bit <b>PD</b> . This bit self-clears.

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.5 Morpho\_Suspend\_DAQ

This function suspends data acquisition and halts all counters.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR13	Sets bit 2

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.6 Morpho\_Resume\_DAQ

This function resumes data acquisition and releases all counters.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR13	Clears bit 2

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.7 Morpho\_End\_DAQ

This function forces an end to all pending and ongoing data acquisition by clearing the control register CR15.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR15	All bits cleared.

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.8 Morpho\_Start\_New\_Histogram

This function will begin a new energy histogram acquisition after erasing the previous histogram and resetting all statistics counters.

For real time runs, compute the requested run time as a 32-bit word, where an LSB corresponds to  $65536 / \text{ADC\_Sampling\_Rate}$ . Write this request to the control registers CR8 and CR9, using Morpho\_Set\_Request. Then call Morpho\_Start\_New\_Histogram with the LSB of hCtrl set to 1.

Alternatively, you can request a fixed number of histogram entries. In this this case load CR8 and CR9 with the number of counts requested for the histogram. Now call Morpho\_Start\_New\_Histogram with realTime=0.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	hCtrl	If LSB == 0, request is for a given number of histogram entries; else run for a given period of time.  If second bit is set histogram pulse maxima, else histogram energies.

Control registers affected:

<i>Reg.</i>	<i>hCtrl &amp; 1</i>	<i>Effect</i>
CR15	1	Set bits R, HT, CH, CS; Clear bit HC;
	0	Set bits R, HC, CH, CS; Clear bit HT;

<i>Reg.</i>	<i>hCtrl &amp; 2</i>	<i>Effect</i>
CR15	1	Set bit HA;
	0	Clear bit HA;

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.9 Morpho\_Add\_To\_Histogram

This function will begin a new energy histogram acquisition without erasing the previous histogram and without resetting all statistics counters. In other words, a previously stopped run will resume.

For real time runs, compute the requested run time as a 32-bit word, where an LSB corresponds to  $65536 / \text{ADC\_Sampling\_Rate}$ . Write this request to the control registers CR8 and CR9 using Morpho\_Set\_Request. Then call Morpho\_Start\_New\_Histogram with realTime=1.

Alternatively, you can request a fixed number of histogram entries. This is otherwise known as lifetime extension. In this case load CR8 and CR9 with the number of counts requested for the histogram. Now call Morpho\_Start\_New\_Histogram with realTime=0.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	hCtrl	If LSB == 0, request is for a given number of histogram entries; else run for a given period of time.  If second bit is set histogram pulse maxima, else histogram energies.

Control registers affected:

<i>Reg.</i>	<i>hCtrl &amp; 1</i>	<i>Effect</i>
CR15	1	Set bits R, HT; Clear bit HC;
	0	Set bits R, HC; Clear bit HT;

<i>Reg.</i>	<i>hCtrl &amp; 2</i>	<i>Effect</i>
CR15	1	Set bit HA;
	0	Clear bit HA;

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.10 Morpho\_Start\_Any\_Histogram

This function will begin a new energy histogram acquisition giving the user full control over how to handle previous data and when to stop the acquisition

For real time or life time runs, compute the requested run time as a 32-bit word, where an LSB corresponds to  $65536/ADC\_SAMPLING\_RATE$ . Write this request to the control registers CR8 and CR9, using Morpho\_Set\_Request.

Alternatively, you can request a fixed number of histogram entries. In this case load CR8 and CR9 with the number of counts requested for the histogram.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	hCtrl	Bitfield controlling the data acquisition

Control registers affected:

<i>Reg.</i>	<i>Affected bits</i>
CR15	Bits R, HT, HC, HA, CH, CS;

The parameter hCtrl is a bitfield controlling the histogram acquisition

<b>15</b>							<b>8</b>							<b>0</b>	
Unused / reserved											DL	CH	HA	LT	RT

**HA:** Set to histogram peak pulseheight values as opposed to energies.

**CH:** Set to clear previous histogram and reset the statistics counters prior to starting the new run.

**DL:** Set to delay starting the histogramming. This used in arrays with a command chain to prime all eMorphos before triggering the first in command to begin the run in all devices.

For Morphos with a version-2 FPGA the LT bit is sterile. When RT is set the run terminates on the condition Real time == Request. When it is not set, the run terminates on the condition No. of counts == Request. For Morphos with a version-3 FPGA the stop condition is selected by the combination of {LT,RT} as shown below.

<i>LT</i>	<i>RT</i>	<i>Stop condition</i>
0	0	Real time == Request
0	1	Real time == Request
1	0	Life time == Request
1	1	No. of counts == Request

Stop condition when histogramming, using an eMorpho with a version 3 FPGA.

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.11 Morpho\_Set\_Request

This function is used in conjunction with histogram data acquisition. It writes the requested run time or number of histogram entries into control registers 8 and 9.

A run time must be a 32-bit value. The least significant bit corresponds to  $65536 / \text{ADC\_Sampling\_Rate}$ .

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	value	Time or number of entries.

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR8	Set to lower word of request value.
CR9	Set to higher word of request value.

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

## 5.12 Morpho\_Start\_Trace

This function is used to start acquisition of an untriggered trace, a triggered trace or a triggered trace with a validated trigger. There are three legal value for “triggerType”: TRACE\_AUTO for untriggered traces; TRACE\_NORMAL for acquiring a triggered trace; and TRACE\_VALIDATED for a trace that was triggered and has been validated to not have been out of range or piled up with another pulse.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long	triggerType	Determines the type of waveform acquisition

Control registers affected:

<i>Reg.</i>	<i>triggerType</i>	<i>Effect</i>
CR15	TRACE_AUTO	Set bits CT, UT; Clear bits TR, VT;
	TRACE_NORMAL	Set bits R, TR, CT;
	TRACE_VALIDATED	Set bits R, VT, CT; Clear bits TR;

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

### 5.13 Morpho\_Start\_ListMode

This function is used to start acquisition of a sequence of events for which energies and time stamps are stored in a list.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number

Control registers affected:

<i>Reg.</i>	<i>Effect</i>
CR15	Set bits R, LM, CL;

Returns 0 if successful, and -1 if Morpho\_Write\_Controls fails.

### 5.14 Morpho\_Get\_Status

This function delivers the data acquisition status word in the first word of the provided data buffer.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	data[0] will hold the 16-bit status word.

Control registers affected: none.

Three bits are used in the status word:

<i>Constant</i>	<i>Value</i>	<i>Description</i>
ST_HISTOGRAM_DONE	1	Cleared when acquisition in process.
ST_LISTMODE_DONE	2	Cleared when acquisition in process.
ST_TRACE_DONE	4	Cleared when acquisition in process.

### 5.15 HV2DAC

This function computes the appropriate high-voltage DAC value for a required photomultiplier tube (PMT) high voltage. The function returns an unsigned long value.

The function limits the returned DAC value to 2730 to ensure that the requested PMT high voltage will never exceed 2000 V.

<i>Input parameter, type and name</i>		<i>Description</i>
double	HV	Required high voltage

## 5.16 DAC2HV

This function computes the photomultiplier tube (PMT) high voltage corresponding to the high-voltage DAC setting. The function returns a double.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	DAC	Value programmed into the high-voltage controlling DAC.

## 5.17 Morpho\_Get\_Rates

This function converts the data obtained from a call to Morpho\_Read\_Statistics into meaningful input/output count rates and data acquisition run times.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	raw data and computed results are stored in this array. There must be room for 9 entries.

The computational results are stored as follows:

<i>Entry</i>	<i>Description</i>
data[0]	Histogram Acquisition Time, in units of 65536 / ADC_Sampling_Rate.
data[1]	Number of histogrammed pulses
data[2]	Number of recognized incoming pulses
data[3]	Trigger dead time in sampling clock cycles.
data[4]	Histogram Acquisition Time, in units of 1.0e-3 s.
data[5]	Histogramming rate, in units of 1.0e-3 c.p.s.
data[6]	Recognized-triggers rate, in units of 1.0e-3 c.p.s.
data[7]	Trigger logic dead time, measured as a fraction of the total run time reported in Count_Rates[0]. 1 LSB = 1.0e-6.

<i>Entry</i>	<i>Description</i>
data[8]	Input count rate assuming random-in-time pulses, in units of 1.0e-3 c.p.s.

Note that data[0] through data[3] contain raw data as reported by the FPGA. The entries data[4] through data[8] are evaluated data.

Returns 0, if successful and -1 or -2 otherwise.

## 5.18 Morpho\_Get\_Rates2

This function converts the data obtained from a call to Morpho\_Read\_Statistics2 into meaningful input/output count rates and data acquisition run times. This function is used with Morpho variants that have two sets of statistics counters.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	devNum	Device number
unsigned long *	data	raw data and computed results are stored in this array. There must be room for 18 entries.

The computational results are stored as in Morpho\_Get\_Rates, except that there are now two 9-word-long block containing count rate data.

Returns 0, if successful and -1 or -2 otherwise.

## 6 . Helper functions

### 6.1 SetBit, ClearBit, ToggleBit

These functions are used to selectively manipulate a single bit in an unsigned long value.

<i>Input parameter, type and name</i>		<i>Description</i>
unsigned long	num	Number of the bit to be manipulated. Enumeration starts at 0.
unsigned long	value	Datum whose content is to be manipulated.

## 7 . Unified Command Interface

The API connects to IGOR-PRO and other graphics user interfaces via a unified command interface: `Morpho_Command()`. This consists of defining a suitable common data structure and a single function to select the appropriate action depending on the command that was sent. This interface provides some new functionality compared to what has been described in the sections above, but mostly it is intended to serve as a convenient collection of all the Morpho functions a programmer might ever need to call.

The function includes a compile time parameter, `FROM_CONSOLE`. Set this to 0 when using IGOR-PRO and to 1 otherwise. The switch allows for some data type checks that are useful when working with IGOR-PRO.

The common command structure is:

```
// This structure must be 2-byte-aligned because it receives parameters from Igor.
struct MorphoCmd {
    waveHndl bufferHandle;    // pointer to data buffer
    double command;          // command word
    double devNum;           // device number
    double result;           // return value
};
```

Note: When not using IGOR-PRO, replace `waveHndl` with `unsigned long*`. IGOR-PRO only uses the type `double` for variables, but allows `unsigned long` for arrays (called “waves” in IGOR). Hence, the three variables in the structure are of type `double`. The function `Morpho_Command` performs the necessary conversions.

On entry the function `Morpho_Command` checks if the DLL that contains the USB hardware driver functions has been loaded. If not, it calls `Load_DLL` to do so.

Assuming that **bufferHandle** points to an unsigned long array called **buffer**, a typical call sequence to `Morpho_Command` is:

```
MorphoCMD.bufferHandle = buffer;
MorphoCMD.command = MCMD_SCAN;
MorphoCMD.devNum = (double) 1;
Morpho_Command(&MorphoCMD);
```

Below follows a list of recognized commands and functions called and actions taken. Note that `ULONG` is shorthand for unsigned long.

MCMD\_SCAN: Morpho\_Scan().  
The number of Morphos found is recorded in buffer[0].

MCMD\_OPEN: Morpho\_Open((ULONG)p->devNum);

MCMD\_CLOSE: Morpho\_Close((ULONG)p->devNum);

MCMD\_GET\_HWDESC: Copy hardware descriptor to buffer. Receives 16 long words.

<i>Word</i>	<i>Description</i>
0	eMorpho version firmware version
1	Number of ADC bits
2	ADC voltage range in mV
3	ADC sampling rate in MHz
4	Communications clock rate in MHz
5	Size of histogram memory, in bytes
6	Size of waveform memory, in bytes
7	Size of list mode data memory, in bytes
8	Size of user data memory, in bytes
9 – 15	Unused

MCMD\_WRITE\_ADVANCED:  
Morpho\_IO( (ULONG)p->devNum,  
(ULONG) MORPHO\_WRITE\_FPGA,  
(ULONG) 2,  
(ULONG\*) ptrBuffer,  
(PFT\_PROGRAM\_DATA) NULL,  
(ULONG) 2\*WR\_SHORT\_DPS );

This writes a new packet header and 16 control register contents to the FPGA. ptrBuffer points to an array with 17 unsigned longs.

MCMD\_READ\_ADVANCED:  
Morpho\_IO( (ULONG)p->devNum,  
(ULONG) MORPHO\_READ\_FPGA,  
(ULONG) 2,  
(ULONG\*) ptrBuffer,  
(PFT\_PROGRAM\_DATA) NULL,  
(ULONG) ptrBuffer[0] );

Read data from the FPGA. The content of the last packet header written to the FPGA determines the memory location from which to read. ptrBuffer[0] initially has the number of bytes to read. Data are stored in the area pointed to by ptrBuffer.

MCMD\_WRITE\_EEPROM:  
Morpho\_IO( (ULONG)p->devNum,

```
(ULONG) MORPHO_WRITE_EEPROM,  
(ULONG) 2,  
(ULONG*) NULL,  
(PFT_PROGRAM_DATA) ptrBuffer,  
(ULONG) NULL );
```

Copies a single PFT\_PROGRAM\_DATA data structure into the EEPROM of the Morpho-DAQ card. Use with caution. The data structure is provided in the area that ptrBuffer points to.

#### MCMD\_READ\_EEPROM:

```
Morpho_IO( (ULONG)p->devNum,  
(ULONG) MORPHO_READ_EEPROM,  
(ULONG) 2,  
(ULONG*) NULL,  
(PFT_PROGRAM_DATA) ptrBuffer,  
(ULONG) NULL);
```

Read the PFT\_PROGRAM\_DATA data structure from the EEPROM of the Morpho-DAQ card. The data are stored in the area that ptrBuffer points to.

#### MCMD\_WRITE\_UA\_EEPROM:

```
Morpho_IO( (ULONG)p->devNum,  
(ULONG) MORPHO_WRITE_UA_EEPROM,  
(ULONG) 2,  
(ULONG*) ptrBuffer,  
(PFT_PROGRAM_DATA) NULL,  
(ULONG) 22);
```

Writes 22 bytes of data, LSB first, to the user area of the EEPROM of the Morpho-DAQ card. Use with caution. The data are read from the area that ptrBuffer points to.

#### MCMD\_READ\_UA\_EEPROM:

```
Morpho_IO( (ULONG)p->devNum,  
(ULONG) MORPHO_READ_UA_EEPROM,  
(ULONG) 2,  
(ULONG*) ptrBuffer,  
(PFT_PROGRAM_DATA) NULL,  
(ULONG) 22);
```

Read 22 bytes from the user area of the EPROM on the Morpho\_DAQ card. The data are transferred to the area pointed to by ptrBuffer.

#### MCMD\_GET\_UA\_SIZE:

```
Morpho_Get_UA_Size( (ULONG) p->devNum,  
(ULONG*) ptrBuffer );
```

Reports the size of the user area in ptrBuffer[0].

#### MCMD\_FACTORY\_INIT:

RESERVED

#### MCMD\_WRITE\_CONTROLS:

```
Morpho_Write_Controls( (ULONG) p->devNum,  
(ULONG*) ptrBuffer );
```

```

MCMD_WRITE_USER:
    Morpho_Write_User(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer,
        (ULONG*) Controls[(ULONG)p->devNum] );

MCMD_READ_STATISTICS:
    Morpho_Read_Statistics(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_STATISTICS2:
    Morpho_Read_Statistics2(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_RESULTS:
    Morpho_Read_Results(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_HISTOGRAM:
    Morpho_Read_Histogram(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_TRACE:
    Morpho_Read_Trace(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_LISTMODE:
    Morpho_Read_ListMode(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer );

MCMD_READ_USER:
    Morpho_Read_User(
        (ULONG) p->devNum,
        (ULONG*) ptrBuffer,
        (ULONG*) Controls[(ULONG)p->devNum] );

MCMD_QUICK_CONFIGURE:
    Morpho_Configure_For_Scintillator(
        (ULONG) p->devNum,
        (ULONG) ptrBuffer[0] );
    The scintillator type is stored ptrBuffer[0].

MCMD_GET_CALIBRATION:
    Morpho_Get_Calibration(
        (ULONG) p->devNum,
        (ULONG*) Controls[(ULONG)p->devNum]
        (ULONG*) ptrBuffer, );

```

MCMD\_SET\_HV:     Morpho\_Program\_HV(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0] );  
The DAC value, used to program the on-board 12-bit DAC, is stored in  
ptrBuffer[0].

MCMD\_END\_DAQ:    Morpho\_End\_DAQ((ULONG) p->devNum);

MCMD\_START\_NEW\_HISTOGRAM:  
Morpho\_Start\_New\_Histogram(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0] & 3 );  
The LSB of ptrBuffer[0] is used to select between real time acquisition  
and live time extension.  
The second bit (ptrBuffer[0] & 2) determines the method of pulse height  
computation: Proper energy summing when 0 and maximum value  
histogram when set to 1.

MCMD\_ADD\_TO\_HISTOGRAM:  
Morpho\_Add\_TO\_Histogram(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0] & 3 );  
The LSB of ptrBuffer[0] is used to select between real time acquisition  
and live time extension.  
The second bit (ptrBuffer[0] & 2) determines the method of pulse height  
computation: Proper energy summing when 0 and maximum value  
histogram when set to 1.

MCMD\_START\_ANY\_HISTOGRAM:  
Morpho\_Start\_New\_Histogram(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0] & 0x1F );

MCMD\_START\_TRACE:  
Morpho\_Start\_Trace(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0] & 3 );  
The content of ptrBuffer[0] is used to select the trigger type.

MCMD\_START\_LISTMODE:  
Morpho\_Start\_ListMode((ULONG) p->devNum);

MCMD\_SET\_REQUEST:  
Morpho\_Set\_Request(  
  (ULONG) p->devNum,  
  (ULONG) ptrBuffer[0]);  
The buffer contains the 32-bit data word.

MCMD\_SET\_GAIN:   Morpho\_Set\_Gain(  
  (ULONG)p->devNum,

(ULONG) ptrBuffer[0] & 0xF );  
The ptrBuffer[0] has the 4-bit gain value.

## 8 . API Flavours

The API has been written in such a manner as to minimize its dependency on the IGOR PRO integrated development system and the dependency on the Windows operating system. As need arises, Bridgeport Instruments will make available variations of the API to suit the technical requirements for integration with other tools or porting to other operating systems.

### 8.1 *Integration with LabView under Windows*

When working with LabView we encourage application programmers to use the unified command interface. It allows access to the entire eMorpho functionality through a single function. The original Morpho\_Command function accepts a single parameter, which is a pointer to a structure. This mechanism will not work with LabView and we provide a modified API for use with LabView. There, Morpho\_Command explicitly shows the four members of the structure:

```
Morpho_Command(ULONG devNum, ULONG cmd, ULONG *IO_buffer, long *result)
```

Here, ULONG is short for unsigned long. The modified API also makes no reference to IGOR specific code and it includes a DLL with all relevant functions visible to LabView.

This code version can be found in the folder LabViewAPI.

### 8.2 *Working under Linux*

There is not yet an API version dedicated to Linux. The LabView API may serve as a starting point. However, a number of changes will be necessary. To populate the DLL directory the header file BPI.h uses the Windows-specific \_\_declspec(dllexport) to make functions public. This has to be modified or removed for operation under Linux.

Secondly, Morpho\_IO has to be partially rewritten to use the Linux driver for the USB micro-controller FT245BM. The Linux driver for the FT245BM can be found at the following URL:

<http://www.ftdichip.com/Drivers/D2XX.htm>

Please note that the eMorpho API has been written to work with the D2XX-style drivers and not the VCP (virtual com-port) drivers.

In a similar manner, Morpho\_Scan will have to be updated as well.

All other functions should work as provided because they channel all their data transfers through Morpho\_IO.

## 9 . Programming Examples

We limit ourselves in this section to a few basic examples. The code fragments we show are not complete batch programs but serve to show how to apply the “tools” discussed so far.

### 9.1 Acquire A Histogram

This example walks you through every single step you have to take after powering up a Morpho\_DAQ board to acquire an energy spectrum—and there are not many steps to take.

After declaring two data buffers, the code proceeds to scan the USB ports to find any attached Morpho-DAQ board. It then opens board no. 1 for communication.

Now we program the HV-DAC, set the desired gain of the input amplifier and configure the Morpho board for operation with a NaI(Tl) scintillator.

Next we choose to acquire a spectrum for 10 seconds and start the data acquisition.

After 11 seconds we read the spectrum and either store it to the computer's hard disk or process in situ.

The examples ends with powering down the high-voltage generator and closing the communications channel to Morpho-DAQ board.

Using the unified command interface, only some 75 lines of application specific code were needed to complete the task.

```
int Take_Spectrum(){
    unsigned long histogram[4096]=0;
    unsigned long buffer[4096];
    struct MorphoCMD doThis;
    int err;

    doThis.devNum = 1;
    doThis.bufferHandle = buffer;

    // First, look for attached Morpho's
    // MorphoCommand will load the USB driver dll,
    // if this has not been done yet.

    doThis.command = MCMD_SCAN;
    MorphoCommand(&doThis);
    if(buffer[0] <= 0) return(-1); // no Morpho found

    doThis.command = MCMD_OPEN; // Now open Morpho number 1
    MorphoCommand(&doThis);
    err = doThis.result; // you can check this for errors

    doThis.command = MCMD_SET_HV;
    buffer[0] = HV2DAC(1000); // set HV to 1000V
    MorphoCommand(&doThis);

    // [...] wait for 10 seconds to let HV stabilize

    doThis.command = MCMD_SET_GAIN;
    buffer[0] = 2; // choose 1.1 k_Ohm transimpedance
    MorphoCommand(&doThis);
}
```

```

// set signal processing parameters
doThis.command = MCMD_QUICK_CONFIGURE;
buffer[0] = SC_NAI_TL; // select NaI(Tl)
MorphoCommand(&doThis);

doThis.command = MCMD_SET_REQUEST;
buffer[0] = 10.0 / 1.365e-3; // request 10 seconds
MorphoCommand(&doThis);

doThis.command = MCMD_START_NEW_HISTOGRAM;
buffer[0] = 1; // request is for a real time run
MorphoCommand(&doThis);

// [...] wait 11 seconds or check if histogram is done

doThis.command = MCMD_READ_HISTOGRAM;
doThis.bufferHandle = histogram;
MorphoCommand(&doThis);

// [...] now process or store your histogram

doThis.command = MCMD_SET_HV;
buffer[0] = HV2DAC(0); // power down HV
MorphoCommand(&doThis);

// close Morpho

doThis.command = MCMD_CLOSE;
doThis.bufferHandle = buffer;
MorphoCommand(&doThis);

return(0);
}

```

## 9.2 Acquire A Triggered Trace

To avoid repetition, we assume in this example that the Morpho-DAQ board has already been opened for communication and that all initializations such as programming the high-voltage DAC and setting the signal processing parameters have been performed.

This code fragment simply starts a waveform acquisition, checks if the acquisition has been completed and then reads the trace.

```

int Get_A_Trace(){
    unsigned long histogram[1024]=0;
    unsigned long trace[1024]
    unsigned long buffer[1024];
    struct MorphoCMD doThis;
    int err;
    int done;

    doThis.devNum = 1;
    doThis.bufferHandle = buffer;

    // assume we have opened a morpho,
    // set all signal processing parameters
    // and programmed the HV
    // Let's just acquire a trace

    doThis.command = MCMD_START_TRACE;

```

```

buffer[0] = TRACE_NORMAL;           // triggered trace
MorphoCommand(&doThis);

do{ // Are we done yet?
doThis.command = MCMD_GET_STATUS;
MorphoCommand(&doThis);
done = buffer[0] & ST_TRACE_DONE; // check status
} while(!done);

doThis.command = MCMD_READ_TRACE;
doThis.bufferHandle = trace;
MorphoCommand(&doThis);

// [...] process or store your scintillator signal
return(0);
}

```

## 10 . Revision History

- P1: Preproduction release. Verified with engineering prototypes of eMorpho and production versions of Morpho. April 2006.
- P2: Stable release; minor edits; October 2006
- R2: Update to include functionality of eMorpho firmware version 3.