BRIGHTSPEC

Reference Manual

BrightSpec Software Development Kit

SDK Reference Manual

Version 1.45

Release date: May 2017

DISCLAIMERS

Microsoft Windows is a registered trademark of Microsoft Corporation.
Visual Studio is a registered trademark of Microsoft Corporation.
Android is a trademark of Google Inc.
Eclipse is a trademark of the Eclipse Foundation.

CONTENTS

INTRODUCTION

This Manual describes the installation and use of the Software Development Kit (SDK) for the bMCA and Topaz series of Multichannel Analyzers.

The bMCA is a compact tube-base digital Multi-Channel Analyzer (MCA). Topaz is a compact stand-alone MCA based on the bMCA design. Both bMCA and Topaz are microprocessor-controlled and perform data acquisition using Digital Signal Processing algorithms. Several versions of the device are available:

- bMCA-USB: Tube base MCA, USB connection to host
- bMCA-Ethernet: Tube base MCA, Power Over Ethernet (POE) connection to host
- Topaz-Pico: Stand-alone MCA, USB connection to host
- Topaz-Pico BNC: Stand-alone, BNC connectors, USB connection to host
- Topaz-X: Stand-alone, high resolution for X-rays, USB connection to host

All device models can be accessed/controlled using a software library. The initial software library was developed for the Microsoft Windows platform. However, software libraries for other platforms are in development. Using these libraries, end-users can integrate the MCA devices into their instruments, experimental setups, or software applications.

Platform	Supported devices	Supported functionality
Microsoft Windows	bMCA USB bMCA Ethernet Topaz-Pico Topaz-X	Basic control Get/Set parameters Get spectrum/Get scope waveform LIST/TLIST mode (Topaz-X)
Android	bMCA USB Topaz-Pico	Basic control Get/Set parameters Get spectrum/Get scope waveform
Linux	bMCA USB bMCA Ethernet Topaz-Pico Topaz-X	Basic control Get/Set parameters Get spectrum/Get scope waveform LIST/TLIST mode (Topaz-X)

This document describes all the functionality implemented in the software library, and provides some basic examples on how to use the different functions from different programming languages. The library used is always called **DIMHW**. DIMHW stands for 'Device Interface Manager, Hardware level'.

Windows platform

On the Windows platform the library is presented as a dynamic link library: **DIMHW.dll**. The library can be used from different programming languages using standard WIN32 API calls.

- C++ using WIN32 API calls. Included project files for Visual studio versions 2005, 2008, 2010, 2012 and 2013.
- C# based on .Net library 2.0. The included project file is compatible with Visual studio version 2005 up to 2013.
- A FORTRAN F90 console based example (No graphical output).
- A Visual Basic 6.0 based example.
- Except for the FORTRAN example, all projects have a similar user interface. The example code shows how to find and access bMCA devices, control spectrum acquisition, change the device settings and visualize the spectrum without using external components. The code can be used as a base for developing software packages and applications.

Android platform

On the Android platform the library is presented as a Java compiled library file: **DIMHW.jar**. The library can be included in an Android project to support MCA functionality. The current version of the Android library only supports USB based bMCA and Topaz-Pico MCA's. To be able to use it, the Android device must support 'USB Host' functionality. Due to the nature of Java, the library has a slightly different interface than the Windows interface. However, most function and parameter names are still the same.

- A Java demo application for android. The project is compatible with Android Developer Tools (ADT).

Linux platform

On Linux, the library is distributed as a shared dynamic-link library: **libdimhw.so.*n*.*m*.*r***, where ***n***, ***m*** and ***r*** are the major version number, minor version number and revision level of the library release. For the library to work, **libusb-1.0** should be installed in the system.

SOFTWARE COMPATIBILITY AND SYSTEM REQUIREMENTS FOR WINDOWS

The software library for Windows runs under the following operating systems (OS) versions:

- Microsoft Windows XP
- Windows Vista
- Windows 7
- Windows 8
- Windows 10

Both 32-bit (x86) and 64-bit (x64) native libraries are available.

The software library for Windows consists of a set of DLLs (Dynamic-Link Libraries) that contains all necessary code to connect to the MCA's. For x86 and x64 Windows environments the DLL's need the Microsoft Visual Studio 2010 Runtime Libraries to be present in the system in order to function correctly. Both 32- and 64-bit versions of the MS Runtime Libraries are distributed with the SDK.

There is no special software or hardware requirements other than having the necessary PC communication port available (USB or Ethernet) for connecting to the bMCA device.

SOFTWARE COMPATIBILITY AND SYSTEM REQUIREMENTS FOR ANDROID

The software library for Android runs under the following operating systems version:

- Android version 4.0.0 and up (SDK version 14). This version of the SDK only supports standard USB MCA's (bMCA USB, Topaz-Pico).

The software library for Android consists of a Java library, **DIMHW.jar**. To be able to use the MCA the Android device needs to support USB Host functionality. This means the Android device can function as a powered USB host which provides power to the USB slave device.

SOFTWARE COMPATIBILITY AND SYSTEM REQUIREMENTS FOR LINUX

The software library for Linux has been tested under the following operating systems variants:

- 32-bit x86 Linux
- 64-bit x86 Linux
- 32-bit ARM Linux
- 64-bit ARM Linux with kernel 4.1.19 (Raspberry-Pi 3)

There are no special hardware requirements other than having the necessary PC communication port available (USB or Ethernet) for connecting to the bMCA device.

From the software point of view, **libusb-1.0** is required (**1.0.19** or newer recommended), even when connecting only to Ethernet devices.

Accessing USB devices under Linux may require setting the appropriate device permissions in order for the application programs to be able to connect to, and control, the bMCA. Modern Linux distributions use **udev** to manage device nodes in the **/dev** directory; in that case, a rules file that grants all users read/write permissions to BrightSpec devices would contain the following line:

```
SUBSYSTEM==”usb”, ATTR{idVendor}==”04d8”, ATTR{idProduct}==”f85d”, MODE=”0666”
```

The file could be named e.g. **brightspec.rules** (following the standard naming convention of *vendor.rules*), and should be placed in the **/etc/udev/rules.d** directory. You will need root permissions for that, of course, and you may have to customize the permissions according to your system policies. If you need, for example, to allow bMCA access only to the members of a certain group, then you should modify the **MODE** argument accordingly and add a **GROUP** statement to specify which group should own the device node. When in doubt, consult the **udev** documentation.

Once the rules file has been added, use the ‘**udevadm control --reload**’ command (also as root) to make the rules current. After that step, plug a bMCA device to a USB port (unplug and reconnect it if it was plugged before reloading the rules) and verify that applications can access it.

Older Linux distributions use **devfs** for the same purpose, which allows setting the permissions in a somewhat similar way. Details can be found in the **devfs** documentation and **devfs.rules** manual pages.

INSTALLATION

The BrightSpec SDK is distributed in an archive form. All the files necessary to develop an application that controls and communicates with BrightSpec MCA are included in the archive. The archive extracts to a folder with the following directory structure:

./BIN/	Contains the binary versions of the **DIMHW.DLL** for different windows architectures. (x86, x64, IA64). These are redistributable files that should be included with you application distribution. For x86 and x64 platforms the Visual Studio 2010 runtime redistributable is also included.
./CODE/	Contains examples with source code for various SDK demonstration programs:
./CODE/C Console SDK examples/	C version.
./CODE/C Console SDK examples/Acquisition.c	Acquisition code example.
./CODE/C Console SDK examples/Parameters.c	Parameters code example.
./CODE/C Console SDK examples/Listmode.c	List Mode code example.
./CODE/CPlusPlus - SDK DEMO/	C++ version.
./CODE/CPlusPlus - SDK DEMO/CODE	Source code.
./CODE/CPlusPlus - SDK DEMO/Projects	MS Visual Studio project files (VS 2005, 2008, 2010, 2012, 2013).
./CODE/CSharp - SDK DEMO/	C# version (.NET 2.0).
./CODE/CSharp - SDK DEMO/CODE	Source code.
./CODE/CSharp - SDK DEMO/Projects	Visual Studio project file (for VS 2005, 2008, 2010, 2012, 2013).
./CODE/FORTRAN - SDK DEMO/	FORTRAN F90 version.
./CODE/FORTRAN - SDK DEMO/CODE	Source code.
./CODE/VB6 - SDK DEMO/	Visual Basic 6.0 (VB6) version.
./CODE/Android Java - SDK DEMO	Java for Android DEMO project (for Eclipse ADT).
./DRIVERS/	Contains driver installers for different MCA versions (at this moment only USB). This should be included with your application distribution.
./DOCS/	Contains SDK documentation.
./INC/	Contains the necessary function declarations and header files to use the SDK with different programming languages.
./LIBS/	Contains the static library files to use the SDK with C/C++ (x86, x64, IA64) and the compiled library for Java Android.
./ABOUT.TXT	About file.

DISTRIBUTION FOR WINDOWS

When distributing your bMCA-supporting Windows application, be sure to add the following files to the installation:

No	File name	Recommended location	Functionality
1	DIMHW.dll	Application or development subdirectory or Windows\System32	Device Interface Manager. Contains the basic functions to communicate with a bMCA.
2	Libusb0.dll (only for older versions of DIMHW.dll, pre-v3.0.0.0)	Windows\System32 or Windows\SysWOW64	Driver for USB communication protocol.
3	Hidapi.dll (only on older versions of DIMHW.dll, pre-v1.2.0.2)	Windows\System32 or Windows\SysWOW64	Driver for USB HID version of the device.
4	vcredist_x64.exe or vcredist_x86.exe	Needs to be installed.	Visual studio 2010 runtime libraries.

INTEGRATION OF THE PROGRAMMING LIBRARIES IN YOUR DEVELOPMENT ENVIRONMENT

In order to make use of the MCA and its functionalities in your project, you must make the software library function calls are available in your development environment software. This is done in different ways for each programming language and platform.

Windows:

- For C++ based projects the function calls are made available through a standard C header file: **DIMHW.h**. Include this file in your C/C++ code in order to access the library functions. In addition, when building your project, make sure that the library definition file **DIMHW.lib** is added to the linking phase. The header file is independent of the building platform, whereas for the library definition file the correct version must be selected according to the target hardware platform (WIN32, x64 or IA64). The base library file **DIMHW.dll** must also be available to the software whenever it is run or being debugged. This can be achieved by placing it in the same folder as the executable or in the system path.

- For C# based projects a single file needs to be included: **DIMHW.cs**. This file contains all necessary definitions to use the software library from C#. The base library file, **DIMHW.dll**, must also be available to the software whenever it is run or debugged. This can be achieved by placing it in the same folder as the executable or in the system path.

- The Fortran F90 code contains all function definitions inside the example file. The functions are called using C binding. The base library file **DIMHW.dll** must also be available to the software whenever it is run or debugged. This can be achieved by placing it in the same folder as the executable or in the system path.

- For Visual basic 6.0 based project the function calls are defined in a module file: **DIMHW.bas**. The module has to be included in the Visual Basic 6 project for the functions to become available. The base library file **DIMHW.dll** must also be available to the software whenever it is run or debugged. This can be achieved by placing it in the same folder as the executable or in the system path.

Android:

- For Android Java projects, the **DIMHW.jar** compiled library needs to be included in the Android project. The library contains all code necessary to communicate with the MCA.

Linux:

- For C/C++ projects, the final executable should be linked with both **libdimhw** and **libusb-1.0**; that is usually done by adding the **–ldimhw** and **–lusb-1.0** options to the gcc/g++ linking command. If the library is not placed in the **/usr/lib** directory (or any other in the linker-loader path), then the **LD_LIBRARY_PATH** environment variable should be set accordingly.

- For Python projects, the **ctypes** module can be used to access directly the **libdimhw** entry points.

GENERIC SOFTWARE LIBRARY FUNCTIONS

In this chapter we describe the functions implemented in the software library. Basic examples are provided as well. For each programming language the corresponding function syntax is given. Depending on the programming language; there may be small differences in function names, parameter names, and parameter types.

Please note that in this document no explanation of the MCA device operation is given. For that purpose please consult the corresponding documentation (e.g. technical specification data sheets, and the Basic Software User's Manual [1 and 2]).

FINDING DEVICES

This function allows the client application to find out how many devices are connected to the client PC.

```
C/C++: int32_t FindDevices(int32_t DeviceType)
C#:    Int32 FindDevices(Int32 DeviceType)
F90:   INTEGER FUNCTION FindDevices(DeviceType)
       INTEGER:: DeviceType
       END FUNCTION FindDevices
VB6:   Function FindDevices(ByVal DeviceType As Long) As Long
Andrd: int FindDevices(int DeviceType)
```

Arguments

C/C++: DeviceType **C#:** DeviceType **F90:** DeviceType **VB6:** DeviceType **Andrd:** DeviceType	int32_t Int32 INTEGER Long int	The type of interface to query for devices: DEF_MCA_INTFC_ANY (0) – Any interface DEF_MCA_INTFC_LIBUSB (2) – LibUSB devices only DEF_MCA_INTFC_ETHERNET (3) – Ethernet devices only

Returns

An integer value representing the number of the devices found for the requested interface.

Notes

Use the parameter definitions stored into the library (see example).

Example (in C++):

```
NrOfDevices = FindDevicesEx(DEF_MCA_INTFC_ALL);
```

FINDING DEVICES (EXTENDED)

The extended version of the **FindDevices** function allows the client application to discover how many devices are connected to the client PC. This function adds a parameter which specifies a timeout value, used mainly for network searches. After the timeout expires, the function stops checking for device responses and returns the current number of found devices. The last parameter determines whether the function should clear its list of discovered devices before searching for new ones.

```
C/C++: int32_t FindDevicesEx(int32_t DeviceType, uint32_t Timeout, uint8_t NewSearch)
C#:    Int32 FindDevicesEx(Int32 DeviceType, UInt32 Timeout, byte NewSearch)
F90:   INTEGER FUNCTION FindDevicesEx (DeviceType, Timeout, NewSearch)
       INTEGER, VALUE :: DeviceType, Timeout, NewSearch
       END FUNCTION FindDevicesEx
VB6:   Function FindDevicesEx(ByVal DeviceType As Long, ByVal Timeout As Long, ByVal
       newSearch As Long) As Long
Andrd: int FindDevicesEx(int DeviceType, int Timeout, boolean NewSearch)
```

Arguments

C/C++: DeviceType **C#:** DeviceType **F90:** DeviceType **VB6:** DeviceType **Andrd:** DeviceType	int32_t Int32 INTEGER Long int	The type of interface to look for: DEF_MCA_INTFC_ANY (0) – Any interface DEF_MCA_INTFC_LIBUSB (2) – LIBUSB devices only DEF_MCA_INTFC_ETHERNET (3) – Ethernet devices only
C/C++: Timeout **C#:** Timeout **F90:** Timeout **VB6:** Timeout **Andrd:** Timeout	uint32_t UInt32 INTEGER Long int	Timeout value in milliseconds to search for devices.
C/C++: NewSearch **C#:** NewSearch **F90:** NewSearch **VB6:** NewSearch **Andrd:** NewSearch	uint8_t byte INTEGER*1 Byte boolean	Flag determines whether to clear internal list of found detectors before searching for new ones. Non-zero value (true) means clear the list.

Returns

An integer value representing the number of the devices found for the requested interface.

Notes

For compatibility with future library releases, use the parameter definitions (see example), and not numbers.

The Ethernet discovery mechanism uses UDP packets, and when searching for Ethernet devices the **FindDevicesEx** function will always block until the timeout fully expires. That is because the function does not know beforehand how many devices are in the network, and therefore cannot know how long to wait for slow packets to arrive. While increasing the timeout value will help with collecting packets in a congested network, a very large value can have a detrimental effect on the responsiveness of the application (e.g. too long waiting time for the user even if the network is not congested). Here is where the **NewSearch** argument becomes handy: when set to *false* it allows the function to be called in a loop with a relatively low timeout value, and thus the information can be updated on the screen as it becomes available, keeping the application responsive. Once done, the **FindDevicesEx** function must be called one last time with **NewSearch** set to *true* in order to close the discovery network socket.

When searching only for USB devices, the function will return when all the devices are enumerated, regardless of the timeout value specified.

Example (in C++):

```
NrOfDevices = FindDevicesEx(DEF_MCA_INTFC_ALL, 100, 1);
```

REQUESTING DEVICE INFORMATION

This function allows the client application to request information about found MCA devices.

```
C/C++: int32_t GetDeviceInfo(int32_t InterfaceNr,
       char *DevicePath, int32_t DevicePathSize,
       char *DeviceName, int32_t DeviceNameSize,
       char *Serial, int32_t SerialSize,
       int32_t *DeviceType);
C#:    Int32 GetDeviceInfo(Int32 InterfaceNr,
       StringBuilder DevicePath,  Int32 DevicePathSize,
       StringBuilder DeviceName, Int32 DeviceNameSize,
       StringBuilder Serial, Int32 SerialSize,
       ref Int32 DeviceType);
F90:   INTEGER FUNCTION GetDeviceInfo(InterfaceNr, DevicePath, DevicePathSize,
       DeviceName, DeviceNameSize, Serial, SerialSize, DeviceType)
       INTEGER, VALUE :: InterfaceNr, DevicePathSize, DeviceNameSize,
       SerialSize, DeviceType
       CHARACTER :: DevicePath(*), DeviceName(*), Serial(*)
       END FUNCTION GetDeviceInfo
VB6:   Function GetDeviceInfo(ByVal InterfaceNr As Long,
       ByVal DevicePath As String, ByVal DevicePathSize As Long,
       ByVal DeviceName As String, ByVal DeviceNameSize As Long,
       ByVal Serial As String, ByVal SerialSize As Long,
       ByRef DeviceType As Long) As Long
Andrd: String GetDeviceInfoPath(int InterfaceNr);
       String GetDeviceInfoName(int InterfaceNr);
       String GetDeviceInfoSerial(int InterfaceNr);
       int GetDeviceInfoType(int InterfaceNr);
```

Arguments (returns)

C/C++: InterfaceNr **C#:** InterfaceNr **F90:** InterfaceNr **VB6:** InterfaceNr **Andrd:** InterfaceNr	int32_t Int32 INTEGER Long int	Interface number, from 0 to the value returned by *FindDevices* or *FindDevicesEx* minus one.
C/C++: DevicePath **C#:** DevicePath **F90:** DevicePath **VB6:** DevicePath	char[] StringBuilder CHARACTER(*) String	Returns the path to the device (e.g. IP address in case of Ethernet devices, or bus number in case of USB).
C/C++: DevicePathSize **C#:** DevicePathSize **F90:** DevicePathSize **VB6:** DevicePathSize	int32_t Int32 INTEGER Long	Size in bytes of the above path string
C/C++: DeviceName **C#:** DeviceName **F90:** DeviceName **VB6:** DeviceName	char[] StringBuilder CHARACTER(*) String	Returns the device name
C/C++: DeviceNameSize **C#:** DeviceNameSize **F90:** DeviceNameSize **VB6:** DeviceNameSize	int32_t Int32 INTEGER Long	Size in bytes of the above name string
C/C++: Serial **C#:** Serial **F90:** Serial **VB6:** Serial	char[] Stringbuilder CHARACTER(*) String	Returns the serial number of the device. Serial numbers are read-only, and unique to each MCA device.
C/C++: SerialSize	int32_t	Size in bytes in the above serial string.

C#: SerialSize **F90:** SerialSize **VB6:** SerialSize	Int32 INTEGER Long	
C/C++: DeviceType **C#:** DeviceType **F90:** DeviceType **VB6:** DeviceType	int32_t Int32 INTEGER Long	The type of device for which the information is given (see the *FindDevices* or *FindDevicesEx* function)

Return

Windows: An integer, 0 meaning success and anything else failure (error codes are listed further in this document).
Android: String or int containing the requested parameter.

Example (in C++)

```
int32_t NrOfDevices;
// Device info
char Path[DEF_MCA_STRING_LENGTH];
char Name[DEF_MCA_STRING_LENGTH];
char Serial[DEF_MCA_STRING_LENGTH];
int32_t DeviceType;

NrOfDevices = FindDevicesEx(DEF_MCA_INTFC_ALL, 100, 1);
// Try to open one
for (int i = 0; i < NrOfDevices; ++i)
{
	// Get device info
	GetDeviceInfo(i, Path, DEF_MCA_STRING_LENGTH, Name, DEF_MCA_STRING_LENGTH,
			Serial, DEF_MCA_STRING_LENGTH, &DeviceType);

	// Print device info
	...
}
```

REQUEST DEVICE INFORMATION (EXTENDED)

This function allows the client application to request extended information about found MCA devices.

```
C/C++: int32_t GetDeviceInfoEx(int32_t InterfaceNr,
       char *Info, uint32_t InfoSize,
       uint32_t InfoIndex);
C#:    Int32 GetDeviceInfoEx(Int32 InterfaceNr,
       StringBuilder Info, UInt32 InfoSize,
       UInt32 InfoIndex);
F90:   INTEGER FUNCTION GetDeviceInfoEx(InterfaceNr, Info, InfoSize, InfoIndex)
       INTEGER, VALUE :: InterfaceNr, InfoSize, InfoIndex
       CHARACTER :: Info(*)
       END FUNCTION GetDeviceInfoEx
VB6:   Function GetDeviceInfoEx(ByVal InterfaceNr As Long,
       ByVal Info As String, ByVal InfoSize As Integer,
       ByVal InfoIndex As Long) As Long
```

Arguments (returns)

C/C++: InterfaceNr **C#:** InterfaceNr **F90:** InterfaceNr **VB6:** InterfaceNr **Andrd:** InterfaceNr	int32_t Int32 INTEGER Long int	Interface number, from 0 to the value returned by *FindDevices* or *FindDevicesEx* minus one.
C/C++: Info **C#:** Info **F90:** Info **VB6:** Info	char[] Stringbuilder CHARACTER(*) String	Returns the requested info
C/C++: InfoSize **C#:** InfoSize **F90:** InfoSize **VB6:** InfoSize	uint32_t UInt32 INTEGER Long	The size in bytes of the pre-allocated info string
C/C++: InfoIndex **C#:** InfoIndex **F90:** InfoIndex **VB6:** InfoIndex	uint32_t UInt32 INTEGER Long	The requested info type

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: String or int containing the requested parameter.

Example (in C++)

```
int32_t NrOfDevices;

// Device info
char szInfo[DEF_MCA_STRING_LENGTH];
uint32_t InfoIndex = DEF_MCA_PARAM_USERID;
int32_t DeviceType;

// Find devices
NrOfDevices = FindDevicesEx(DEF_MCA_INTFC_ALL, 100, true);

// Try to open one
for (int i = 0; i < NrOfDevices; ++i)
{
	// Get device info
	GetDeviceInfoEx(i, szInfo, DEF_MCA_STRING_LENGTH, InfoIndex);

	// Print device info
	...
}
```

OPEN DEVICE

Establishes the communication with a MCA device.

```
C/C++: int32_t OpenDevice(char *DevicePath, int32_t DeviceType,
       HANDLE *DeviceHandle);
C#:    Int32 OpenDevice(StringBuilder DevicePath, Int32 DeviceType,
       ref IntPtr DeviceHandle);
F90:   INTEGER FUNCTION OpenDevice(DevicePath, DeviceType, DeviceHandle)
       CHARACTER :: DevicePath(*)
       INTEGER, VALUE :: DeviceType
       INTEGER :: DeviceHandle
       END FUNCTION OpenDevice
VB6:   Function OpenDevice(ByVal DevicePath As String, ByVal DeviceType As Long,
       DeviceHandle As Any) As Long
Andrd: MCAdevice OpenDevice(String DevicePath, int DeviceType);
```

Arguments (returns)

C/C++: DevicePath **C#:** DevicePath **F90:** DevicePath **VB6:** DevicePath **Andrd:** DevicePath	char[] Stringbuilder CHARACTER(*) String String	The path to the device. For Ethernet MCA this is the device's IP address; for USB devices this is typically the system-dependent bus number. Generally, the value returned by *GetDeviceInfo* or *GetDeviceInfoEx* is used.
C/C++: DeviceType **C#:** DeviceType **F90:** DeviceType **VB6:** DeviceType **Andrd:** DeviceType	int32_t Int32 INTEGER Long int	The type of device to communicate with (see the *FindDevices* or *FindDevicesEx* function).
C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle	HANDLE IntPtr INTEGER Long	Returns a unique OS-dependent handle to the device.

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: MCAdevice object which represents the device.

Example (in C++)

```
HANDLE DeviceHandle = NULL;

int32_t Ret = OpenDevice(Path, DeviceType, &DeviceHandle);
if (Ret != MCA_SUCCESS)
{
       // Error
       ...
}
```

OPEN DEVICE (EXTENDED)

Extended function for establishing the communication with a MCA device. This function adds the possibility to define a timeout connection value.

```
C/C++: int32_t OpenDeviceEx(char *DevicePath, int32_t DeviceType,
       HANDLE *DeviceHandle, int32_t Timeout);
C#:    Int32 OpenDeviceEx(StringBuilder DevicePath, Int32 DeviceType,
       ref IntPtr DeviceHandle, uint32_t Timeout);
F90:   INTEGER FUNCTION OpenDeviceEx(DevicePath, DeviceType, DeviceHandle, Timeout)
       CHARACTER :: DevicePath(*)
       INTEGER, VALUE :: DeviceType, Timeout
       INTEGER :: DeviceHandle
       END FUNCTION OpenDeviceEx
VB6:   Function OpenDeviceEx(ByVal DevicePath As String, ByVal DeviceType As Long,
       DeviceHandle  As Any, ByVal Timeout As Long) As Long
Andrd: MCAdevice OpenDeviceEx(String DevicePath, int DeviceType, int Timeout);
```

Arguments (returns)

C/C++: DevicePath **C#:** DevicePath **F90:** DevicePath **VB6:** DevicePath **Andrd:** DevicePath	char[] Stringbuilder CHARACTER(*) String String	The path to the device. For Ethernet MCA this is the device's IP address; for USB devices this is typically the system-dependent bus number. Generally, the value returned by the *GetDeviceInfo* or *GetDeviceInfoEx* function is used.
C/C++: DeviceType **C#:** DeviceType **F90:** DeviceType **VB6:** DeviceType **Andrd:** DeviceType	int32_t Int32 INTEGER Long int	The type of device to communicate with (see the *FindDevices* or *FindDevicesEx* function).
C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle	HANDLE IntPtr INTEGER Long	Returns a unique OS handle to the device.
C/C++: Timeout **C#:** Timeout **F90:** Timeout **VB6:** Timeout **Andrd:** Timeout	unsigned long unsigned long INTEGER Long int	Timeout value for the function to connect to the device in milliseconds. If the connection cannot be made within this time, the function returns an error.

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: MCAdevice object which represents the device.

Example (in C++)

```
HANDLE hMCA = NULL;

int32_t Ret = OpenDeviceEx(DevicePath, DeviceType, &DeviceHandle, 2000);
if (Ret != MCA_SUCCESS)
{
      // Error
      ...
}
```

CLOSE CONNECTION

This function properly closes the connection to a MCA device

```
C/C++: int32_t CloseDevice(HANDLE DeviceHandle);
C#:    Int32 CloseDevice(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION CloseDevice(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION CloseDevice
VB6:   Function CloseDevice(ByVal DeviceHandle As Long) As Long
Andrd: boolean CloseDevice(MCAdevice DeviceHandle);
```

Arguments (returns)

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
CloseDevice(DeviceHandle);
DeviceHandle = NULL;
```

GET A PARAMETER FROM THE DEVICE

This function allows the client application to read a specific parameter from the MCA device. There are different functions for different programming languages.

```
C/C++: int32_t GetParam(HANDLE DeviceHandle, int32_t ParamId, void *ParamValue,
       int32_t ParamSize);
C#:    Int32 GetParam(IntPtr DeviceHandle, Int32 ParamId, StringBuilder ParamValue,
       Int32 ParamSize);
       Int32 GetParam(IntPtr DeviceHandle, Int32 ParamId, ref float[] ParamValue,
       Int32 ParamSize);
       Int32 GetParam(IntPtr DeviceHandle, Int32 ParamId, ref float ParamValue,
       Int32 ParamSize);
       Int32 GetParam(IntPtr DeviceHandle, Int32 ParamId, ref int[] ParamValue,
       Int32 ParamSize);
       Int32 GetParam(IntPtr DeviceHandle, Int32 ParamId, ref int ParamValue,
       Int32 ParamSize);
F90:   INTEGER FUNCTION GetParamStr(DeviceHandle, ParamId, ParamValue, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       CHARACTER :: ParamValue (*)
       END FUNCTION GetParamStr
       INTEGER FUNCTION GetParamSng(DeviceHandle, ParamId, ParamValue, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       REAL :: ParamValue
       END FUNCTION GetParamSng
       INTEGER FUNCTION GetParamInt(DeviceHandle, ParamId, ParamValue, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       INTEGER :: ParamValue
       END FUNCTION GetParamInt
VB6:   Function GetParam(ByVal DeviceHandle As Long, ByVal ParamId As Long,
       ParamValue As Any, ByVal ParamSize As Long) As Long
Andrd: String GetParamStr(MCAdevice DeviceHandle, int ParamId);
       int GetParamInt(MCAdevice DeviceHandle, int ParamId);
       float GetParamSng(MCAdevice DeviceHandle, int ParamId);
       float[] GetParamSngArr(MCAdevice DeviceHandle, int ParamId);
```

Arguments (returns)

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The device's unique handle identifying the MCA device.
C/C++: ParamId **C#:** ParamId **F90:** ParamId **VB6:** ParamId **Andrd:** ParamId	int32_t Int32 INTEGER Long int	Parameter type (see the parameter constants further in this document).
C/C++: ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue **C/C++:** ParamValue **C#:** ParamValue **VB6:** ParamValue	char[] StringBuilder CHARACTER(*) String float[] float[] float()	Returns the requested string parameter Returns the requested float array parameter

C/C++: ParamValue **C#:** ParamValue **VB6:** ParamValue	int[] int[] Integer()	Returns the requested int array parameter
C/C++: ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue	int int INTEGER integer	Returns the requested int parameter
C/C++: ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue	float float REAL Single	Returns the requested float parameter
C/C++: ParamSize **C#:** ParamSize **F90:** ParamSize **VB6:** ParamSize	int32_t Int32 INTEGER Long	Size in bytes of the parameter. This is 4 four for int or float parameters.

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: String, int, float, float[] containing the requested parameter.

Example (in C++)

In this example, parameters of several types are read. A thorough description of all parameters is available in the next chapter.

```
float AcqPreset = 0.0f;
float HVvalue = 0.0f;       // in Volts
int32_t HVstatus = 0;       // 0 = OFF, else ON
float CalFactors[3];

// Read the acquisition preset time
Ret = GetParam(DeviceHandle, DEF_MCA_PARAM_ACQ_PRESET, &AcqPreset, 4);

// Read the high voltage value
Ret = GetParam(DeviceHandle, DEF_MCA_PARAM_HV_VALUE, &HVvalue, 4);
Ret = GetParam(DeviceHandle, DEF_MCA_PARAM_HV_STATUS, &HVstatus, 4);

// Get the calibration parameters
Ret = GetParam(DeviceHandle, DEF_MCA_PARAM_CALIBRATION, &CalFactors[0], 12);
Offset = CalFactors[0];
Slope = CalFactors[1];
Quadratic = CalFactors[2];
```

SET A DEVICE PARAMETER

This function allows the client application to set a specific parameter of the MCA device. The C# wrapper has several overrides according to the parameter type.

```
C/C++: int32_t SetParam(HANDLE DeviceHandle, int32_t ParamId, void *ParamValue,
       int32_t ParamSize);
C#:    Int32 SetParam(IntPtr DeviceHandle, Int32 ParamId, StringBuilder ParamValue,
       Int32 ParamSize);
       Int32 SetParam(IntPtr DeviceHandle, Int32 ParamId, ref float[]ParamValue,
       Int32 ParamSize);
       Int32 SetParam(IntPtr DeviceHandle, int ParamId, ref float ParamValue,
       Int32 ParamSize);
       Int32 SetParam(IntPtr DeviceHandle, Int32 ParamId, ref int[]ParamValue,
       Int32 ParamSize);
       Int32 SetParam(IntPtr DeviceHandle, Int32 ParamId, ref int ParamValue,
       Int32 ParamSize);
F90:   INTEGER FUNCTION SetParamStr(DeviceHandle, ParamId, ParamValue, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       CHARACTER :: ParamValue (*)
       END FUNCTION SetParamStr
       INTEGER FUNCTION SetParamSng(DeviceHandle, ParamId, ParamValue, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       REAL:: ParamValue
       END FUNCTION SetParamSng
       INTEGER FUNCTION SetParamInt(DeviceHandle, ParamId, iParam, ParamSize)
       INTEGER, VALUE :: DeviceHandle, ParamId, ParamSize
       INTEGER :: ParamValue
       END FUNCTION SetParamInt
VB6:   Function SetParam(ByVal DeviceHandle As Long, ByVal ParamId As Long,
       ParamValue As Any, ByVal ParamSize As Long) As Long
Andrd: boolean SetParamInt(MCAdevice DeviceHandle, int ParamId, int ParamValue);
       boolean SetParamSng(MCAdevice DeviceHandle, int ParamId, float ParamValue);
       boolean SetParamSngArr(MCAdevice DeviceHandle, float[] ParamId, );
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.
C/C++: ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue **Andrd:** *Unsupported*	char[] StringBuilder CHARACTER(*) String	Contains the required string parameter to set
C/C++: ParamValue **C#:** ParamValue **VB6:** ParamValue **Andrd:** ParamValue	float[] float[] Single() float[]	Contains the float array parameter to set
C/C++: ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue	int int INTEGER Integer	Contains the int parameter to set

Andrd: ParamValue **C/C++:** ParamValue **C#:** ParamValue **VB6:** ParamValue **Andrd:** *Unsupported* **C/C++:** ParamValue **C#:** ParamValue **F90:** ParamValue **VB6:** ParamValue **Andrd:** ParamValue	Int int[] int[] Integer() float float Float Single float	Contains the int array parameter to set Contains the float parameter to set
C/C++: ParamSize **C#:** ParamSize **F90:** ParamSize **VB6:** ParamSize	int int INTEGER Long	Size in bytes of the parameter. This is 4 for int or float parameters.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

In this example the High Voltage value is set into a MCA device.

```
// Set the number of channels
iNumChannels = 1024;
iRet = SetParam(DeviceHandle, DEF_MCA_PARAM_NUM_CHANNELS, &iNumChannels, 4);

// Set the acquisition mode
iAcqMode = DEF_MCA_ACQ_MODE_TIME | DEF_MCA_ACQ_MODE_LIVE_TIME;
iRet = SetParam(DeviceHandle, DEF_MCA_PARAM_ACQ_MODE_EX, &iAcqMode, 4);

// Set acquisition preset time
sPresetTime = 600.0f;
iRet = SetParam(DeviceHandle, DEF_MCA_PARAM_ACQ_PRESET, &sPresetTime, 4);
```

START DATA ACQUISITION

This function is used to start the data acquisition using the settings previously stored into the device.

```
C/C++: int32_t StartAcquisition(HANDLE DeviceHandle);
C#:    Int32 StartAcquisition(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION StartAcquisition(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION StartAcquisition
VB6:   Function StartAcquisition(ByVal DeviceHandle As Long) As Long
Andrd: boolean StartAcquisition(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
if (!DeviceHandle) return;
// Start acquisition
Ret = StartAcquisition(DeviceHandle);
if (Ret != MCA_SUCCESS)
{
       // Error
}
```

Notes

The preset time, acquisition mode, high voltage, and all other applicable parameters should be set accordingly before calling this function.

Also, keep in mind that for some devices it may take some time for the high voltage to reach its preset value from the moment it is turned on (e.g. for bMCA and Topaz-Pico the delay may be up to 10 seconds). That is done to protect the electronics and the detector and/or Photo-Multiplier Tube (PMT).

It is not necessary to turn the high voltage off between measurements. That will avoid having to wait each time a few seconds before starting the acquisition. Furthermore, PMTs like the ones used in standard 14-pin detectors are designed to work continuously with HV applied 24 hours a day for the 365 days of the year. In fact, cycling the HV too often may cause the PMT to age somewhat faster.

STOP DATA ACQUISITION

This function is used to stop the bMCA data acquisition. The acquired data so far is kept in memory.

```
C/C++: int32_t StopAcquisition(HANDLE DeviceHandle);
C#:    Int32 StopAcquisition(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION StopAcquisition(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION StopAcquisition
VB6:   Function StopAcquisition(ByVal DeviceHandle As Long) As Long
Andrd: boolean StopAcquisition (MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
if (!DeviceHandle) return;
// Stop acquisition
Ret = StopAcquisition(DeviceHandle);
if (Ret != MCA_SUCCESS)
{
       // Error
}
```

GET DATA ACQUISITION STATUS

This function returns the current data acquisition status.

```
C/C++: int32_t GetAcquisitionStatus(HANDLE DeviceHandle, uint32_t *AcquisitionStatus);
C#:    Int32 GetAcquisitionStatus(IntPtr DeviceHandle, ref UInt32 AcquisitionStatus);
F90:   INTEGER FUNCTION GetAcquisitionStatus(DeviceHandle, AcquisitionStatus)
       INTEGER, VALUE :: DeviceHandle
       INTEGER :: AcquisitionStatus
       END FUNCTION
VB6:   Function GetAcquisitionStatus(ByVal DeviceHandle As Long,
       ByRef AcquisitionStatus As Long) As Long
Andrd: int GetAcquisitionStatus(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.
C/C++: AcquisitionStatus **C#:** AcquisitionStatus **F90:** AcquisitionStatus **VB6:** AcquisitionStatus	uint32_t UInt32 INTEGER Long	Returns the acquisition status (see table below)

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document). The actual status of the acquisition is returned in the **AcquisitionStatus** variable:

Status value	Meaning
DEF_MCA_ACQ_STATUS_STOPPED	Acquisition stopped, device not acquiring (idle)
DEF_MCA_ACQ_STATUS_ACQUIRING	Acquisition in progress
DEF_MCA_ACQ_STATUS_WAITING*	Device waiting for external start signal
DEF_MCA_ACQ_STATUS_STANDBY*	Acquisition paused by an external suspend signal

*Supported only by devices with external acquisition control (Topaz-X)

Android: int which represents acquisition status (1 = ACQUIRING / 0 = IDLE (not acquiring)).

Example (in C++)

```
int32_t iRet, AcquisitionStatus;
iRet = GetAcquisitionStatus(DeviceHandle, &AcquisitionStatus);
if (iRet == DEF_MCA_SUCCES)
{
	if (AcquisitionStatus == DEF_MCA_ACQ_STATUS_ACQUIRING)
	{
		// MCA is acquiring
		...
	}
}
```

CLEAR SPECTRAL DATA MEMORY

This function call will clear the MCA data memory.

```
C/C++: int32_t ClearMemory(HANDLE DeviceHandle);
C#:    Int32 ClearMemory(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION ClearMemory(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION ClearMemory
VB6:   Function ClearMemory(ByVal DeviceHandle As Long) As Long
Andrd: boolean ClearMemory(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
// Clear memory
iRet = ClearMemory(DeviceHandle);
```

CLEAR THE ACQUISITION TIME

This function call will erase the elapsed acquisition counters. If the acquisition is active, the counters are reset and the acquisition will continue.

```
C/C++: int32_t ClearTime(HANDLE DeviceHandle);
C#:    Int32 ClearTime(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION ClearTime(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION ClearTime
VB6:   Function ClearTime(ByVal DeviceHandle As Long) As Long
Andrd: boolean ClearTime (MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
// Clear time
iRet = ClearTime(DeviceHandle);
```

CLEAR ALL DATA

This function will clear both the spectrum memory and the elapsed acquisition timers.

```
C/C++: int32_t ClearAll(HANDLE DeviceHandle);
C#:    Int32 ClearAll(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION ClearAll(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION ClearAll
VB6:   Function ClearAll(ByVal DeviceHandle As Long) As Long
Andrd: boolean ClearAll(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
// Clear All
iRet = ClearAll(DeviceHandle);
```

ARM DIGITAL SCOPE

The MCA device has a built-in digital oscilloscope [2]. Using this scope the user can inspect the electronic pulses at the different stages of the signal processing path in the MCA device (e.g. digitized and processed pulses). This function call will instruct the MCA device to arm the scope in anticipation to a pulse.

```
C/C++: int32_t ArmScope(HANDLE DeviceHandle);
C#:    Int32 ArmScope(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION ArmScope(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION ArmScope
VB6:   Function ArmScope(ByVal DeviceHandle As Long) As Long
Andrd: boolean ArmScope(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

This example arms the bMCA scope:

```
// Arm the MCA scope
if (ArmScope(DeviceHandle) != DEF_MCA_SUCCESS)
       return false;
```

Notes

The scope needs to be armed always before capturing a waveform. Once armed, the hardware waits for the selected trigger source event (see the **DEF_MCA_PARAM_SCOPE_TRIG_SRC** parameter) before freezing the waveform in the scope memory. As soon as that happens, the **GetScopeStatus** function returns 1 to indicate that the waveform can be read via the **ReadScope** or **ReadScopeEx** function.

GET SCOPE STATUS

This function is used to retrieve the status of the MCA built-in digital oscilloscope.

```
C/C++: int32_t GetScopeStatus(IntPtr DeviceHandle, ref uint32_t ScopeStatus);
C#:    Int32 GetScopeStatus(IntPtr DeviceHandle, ref UInt32 ScopeStatus);
F90:   INTEGER FUNCTION GetScopeStatus(DeviceHandle, ScopeStatus)
       INTEGER, VALUE :: DeviceHandle
       INTEGER :: ScopeStatus
       END FUNCTION GetScopeStatus
VB6:   Function GetScopeStatus(ByVal DeviceHandle As Long,
       ByRef ScopeStatus As Long) As Long
Andrd: int GetScopeStatus(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long int	The unique device handle identifying the MCA device.
C/C++: ScopeStatus **C#:** ScopeStatus **F90:** ScopeStatus **VB6:** ScopeStatus	uint32_t UInt32 INTEGER Long	Returns the digital oscilloscope status. 0 – Armed, waiting for trigger 1 – Triggered

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: int which represents scope status (0 – Waiting for trigger / 1 – Triggered).

Example (in C++)

This example reads the scope status prior to reading the waveform:

```
uint32_t ScopeStatus = 0;
// Get the scope status
if (GetScopeStatus(DeviceHandle, &ScopeStatus) == DEF_MCA_SUCCESS)
{
    // If triggered, get the waveform
    if (ScopeStatus != 0)
    {
        int16_t Waveform[1024];
        if (ReadScope(DeviceHandle, Waveform) == DEF_MCA_SUCCESS)
            ProcessWaveform(Waveform);
    }
}
else
    return false;
```

Notes

The scope always needs to be armed by calling **ArmScope** before capturing a waveform. Once armed, the hardware waits for the selected trigger source event (see the **DEF_MCA_PARAM_SCOPE_TRIG_SRC** parameter) before freezing the waveform in the scope memory. As soon as the waveform has been stored, the **GetScopeStatus** function returns 1 indicating that the waveform can be read using the **ReadScope** or **ReadScopeEx** function.

READ SPECTRAL DATA

This function is used to read the current spectral data (channel contents) and elapsed time values.

```
C/C++: int32_t ReadSpectrum(HANDLE DeviceHandle, uint32_t *Spectrum, uint32_t
       *NrOfChannels, float *ElapsedRT, float *ElapsedLT);
C#:    Int32 ReadSpectrum(IntPtr DeviceHandle, UInt32[] Spectrum, ref UInt32
       NrOfChannels, ref float ElapsedRT, ref float ElapsedLT);
F90:   INTEGER FUNCTION ReadSpectrum(DeviceHandle, Spectrum, NrOfChannels, ElapsedRT,
       ElapsedLT)
       INTEGER, VALUE :: DeviceHandle
       INTEGER :: Spectrum(*), NrOfChannels
       REAL :: ElapsedRT, ElapsedLT
       END FUNCTION ReadSpectrum
VB6:   Function ReadSpectrum(ByVal DeviceHandle As Long, Spectrum As Any,
       ByRef NrOfChannels As Long, ByRef ElapsedRT As Single,
       ByRef ElapsedLT As Single) As Long
Andrd: CSpectrum ReadSpectrum(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.
C/C++: Spectrum **C#:** Spectrum **F90:** Spectrum **VB6:** Spectrum	unsigned long[ ] Uint32[ ] INTEGER(*) long()	Returns the channels contents in an array of 32-bit integers
C/C++: NrOfChannels **C#:** NrOfChannels **F90:** NrOfChannels **VB6:** NrOfChannels	uint32_t UInt32 INTEGER Long	Returns the number of channels read
C/C++: ElapsedRT **C#:** ElapsedRT **F90:** ElapsedRT **VB6:** ElapsedRT	float float REAL Single	Returns the corresponding elapsed real time, in seconds.
C/C++: ElapsedLT **C#:** ElapsedLT **F90:** ElapsedLT **VB6:** ElapsedLT	float float REAL Single	Returns the corresponding elapsed life time, in seconds.

Return

Windows: An integer, 0 meaning success and anything else failure (see error codes further in this document).
Android: CSpectrum object which represents the retrieved spectrum.

Example (in C++)

```
int32_t Ret = 0;
uint32_t Spectrum[4096];
uint32_t NrOfChannels;
float ElapsedLiveTime = 0.0f;
float ElapsedRealTime = 0.0f;

// Check if the handle is valid
if (DeviceHandle == NULL)
	return FALSE;

// Get a spectrum from the detector
Ret = ReadSpectrum(DeviceHandle, Spectrum, &NrOfChannels,
                                 &ElapsedRealTime, sElapsedLiveTime);
if (Ret != MCA_SUCCESS)
	// Error
	return FALSE;
```

READ LIST MODE DATA

This function is used to read and clear the current list mode buffer of the MCA.

```
C/C++: int32_t ReadListEvents(HANDLE DeviceHandle, uint32_t *List,
       uint32_t MaxNrOfEvents, uint32_t * NrOfEvents, uint32_t *Overflow);
C#:    Int32 ReadListEvents (IntPtr DeviceHandle, UInt32[] List,
       UInt32 MaxNrOfEvents, ref UInt32 NrOfEvents, ref UInt32 Overflow);
F90:   INTEGER FUNCTION ReadListEvents(DeviceHandle, List,
       MaxNrOfEvents, NrOfEvents, Overflow)
       INTEGER, VALUE :: DeviceHandle, MaxNrOfEvents
       INTEGER :: List(*), NrOfEvents, Overflow
       END FUNCTION ReadListEvents
VB6:   Function ReadListEvents(ByVal DeviceHandle As Long, List As Any,
       ByVal MaxNrOfEvents As Long, ByRef NrOfEvents As Long,
       ByRef Overflow As Long) As Long
Andrd: *UNSUPPORTED*
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle	HANDLE IntPtr INTEGER Long	The unique device handle identifying the MCA device.
C/C++: List **C#:** List **F90:** List **VB6:** List	uint32_t[ ] Uint32[ ] INTEGER(*) long()	Returns the list of events as a 32-bit integer
C/C++: MaxEvents **C#:** MaxEvents **F90:** MaxEvents **VB6:** MaxEvents	uint32_t int INTEGER Long	Returns the number of events in the list
C/C++: NrOfEvents **C#:** NrOfEvents **F90:** NrOfEvents **VB6:** NrOfEvents	uint32_t int INTEGER Long	Returns the number of events in the list
C/C++: Overflow **C#:** Overflow **F90:** Overflow **VB6:** Overflow	uint32_t int INTEGER Long	Returns if the internal buffer overflowed with list events (how many events were lost)

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Notes

LIST mode is currently supported only by the Topaz-X MCA devices.

Example (in C++)

```
int32_t Ret = 0;
uint32_t List[2048];
uint32_t NrOfEvents = 0;
uint32_t Overflow = 0;

// Check if the handle is valid
if (DeviceHandle == NULL)
	return FALSE;

// Get the latest event buffer from the MCA
Ret = ReadListEvents(DeviceHandle, List, 2048, &NrOfEvents, &Overflow);
if (Ret != MCA_SUCCESS)
	// Error
	return FALSE;
```

Data format

The format of the returned data depends on the mode selected by the **DEF_MCA_PARAM_LIST_MODE** parameter:

LIST mode

This is a simple LIST mode: for every processed event the MCA sends a 32-bit number where the lower 14 bits represent the channel number of the event and the remaining high bits are zeroed. No timing information is sent:

Energy event																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14-bit event channel number													

TLIST mode

This is a Time-stamped LIST mode: for every processed event the MCA sends a 32-bit word with the following information:

Energy event																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	17-bit Real-Time in 40ns ticks																	14-bit event channel number													

Real-Time clock will roll-over every 125000 ticks (5msec); at that moment the MCA will emit two additional words:

Real-Time mark																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	30-bit Real-Time in 5ms ticks																													

Live-Time mark																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	30-bit Live-Time in 5ms ticks																													

Using the above information, the full time-stamp of an event can be computed as:

event_time = (latest_RT_mark + event_timestamp * 4.0E-8) * 5.0E-3

READ SCOPE DATA

Using this function the application program can read the digital oscilloscope data. Please note that you have to arm the digital oscilloscope functionality in the MCA before making use of it, see **ArmScope** function described earlier.

```
C/C++: int32_t ReadScope(HANDLE DeviceHandle, int16_t *Waveform);
C#:    Int32 ReadScope(IntPtr DeviceHandle, short[] Waveform);
F90:   INTEGER FUNCTION ReadScope(DeviceHandle, Waveform)
       INTEGER, VALUE :: DeviceHandle
       INTEGER*2 Waveform(*)
       END FUNCTION ReadScope
VB6:   Function ReadScope(ByVal DeviceHandle As Long, Waveform As Any) As Long
Andrd: int ReadScope(MCAdevice DeviceHandle, short[] Waveform);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.
C/C++: Waveform **C#:** Waveform **F90:** Waveform **VB6:** Waveform **Andrd:** Waveform	int16_t[ ] short[ ] INTEGER*2(*) Integer() short[ ]	The waveform contents in an array of 16-bit integers.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

This example will retrieve the captured waveform. Assumes that the correct waveform has been selected and that the scope has been triggered:

```
uint32_t ScopeStatus = 0;
// Get the scope status
if (GetScopeStatus(DeviceHandle, &ScopeStatus) == DEF_MCA_SUCCESS)
{
    // If triggered, get the waveform
    if (ScopeStatus != 0)
    {
        int16_t Waveform[1024];
        if (ReadScope(DeviceHandle, Waveform) == DEF_MCA_SUCCESS)
           ProcessWaveform();
    }
}
else
    return false;
```

Notes

The scope always needs to be armed by calling **ArmScope** before capturing a waveform. Once armed, the hardware waits for the selected trigger source event (see the **DEF_MCA_PARAM_SCOPE_TRIG_SRC** parameter) before freezing the waveform in the scope memory. As soon as the last happens, the **GetScopeStatus** function returns 1 indicating that the waveform can be read using the **ReadScope** or **ReadScopeEx** function.

Data format

The returned data is an array of 1024 16-bit integers, each value representing a waveform sample as a signed value in the range -32768 to +32767. The Dwell interval (time between samples) is the inverse of the ADC sample rate, which can be queried via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE** parameter.

Topaz-X devices implement a dual-channel scope. Thus, the **ReadScopeEx** function should be used instead in order to gain access to the additional data.

READ SCOPE DATA (EXTENDED)

Using this function the application program can read the extended digital oscilloscope data. Please note that you have to arm the digital oscilloscope functionality in the MCA before making use of it, see **ArmScope** function described earlier.

```
C/C++: int32_t ReadScopeEx(HANDLE DeviceHandle, int16_t *Ch1Waveform,
       int16_t *Ch2Waveform, uint16_t *DigitalBits);
C#:    Int32 ReadScopeEx(IntPtr DeviceHandle, short[] Ch1Waveform, short[] Ch2Waveform,
       short[] DigitalBits);
F90:   INTEGER FUNCTION ReadScopeEx(DeviceHandle, Ch1Waveform, Ch2Waveform,
       DigitalBits)
       INTEGER, VALUE :: DeviceHandle
       INTEGER*2 Ch1Waveform(*), Ch2Waveform(*), DigitalBits(*)
       END FUNCTION ReadScopeEx
VB6:   Function ReadScopeEx(ByVal DeviceHandle As Long, Ch1Waveform As Any,
       Ch2Waveform As Any, DigitalBits As Any) As Long
Andrd: *UNSUPPORTED*
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle	HANDLE IntPtr INTEGER Long	The unique device handle identifying the MCA device.
C/C++: Ch1Waveform **C#:** Ch1Waveform **F90:** Ch1Waveform **VB6:** Ch1Waveform	int16_t[ ] short[ ] INTEGER*2(*) Integer()	The waveform contents in an array of 16-bit integers
C/C++: Ch2Waveform **C#:** Ch2Waveform **F90:** Ch2Waveform **VB6:** Ch2Waveform	int16_t[ ] short[ ] INTEGER*2(*) Integer()	The waveform contents in an array of 16-bit integers
C/C++: DigitalBits **C#:** DigitalBits **F90:** DigitalBits **VB6:** DigitalBits	int16_t[ ] short[ ] INTEGER*2(*) Integer()	The waveform contents in an array of 16-bit integers

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

This example will retrieve the captured waveform. Assumes that the correct waveform has been selected and that the scope has been triggered:

```
int32_t ScopeStatus = 0;
// Get the scope status
if (GetScopeStatus(DeviceHandle, &ScopeStatus) == DEF_MCA_SUCCESS)
{
    // If triggered, get the waveform
    if (ScopeStatus != 0)
    {
        int16_t Waveform1[1024], Waveform2[1024], DigitalBits[1024];
        if (ReadScopeEx(DeviceHandle,
                        Waveform1, Waveform2, DigitalBits) == DEF_MCA_SUCCESS)
           ProcessWaveform();
    }
}
else
    return false;
```

Notes

The scope always needs to be armed by calling **ArmScope** before capturing a waveform. Once armed, the hardware waits for the selected trigger source event (see the **DEF_MCA_PARAM_SCOPE_TRIG_SRC** parameter) before freezing the waveform in the scope memory. As soon as the last happens, the **GetScopeStatus** function returns 1 indicating that the waveform can be read using the **ReadScope** or **ReadScopeEx** function.

Data format

The waveform data is returned in two arrays (**Waveform1** and **Waveform2**) of 1024 16-bit integers each, where the values represents samples of the selected waveforms as a signed value between -32768 to +32767.

The **DigitalBits** array is also a 1024-element array of 16-bits integers, where individual bits contain information about several digital MCA signals:

Bit	Signal
0	Reserved
1	Reserved
2	TRP interval
3	Pile-up event
4	Pile-up inspect interval
5	Baseline detection
6	Event sampling instant
7	Fast discriminator event
8..15	Reserved

All three waveforms are acquired simultaneously, the Dwell interval (time between samples) being the inverse of the ADC sample rate which can be obtained via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE** parameter.

Topaz-Pico and bMCA devices implement only a single-channel scope. Thus, when using the **ReadScopeEx** function both **Waveform2** and **DigitalBits** will return zeros.

STORE DATA PARAMETERS INTO THE DEVICE'S PERMANENT MEMORY

Using this function call will result in saving any MCA parameters you have changed into the device's EEPROM. When using **SetParam** to change a device parameter, the changes will remain active until the power is cycled. If you want to make the changes persistent, use this function.

```
C/C++: int32_t SaveSettings(HANDLE DeviceHandle);
C#:    Int32 SaveSettings(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION SaveSettings(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION SaveSettings
VB6:   Function SaveSettings(ByVal DeviceHandle As Long) As Long
Andrd: boolean SaveSettings(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
// Save settings to EEPROM
Ret = SaveSettings(hDevice);
if (Ret == MCA_SUCCESS)
       MessageBox(NULL, TEXT("Settings saved successfully into EEPROM."), TEXT("bMCA"),
                  MB_OK);
```

Notes

Avoid calling this function unnecessarily, as EEPROM devices have a limited number of erase/write cycles. The memory chips used in bMCA and Topaz devices have a guaranteed life of more than 100,000 write cycles.

RESTORE DEFAULT SETTINGS

The MCA has a set of default settings that ensure proper device functionality. These settings are saved into the device's permanent memory at the factory, and can be recalled at any time later using this function.

```
C/C++: int32_t RestoreDefaults(HANDLE DeviceHandle);
C#:    Int32 RestoreDefaults(IntPtr DeviceHandle);
F90:   INTEGER FUNCTION RestoreDefaults(DeviceHandle)
       INTEGER, VALUE :: DeviceHandle
       END FUNCTION
VB6:   Function RestoreDefaults(ByVal DeviceHandle As Long) As Long
Andrd: boolean RestoreDefaults(MCAdevice DeviceHandle);
```

Arguments

C/C++: DeviceHandle **C#:** DeviceHandle **F90:** DeviceHandle **VB6:** DeviceHandle **Andrd:** DeviceHandle	HANDLE IntPtr INTEGER Long MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in C++)

```
// Restore default settings from EEPROM
Ret = RestoreDefaults(DeviceHandle);
if (Ret == MCA_SUCCESS)
       MessageBox(NULL, TEXT("Default settings loaded successfully."), TEXT("bMCA"),
                  MB_OK);
```

Notes

Calling this function also causes the default settings to be saved into the device's permanent memory (EEPROM). Thus, avoid calling this function unnecessarily, since EEPROM devices have a limited number of erase/write cycles.

ANDROID-SPECIFIC SOFTWARE LIBRARY FUNCTIONS

The following section describes support functions that are specific to the Android version of the library. These functions are necessary for correct application operation on the Android platform.

SET CONTEXT

This function sets the internal application context, which contains global information about an application environment. *The context needs to be set before calling any other function!*

```
void SetContext(Context context);
```

Arguments

Andrd: context	Context	The environment context.

Return

None.

Example (in Java)

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_main);

	// Set context
	DIMHW.SetContext(this);
}
```

GET LAST ERROR

This function returns the last error code of the MCA operation. Since the Java functions of the Android **DIMHW** library don't return error codes, the user can retrieve the last error code via this function.

```
int GetLastError(MCAdevice DeviceHandle);
```

Arguments

Andrd: DeviceHandle	MCAdevice	The unique device handle identifying the MCA device.

Return

An integer, 0 meaning success and anything else failure (see error codes further in this document).

Example (in Java)

```
// Open first device
mMCAdevice = DIMHW.OpenDevice(szPath, iDeviceType);
int iError = DIMHW.GetLastError(mMCAdevice);
if (iError != MCAdevice.MCA_SUCCESS)
{
	ShowMessageBox("bMCA error", "No devices found.");
	return;
}
```

CHECK IF THE MCA IS CONNECTED

This function checks if the MCA is connected.

```
boolean IsConnected(MCAdevice DeviceHandle);
```

Arguments

Andrd: DeviceHandle	MCAdevice	The unique device handle identifying the MCA device.

Return

A boolean which represents whether or not the MCA is connected (true = connected, false = not connected).

Example (in Java)

```
if (!DIMHW.IsConnected(mMCAdevice))
{
	// Find connected devices
	int iFound = DIMHW.FindDevices(DIMHW.DEF_MCA_INTFC_ALL);
	if (iFound <= 0)
	{
		ShowMessageBox("bMCA error", "No devices found.");
		return;
	}
	...
```

SET MCA EVENT LISTENERS

These functions set the listeners (events) which are executed when a certain operation is finished.

```
void SetMCADeviceOpenedListener(MCADeviceOpenedListener listener);
void SetMCADeviceClosedListener(MCADeviceClosedListener listener);
void SetMCADeviceErrorListener(MCADeviceErrorListener listener);
```

Arguments

Andrd: listener	MCADeviceOpenedListener MCADeviceClosedListener MCADeviceErrorListener	Function which is run when the device is opened successfully. Function which is run when the device is opened successfully. Function which is run when the device is encounters an error.

Return

None.

Example (in Java)

```
public class MainActivity extends Activity implements DIMHW.MCADeviceOpenedListener,
DIMHW.MCADeviceClosedListener, DIMHW.MCADeviceErrorListener
{
	protected void onCreate(Bundle savedInstanceState)
	{
		super.onCreate(savedInstanceState);

		// Set context
		DIMHW.SetContext(this);
		// Set event handlers
		DIMHW.SetMCADeviceOpenedListener(this);
		DIMHW.SetMCADeviceClosedListener(this);
		DIMHW.SetMCADeviceErrorListener(this);
	}
	// Interface implements
	public void MCADeviceOpened(MCAdevice arg0)
	{
		ShowMessageBox("bMCA opened", "bMCA opened");
	}
	public void MCADeviceClosed(MCAdevice arg0)
	{
		// Close the MCA if still open
		if(DIMHW.IsConnected(arg0))
			DIMHW.CloseDevice(arg0);
	}
	public void MCADeviceError(MCAdevice device, int iErrorCode)
	{
		// An error occurred, iErrorCode contains the error-type
		ShowMessageBox("Error", "Error");
	}

	...
}
```

NAMED MCA PARAMETERS SUPPORTED BY DIMHW

The previous section described the different functions that are available to access, control and configure the MCA. Using the **GetParam** and **SetParam** sets of functions, the MCA characteristics can be changed or inquired. This section describes all the different parameters that can be configured.

In order to ensure portability and compatibility with future releases, it is recommended to use the named parameters instead of specifying numeric values in the corresponding function calls.

Parameter name	Type	Description
DEF_MCA_PARAM_DEV_NAME	String	Device name. Read-only parameter.
DEF_MCA_PARAM_SERIAL_NUMBER	String	Device serial number. Read-only parameter.
DEF_MCA_PARAM_FW_VERSION	String	Firmware version. Read-only parameter.
DEF_MCA_PARAM_FW_DATE	String	Firmware date. Read-only parameter.
DEF_MCA_PARAM_PROD_DATE	String	Production date. Read-only parameter.
DEF_MCA_PARAM_MCA_MODE	Int32	MCA operating mode. 1=MCS, 0=PHA
DEF_MCA_PARAM_NUM_CHANNELS	Int32	Number of spectrum channels.
DEF_MCA_PARAM_PHA_LLD	Int32	Lower Level Discriminator (LLD) channel.
DEF_MCA_PARAM_PHA_ULD	Int32	Upper Level Discriminator (ULD) channel.
DEF_MCA_PARAM_ACQ_PRESET	Float	Acquisition preset time. Deprecated, use DEF_MCA_PARAM_PRESET_TIME instead.
DEF_MCA_PARAM_PRESET_TIME	Float	Acquisition preset time.
DEF_MCA_PARAM_ACQ_MODE	Int32	Acquisition mode. Deprecated, use DEF_MCA_PARAM_ACQ_MODE_EX instead.
DEF_MCA_PARAM_INP_POLARITY	Int32	Input polarity. 1=Positive, 0=Negative.
DEF_MCA_PARAM_GAIN_DIGITAL	Int32	Digital gain, 0 to 9 as power of 2.
DEF_MCA_PARAM_GAIN_COARSE	Int32	Coarse gain, 0 to 3 as power of 2
DEF_MCA_PARAM_GAIN_FINE	Int32	Fine gain, 0 to 4095 representing gain multiplier between 1 and 2.
DEF_MCA_PARAM_RISE_TIME	Int32	Rise time, 0 to 511.
DEF_MCA_PARAM_FLAT_TOP	Int32	Flat top, 0 to 511.
DEF_MCA_PARAM_PZ_ADJ	Int32	Digital Pole/Zero adjustment, 0 to 511.
DEF_MCA_PARAM_DIGITAL_PZ_ADJ	Int32	Same as DEF_MCA_PARAM_PZ_ADJ.
DEF_MCA_PARAM_THRESHOLD	Int32	Fast discriminator (noise) threshold.
DEF_MCA_PARAM_BLR_ENABLE	Int32	Baseline restorer. 1=enabled. 0=disabled.
DEF_MCA_PARAM_PUR_ENABLE	Int32	Pile up rejector. 1=enabled. 0=disabled.
DEF_MCA_PARAM_HV_VALUE	Int32	High voltage value. 0 to 1500V in 0.1V increments.
DEF_MCA_PARAM_HV_STATUS	Int32	High voltage status. 1=enabled. 0=disabled.
DEF_MCA_PARAM_SCOPE_TRIG_LVL	Int32	Scope trigger level.
DEF_MCA_PARAM_SCOPE_WAVEFORM	Int32	Scope waveform.
DEF_MCA_PARAM_CALIBRATION	Float[3]	Energy calibration coefficients as 3-element array: offset, slope and quadratic terms.
DEF_MCA_PARAM_MCS_CHANNELS	Int32	Number of channels in MCS mode.
DEF_MCA_PARAM_MCS_CURRENT_CHANNEL	Int32	Current MCS channel.
DEF_MCA_PARAM_PRESET_COUNTS	Int32	Preset counts for preset counts acquisition mode.
DEF_MCA_PARAM_CURRENT_COUNTS	Int32	Current counts in preset counts ROI.
DEF_MCA_PARAM_ACQ_MODE_EX	Int32	Extended acquisition mode.
DEF_MCA_PARAM_PHA_PRESET_LLD	Int32	LLD of ROI for preset counts mode.
DEF_MCA_PARAM_PHA_PRESET_ULD	Int32	ULD of ROI for preset counts mode.
DEF_MCA_PARAM_EXT_COUNTS	Int32	Number of counts from external counter input.
DEF_MCA_PARAM_EXT_IO_MODE	Int32	External I/O direction. 0=input, 1=output.
DEF_MCA_PARAM_ICR_COUNTS	Int32	Incoming Counts register.
DEF_MCA_PARAM_HW_PZ_ADJ	Int32	Hardware Pole-Zero adjustment.
DEF_MCA_PARAM_OCR_COUNTS	Int32	Output Counts register.

DEF_MCA_PARAM_TRP_COUNTS	Int32	Reset Pulse counts.
DEF_MCA_PARAM_PUR_GUARD	Int32	Pile-up guard time.
DEF_MCA_PARAM_TRP_GUARD	Int32	Reset pulse guard time.
DEF_MCA_PARAM_LT_TRIM	Int32	Extended Live-Time adjustment.
DEF_MCA_PARAM_ADC_SAMPLING_RATE	Int32	ADC sampling rate.
DEF_MCA_PARAM_TIMING_INFO	Int32[]	Timing info.
DEF_MCA_PARAM_HV_INFO	Int32[]	High voltage info.
DEF_MCA_PARAM_LIST_MODE	Int32	List mode.
DEF_MCA_PARAM_SCOPE_TRIG_SRC	Int32	Scope trigger source.
DEF_MCA_PARAM_DIFF_SEL	Int32	Pulse-tail time constant selector.
DEF_MCA_PARAM_GPIO1_MODE	Int32	GPIO1 function selection.
DEF_MCA_PARAM_GPIO2_MODE	Int32	GPIO2 function selection.
DEF_MCA_PARAM_NETBIOS_NAME	String	Netbios name.
DEF_MCA_PARAM_DEFAULT_IP	UInt32[]	Default TCP/IP address.
DEF_MCA_PARAM_REMOTE_IP	UInt32	Remote IP address.
DEF_MCA_PARAM_REMOTE_PORT	UInt16	Remote Port number.
DEF_MCA_PARAM_USERID	String	User ID
DEF_MCA_PARAM_GROUPID	String	Group ID
DEF_MCA_PARAM_USERDATA	UInt8[]	User data.

DEF_MCA_PARAM_DEV_NAME

Device name. String parameter. Read-only.

In addition to the serial number, the string returned by this parameter can be used to identify a device before connecting to it.

DEF_MCA_PARAM_SERIAL_NUMBER

Device serial number. String parameter. Read-only. Always 8 characters; the first two digits can be used to determine the device type:

Device code	Device type
01	bMCA USB
02	bMCA Ethernet
03	bPAD
04	bPAD+ and early bPAD-VR devices
05	bPAD-VR
06	Topaz-Pico
07	bPAC
08	Desktop MCA
09	bNdisc
10	Topaz-X

In addition to the device name, the serial number can be used to identify a device before connecting to it.

When writing an application that supports several *classes* of devices (e.g. Single- and Multi-Channel Analyzers), the device type obtained as described above can be used to determine the feature set supported by the specific device the application is connected to.

DEF_MCA_PARAM_FW_VERSION

Device firmware version. String parameter. Read-only. Typically in the form **V*n.m.rrr***, where ***n*** represents the major version number, ***m*** the minor version number and ***rrr*** the revision level.

Generally this parameter is used for informational purposes only. If some function is not supported by the specific firmware version of the device, the device itself and/or the access library will return an appropriate error message. If a function or parameter requires some type of translation because of firmware variations, the access library will normally perform the appropriate action.

DEF_MCA_PARAM_FW_DATE

Release date of the device firmware. String parameter. Read-only. Format is eight-digit ***YYYYMMDD***, where ***YYYY*** represents the year, ***MM*** the month (1-12) and ***DD*** the day (1-31).

Devices are generally delivered from the factory with a firmware date older than the production date. A newer date indicates an upgraded device. This parameter is intended for informational purposes only.

DEF_MCA_PARAM_PROD_DATE

Production date of the device. String parameter. Read-only. Format is eight-digit ***YYYYMMDD***, where ***YYYY*** represents the year, ***MM*** the month (1-12) and ***DD*** the day (1-31).

This parameter is intended for informational purposes only.

DEF_MCA_PARAM_MCA_MODE

MCA operating mode. Integer parameter.

Value	MCA mode
0	Pulse height analysis mode (PHA)
1	Multi-channel scaling mode (MCS)

DEF_MCA_PARAM_NUM_CHANNELS

Number of channels in the PHA spectrum. Integer parameter. Supported values are:

Value
256
512
1024
2048
4096
8192 (Topaz-X only)

The number of channels does not have to be a power of two, as traditionally done in most MCAs. However, it must be a value in the range specified above. Nevertheless, for compatibility with existing analysis programs it is recommended to use a value that is a power of two.

Note that when changing the number of channels for a bMCA, Topaz-Pico or a Topaz-X device, the Digital Gain needs to be adjusted accordingly (see **DEF_MCA_PARAM_DIGITAL_GAIN**).

DEF_MCA_PARAM_PHA_LLD

Lower level discriminator (LLD) value for PHA and MCS modes. Integer parameter, representing the channel number. Possible values from 0 to 4095 (up to 8191 for Topaz-X).

DEF_MCA_PARAM_PHA_ULD

Upper level discriminator (ULD) value for PHA and MCS modes. Integer parameter, representing the channel number. Possible values from 0 to 4095 (up to 8191 for Topaz-X).

DEF_MCA_PARAM_ACQ_PRESET/DEF_MCA_PARAM_PRESET_TIME

Acquisition preset time for PHA mode, or Dwell time for MCS mode. Float parameter. Values are in seconds with 0.005-second resolution (0.1-second resolution for bMCA and Topaz-Pico devices with firmware version **V2.1.xxx** and older).

DEF_MCA_PARAM_ACQ_MODE

Acquisition mode. Integer parameter. **Deprecated**, use **DEF_MCA_PARAM_ACQ_MODE_EX** instead.

Value	Acquisition mode
0	Count to Real Time
1	Count to Live Time

DEF_MCA_PARAM_INP_POLARITY

MCA Input polarity. Integer parameter.

Value	Input polarity
0	Negative polarity
1	Positive polarity

For tube-based MCAs the polarity is typically set to **negative** and does not need to be changed.

DEF_MCA_PARAM_GAIN_DIGITAL

Digital gain. Integer parameter representing the exponent of power of two (2^x) of the digital gain.

Value	Digital amplification
0	x1
1	x2
2	x4
3	x8
4	x16
5	x32
6	x64
7	x128

In order for the device to function properly, the digital gain value must the set to a minimum value that depends of the current number of spectrum channels according to the table below. Failure to do so will cause spectrum truncation.

Spectrum channels	Minimum Digital Gain
256	x8
512	x16
1024	x32
2048	x64
4096	x128
8192	x256

DEF_MCA_PARAM_GAIN_COARSE

Coarse analog gain. Integer parameter representing the exponent of power of two (2^x) of the coarse gain.

Value	Coarse amplification
0	x1
1	x2
2	x4
3	x8
4	x16*
5	x32*
6	x64*
7	x128*

*Topaz-X only. bMCA devices support extended gain only as a custom request. Not supported by Topaz-Pico.

The total gain of the analog stage of the MCA is computed as (coarse gain * fine gain).

DEF_MCA_PARAM_GAIN_FINE

Fine gain. Integerparameter. The fine gain controls the amplification multiplier between 1.0 and 2.0 in 4096 steps (12-bits), thus allowing for a very precise control. A value of zero corresponds to a gain of 1.0, while a value of 4095 is corresponds to approximately 2.0 (exact value depends of component tolerances). Thus, the fine gain amplification is calculated as ((fine gain value /4095) + 1).

Value (examples)	Fine gain amplification
0	x1
4095	x2
2988	x1.730

The total gain of the analog stage of the MCA equals to coarse gain * fine gain.

DEF_MCA_PARAM_RISE_TIME

Rise time. Integer parameter. The parameter value ranges from 0 to 511. To convert the value to microseconds, divide the value by the ADC sampling rate in MHz.

The ADC sample rate can be read via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE**. Devices that do not support such parameter are assumed to have a sample rate of 25MHz (0.04 µs).

DEF_MCA_PARAM_FLAT_TOP

Flat top. Integer parameter. The parameter value ranges from 0 to 511. To convert the value to microseconds, divide the value by the ADC sampling rate in MHz.

The ADC sample rate can be read via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE**. Devices that do not support such parameter are assumed to have a sample rate of 25MHz (0.04 µs).

DEF_MCA_PARAM_PZ_ADJ

Pole-zero adjustment. Integer parameter. To convert the value to microseconds, divide the value by the ADC sampling rate in MHz.

The ADC sample rate can be read via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE**. Devices that do not support such parameter are assumed to have a sample rate of 25MHz (0.04 µs).

DEF_MCA_PARAM_THRESHOLD

Fast-discriminator threshold. Integer parameter. Sets the threshold level that discriminates between noise and useful signal. Typically 6-8 for bMCA and Topaz-Pico with most NaI detectors.

DEF_MCA_PARAM_BLR_ENABLE

Baseline restorer. Integer parameter.

Value	Baseline restorer
0	Disabled
1	Enabled

For the optimal performance of bMCA and Topaz-Pico devices, it is recommended to keep the baseline restorer **always** enabled. In Topaz-X devices the parameter disables only the analog baseline tracking, the digital restorer cannot be disabled.

DEF_MCA_PARAM_PUR_ENABLE

Pile-up rejector. Integer parameter.

Value	Pile-up rejector
0	Disabled
1	Enabled

For best spectral performance the pile-up rejector should be turned on. In those cases where counting rate is more important than spectrum quality the pile-up rejector can be turned off.

DEF_MCA_PARAM_HV_VALUE

High voltage value. Float parameter. 0 to 1500V in 0.1V increments.

DEF_MCA_PARAM_HV_STATUS

High voltage status. Integer parameter.

Value	High volt status
0	High voltage off
1	High voltage on

DEF_MCA_PARAM_SCOPE_TRIG_LVL

Scope trigger level. Integer parameter. 12-bits correspond to scope trigger voltage -1V to +1V.

This parameter is no longer used in devices with firmware **V2.2.001** and newer, in these cases the fast discriminator pulse is used as trigger source for more reliable waveform display.

DEF_MCA_PARAM_SCOPE_WAVEFORM

Selected scope waveform. Integer parameter.

Value	Selected scope waveform
0	Input signal

7	Processed pulse.
1-6	Internal waveform. Internal use only.

Values 1 to 6 are device-dependent, and some devices and/or firmware versions do not support them at all.

DEF_MCA_PARAM_CALIBRATION

Energy calibration coefficients. Array of 3 float parameters.

Array index	Calibration coefficient
0	Offset
1	Slope
2	Quadratic term

The calibration parameters are used by the application software and not directly by the bMCA. They are stored in the device for convenience only.

DEF_MCA_PARAM_MCS_CHANNELS

Number of channels in Multi-Channel Scaling (MCS) mode. Integer parameter.

DEF_MCA_PARAM_MCS_CURRENT_CHANNEL

MCS current channel. Integer parameter. Read only, 0 to DEF_MCA_PARAM_MCS_CHANNELS - 1.

When the MCS acquisition is running, the channel that is currently accumulating events can be retrieved via this parameter.

DEF_MCA_PARAM_PRESET_COUNTS

Preset counts value for preset count mode. Integer parameter.

DEF_MCA_PARAM_CURRENT_COUNTS

Number of counts in the predefined Region of Interest (ROI). Integer parameter.

The ROI limits are defined via **DEF_MCA_PARAM_PHA_PRESET_LLD** and **DEF_MCA_PARAM_PHA_PRESET_ULD** parameters.

DEF_MCA_PARAM_ACQ_MODE_EX

Extended acquisition mode. Integer parameter. Available in devices with firmware **V2.0.003** and newer, supersedes **DEF_MCA_PARAM_ACQ_MODE**.

This parameter is a combination of several status flags. The bits can be set in the code via a set of named bit mask parameters:

Bit parameter	Value	Meaning
DEF_MCA_ACQ_MODE_REAL_TIME	0x00	Preset time is Real Time
DEF_MCA_ACQ_MODE_LIVE_TIME	0x01	Preset time is Live Time
DEF_MCA_ACQ_MODE_TIME	0x02	Acquire until preset time is reached
DEF_MCA_ACQ_MODE_COUNTS	0x04	Acquire until ROI preset counts are reached
DEF_MCA_ACQ_MODE_EXT_CONTROL*	0x08	Acquire until an external signal stops the acquisition

*Topaz-X only.

If both preset time and preset counts bits are cleared, the acquisition will run forever or until stopped by software.

Live Time bit 2^0	Time Preset bit 2^1	Counts Preset bit 2^2	Ext control bit 2^3*	Value	Acquisition mode
0	0	0	0	0	Acquire forever
1	0	0	0	1	Acquire forever
0	1	0	0	2	Acquisition on time preset. Real Time
1	1	0	0	3	Acquisition on time preset. Live Time
0	0	1	0	4	Acquisition on counts preset
1	0	1	0	5	Acquisition on counts preset
0	1	1	0	6	Dual acquisition preset mode: on time (real time) or counts, whatever is reached first
1	1	1	0	7	Dual acquisition preset mode: on time (live time) or counts, whatever is reached first
0	0	0	1	8	Acquire until external signal arrives
1	0	0	1	9	Acquire until external signal arrives
0	1	0	1	10	Acquisition until Real Time expires or external signal arrives
1	1	0	1	11	Acquisition until Live Time expires or external signal arrives
0	0	1	1	12	Acquisition until preset counts reached or external signal arrives
1	0	1	1	13	Acquisition until preset counts reached or external signal arrives
0	1	1	1	14	Acquisition until Real Time expires, preset counts are reached, or external signal arrives
1	1	1	1	15	Acquisition until Live Time expires, preset counts are reached, or external signal arrives

*Topaz-X only.

DEF_MCA_PARAM_PHA_PRESET_LLD

PHA lower level discriminator (LLD) for the lower limit of the ROI that is used for the Preset Counts acquisition mode and/or Multi-Channel Scaling (MCS) mode. Integer parameter.

DEF_MCA_PARAM_PHA_PRESET_ULD

PHA upper level discriminator (LLD) for the upper limit of the ROI that is used for the Preset Counts acquisition mode and/or Multi-Channel Scaling (MCS) mode. Integer parameter.

DEF_MCA_PARAM_EXT_COUNTS

Number of counts from external counter input. Integer parameter.

This parameter is supported only on devices that have an external I/O connector (Topaz-Pico, Topaz-X).

DEF_MCA_PARAM_EXT_IO_MODE

External I/O direction. Integer parameter.

Value	I/O direction
0	Input
Non-0	Output

This parameter is supported only by the Topaz-Pico:

- When set to **input**, the device will count TTL pulses applied to the external I/O line while the acquisition is running; the value can be retrieved via the **DEF_MCA_PARAM_EXT_COUNTS** parameter.
- When set to **output**, the device will output a TTL pulse per event that falls within the ROI determined by **DEF_MCA_PARAM_PHA_PRESET_LLD** and **DEF_MCA_PARAM_PHA_PRESET_ULD**, much like a Single-Channel Analyzer (SCA).

DEF_MCA_PARAM_ICR_COUNTS

Total input counts (Fast Discriminator pulses) since the acquisition was started. Integer parameter.

DEF_MCA_PARAM_HW_PZ_ADJ

Analog Pole/Zero adjust. Integer parameter, 0 to 4095.

In the Topaz-X MCA, the **DEF_MCA_PARAM_PZ_ADJ** must be used *only* to match the tailing time-constant of the analog stage; pole/zero cancellation of the external preamplifier time-constant should be done then via this parameter. For the optimal setting, the output waveform should be monitored using the built-in oscilloscope and the parameter adjusted until pole/zero cancellation occurs. When using TRP preamplifiers, such as for SDD X-ray detectors, this parameter should be set to zero.

DEF_MCA_PARAM_OCR_COUNTS

Total output counts since the acquisition was started. Integer parameter.

That is the number of processed pulses, after discarding pile-ups, whose amplitude falls within the limits of the PHA discriminator (i.e. between **DEF_MCA_PARAM_PHA_LLD** and **DEF_MCA_PARAM_PHA_ULD**). In other words, these are the counts that end up in the spectrum. But note that since **DEF_MCA_PARAM_PHA_ULD** is allowed to be above the current number of channels, **DEF_MCA_PARAM_OCR_COUNTS** may return a number that is higher (albeit normally only slightly) that the total counts in the spectrum returned by the **ReadSpectrum** function.

DEF_MCA_PARAM_OCR_COUNTS, together with **DEF_MCA_PARAM_ICR_COUNTS**, may be used for accurate spectral count rate computations, as an alternative (or complement) to using dead time.

DEF_MCA_PARAM_TRP_COUNTS

Total number of reset pulses since the acquisition was started. Integer parameter.

Supported only by the Topaz-X MCA. Reset pulses are generated only by certain kinds of detector preamplifiers, such as X-ray types.

DEF_MCA_PARAM_PUR_GUARD

Pile-up guard time. Integer parameter. To convert the value to microseconds, divide the value by the ADC sampling rate in MHz. The ADC sample rate can be read via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE**. Devices that do not support such parameter are assumed to have a sample rate of 25MHz (0.04 µs).

The pile-up guard extends the pile-up rejection interval in order to protect an event from being corrupted by anomalies present in the tail of the previous event. Note that increasing the pile-up guard time results in the throughput being reduced. The factory default value is set to approximately 1.1 times the shaping time and provides optimum performance and maximum throughput for most detector applications.

DEF_MCA_PARAM_TRP_GUARD

TRP guard time. Integer parameter. To convert the value to microseconds, divide the value by the ADC sampling rate in MHz. The ADC sample rate can be read via the **DEF_MCA_PARAM_ADC_SAMPLING_RATE**. Devices that do not support such parameter are assumed to have a sample rate of 25MHz (0.04 µs).

Supported only by the Topaz-X MCA. The pile-up guard extends the preamplifier reset pulse interval in order to protect subsequent events from being corrupted by anomalies present in the tail of the reset pulse. Note that increasing the TRP guard time results in the throughput being reduced. The factory default value provides optimum performance and maximum throughput for most X-ray detector applications.

DEF_MCA_PARAM_LT_TRIM

Extended Live-Time adjust factor. Integer parameter.

To parameter allows minor adjustment of the pulse evolution time of the processed digital trapezoid signal in order to increase the accuracy of the dead time. The default factory setting gives good Live Time correction performance for most applications and normally will not require changing.

DEF_MCA_PARAM_ADC_SAMPLING_RATE

Sampling rate in Hertz (Hz) of the fast ADC. Integer parameter.

The sampling interval, in seconds, is the inverse of this parameter, and is used to convert the integer interval values returned by the parameters listed below to microseconds.

Parameter
DEF_MCA_PARAM_RISE_TIME
DEF_MCA_PARAM_FLAT_TOP
DEF_MCA_PARAM_PZ_ADJ
DEF_MCA_PARAM_PUR_GUARD
DEF_MCA_PARAM_TRP_GUARD

The conversion to microseconds is done as (param_value * 1.E6 / DEF_MCA_PARAM_ADC_SAMPLING_RATE).

DEF_MCA_PARAM_TIMING_INFO

Returns information about the acquisition timer parameters. Array of 2 32-bit integer values.

Array index	Contents
0	Internal timer dwell period, in ns
1	Acquisition time granularity, in ns

Topaz-Pico and bMCA devices with firmware versions older than **V2.2.001** do not support this parameter. In this case the following values are assumed:

Parameter	Value
Timer dwell period	20ns
Acquisition time granularity	100,000,000ns (0.1s)

Normally, an application program will not need to use the information provided by this parameter. The values are used internally by the **DIMHW** library in order to perform the necessary acquisition time conversions.

DEF_MCA_PARAM_HV_INFO

Returns info about the high-voltage supply of the device. Not implemented.

DEF_MCA_PARAM_LIST_MODE

Selects between LIST and TLIST mode. Integer parameter.

Value	Mode
0	Simple LIST mode
1	Time-stamped TLIST mode

LIST/TLIST mode is currently supported only by the Topaz-X MCA devices. See the description of the **ReadListEvents** function for the LIST/TLIST data format.

DEF_MCA_PARAM_SCOPE_TRIG_SRC

Selects the trigger source for the built-in scope. Integer parameter.

Value	Trigger source
0	Fast discriminator pulse (default)
1	Start of TRP signal
2	End of TRP signal
3	Event amplitude sampling instant
4	Pile-up event

This parameter is supported only by the Topaz-X MCA device. After arming the scope (see **ArmScope** function), the hardware waits for the selected trigger source event before freezing the waveform in the scope memory. Once that happens, the **GetScopeStatus** function returns 1 indicating that the waveform can be read.

DEF_MCA_PARAM_DIFF_SEL

Selects the analog differentiator time constant. Integer parameter.

Value	Time constant
0	Automatic, according to selected shaping time
1	0.5 µs
2	1.0 µs
3	2.5 µs
4	5.0 µs

This parameter is supported only by the Topaz-X MCA device. Short time constants improve the throughput by avoiding or minimizing the chances of saturation of the analog stages of the MCA by multiple pile-ups, but have a detrimental effect on the spectral resolution. Similarly, larger time constants generally improve signal-to-noise ratio and spectral resolution, but impose a limit on the count rate. The default value of 0 selects a differentiator time constant which is not longer than two times the chosen shaping time. A good compromise for manual setting is 2.5 µs, which gives overall good resolution at input rates well above 500,000 counts per second.

DEF_MCA_PARAM_GPIO1_MODE, DEF_MCA_PARAM_GPIO2_MODE

Selects the function of the corresponding General-Purpose I/O line. Integer value.

This parameter is supported only by the Topaz-X MCA.

The Topaz-X has two TTL-compatible, user-configurable General-Purpose I/O connectors labeled GPIO1 and GPIO2 that provide access to counters, acquisition status and start/stop control, among other functions. The 32-bit parameter value selects the desired mode, where bits 0 to 5 select the actual function and bit 15 selects the signal polarity:

- If bit 15 is not set, polarity is positive (active-high)
- If bit 15 is set, polarity is negative (active-low)

GPIO mode																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	±	0	0	0	0	0	0	0	0	0	GPIO function					

If both GPIO lines are set to the same input function, the Topaz-X will use the logical OR of both input signals to execute the action.

Parameter values for the different modes can be found in the **DIMHW.h** file. Additionally, two pre-defined parameters can be OR'ed with the desired mode to select the polarity: **DEF_MCA_GPIO_POLARITY_POSITIVE** and **DEF_MCA_GPIO_POLARITY_NEGATIVE**:

Parameter name	Value	I/O mode	Description
DEF_MCA_GPIO_NOP	0	Input	No operation
DEF_MCA_GPIO_EXT_COUNTER_1_INPUT	1	Input	External counter 1 input
DEF_MCA_GPIO_EXT_COUNTER_2_INPUT	2	Input	External counter 2 input
DEF_MCA_GPIO_ROI_1_OUTPUT*	3	Output	ROI 1 output (SCA 1)
DEF_MCA_GPIO_ROI_2_OUTPUT*	4	Output	ROI 2 output (SCA 2)
DEF_MCA_GPIO_ICR_OUTPUT	5	Output	ICR (fast discriminator) pulse
DEF_MCA_GPIO_OCR_OUTPUT	6	Output	OCR output
DEF_MCA_GPIO_TRP_INPUT*	7	Input	TRP input
DEF_MCA_GPIO_TRP_OUTPUT	8	Output	TRP output
DEF_MCA_GPIO_HV_INHIBIT_INPUT*	9	Input	HV inhibit input
DEF_MCA_GPIO_HV_STATUS_OUTPUT*	10	Output	HV status output
DEF_MCA_GPIO_ACQ_START_INPUT	11	Input	Acquisition start input
DEF_MCA_GPIO_ACQ_STOP_INPUT	12	Input	Acquisition stop input
DEF_MCA_GPIO_ACQ_START_STOP_INPUT	13	Input	Acquisition start/stop input
DEF_MCA_GPIO_ACQ_SUSPEND_INPUT	14	Input	Acquisition suspend/resume input
DEF_MCA_GPIO_ACQ_STANDBY_OUTPUT	15	Output	Acquisition suspended output
DEF_MCA_GPIO_ACQ_STATUS_OUTPUT	16	Output	Acquisition status output (level)
DEF_MCA_GPIO_ACQ_START_STOP_OUTPUT	17	Output	Acquisition status output (pulses)
DEF_MCA_GPIO_ACQ_WAITING_OUTPUT	18	Output	Acquisition waiting for start
DEF_MCA_GPIO_MCS_CHANNEL_ADV_INPUT*	19	Input	MCS channel advance input
DEF_MCA_GPIO_MCS_CHANNEL_ADV_OUTPUT*	20	Output	MCS channel advance output
DEF_MCA_GPIO_MCS_READY_OUTPUT*	21	Output	MCS ready output
DEF_MCA_GPIO_MCS_SWEEP_ADV_INPUT*	22	Input	MCS sweep advance input
DEF_MCA_GPIO_MCS_SWEEP_ADV_OUTPUT*	23	Output	MCS sweep advance output
DEF_MCA_GPIO_COINCIDENCE_INPUT*	24	Input	Coincidence input (gate)
DEF_MCA_GPIO_ANTICOINCIDENCE_OUTPUT*	25	Output	Anticoincidence output (veto)
DEF_MCA_GPIO_LT_GATE_OUTPUT	26	Output	Live-Time gate output
DEF_MCA_GPIO_TLIST_SYNC_INTPUT*	27	Input	TLIST sync input
DEF_MCA_GPIO_TLIST_SYNC_OUTPUT*	28	Output	TLIST sync output
DEF_MCA_GPIO_PULSE_GEN_1_OUTPUT	29	Output	Pulse rate generator 1 output
DEF_MCA_GPIO_PULSE_GEN_2_OUTPUT	30	Output	Pulse rate generator 2 output
DEF_MCA_GPIO_EXT_USER_1_INPUT*	31	Input	External user input 1
DEF_MCA_GPIO_EXT_USER_2_INPUT*	32	Input	External user input 2

DEF_MCA_GPIO_USER_OUTPUT*	33	Output	User output (level)
DEF_MCA_GPIO_MCS_COUNTER_INPUT*	34	Input	Extra input for MCS counting
DEF_MCA_GPIO_PHA_GROUP_SELECT_INPUT*	35	Input	PHA group selection input
DEF_MCA_GPIO_PHA_GROUP_SELECT_OUTPUT*	36	Output	PHA group selection output

*Means not implement in firmware versions **V3.1.001** and earlier.

For information on how to use external acquisition control with the Topaz-X MCA, refer to the corresponding application note [6].

DEF_MCA_PARAM_NETBIOS_NAME

Not used by current firmware versions.

DEF_MCA_PARAM_DEFAULT_IP

Default IP-address. Array of 3 32-bit integers. Supported only by devices with an Ethernet interface (currently only bMCA/Ethernet).

Array index	Contents
0	IP address
1	Subnet mask
2	Gateway address

DEF_MCA_PARAM_REMOTE_IP

IP address of remote host. Integer parameter. Supported only by the bPAC fast discriminator and counter, not implemented by MCA devices.

The bPAC device uses UDP datagrams to send measurement data to a remote host. By default, the packets are broadcast, so any listening application and/or host can receive them. However, this may cause excessive traffic that adds to network congestions. In addition, broadcast traffic is usually restricted to hosts on the same subnet and therefore generally blocked by routers. Furthermore, there are cases where only one application should be allowed to receive the measurement data and therefore the broadcast mode is simply not desirable. In these cases, this parameter can be used to specify the IP address of the host that will receive the datagrams; in doing so the broadcast mode will be disabled and the UDP packets sent only to the specified host. To enable broadcast mode again, specify an IP address of zero.

DEF_MCA_PARAM_REMOTE_PORT

Port number of remote host. Integer parameter. Supported only by the bPAC fast discriminator and counter, not implemented by MCA devices.

The bPAC device uses UDP datagrams to send measurement data to a remote host, by default via port 8889. Using this parameter, a different port number can be specified. The same port is used for both broadcast and unicast modes (see also the comments for the **DEF_MCA_PARAM_REMOTE_IP** parameter above).

DEF_MCA_PARAM_USERID

Not implemented.

DEF_MCA_PARAM_GROUPID

Not implemented.

DEF_MCA_PARAM_USERDATA

Not implemented.

ERROR CODES RETURNED BY DIMHW FUNCTIONS

Most of the functions of the **DIMHW** library return to the caller a code describing whether the function succeeded or not; in the last case the returned value gives a hint on what went wrong. The following table lists the return codes and their meaning:

Named Error	Description
MCA_SUCCESS	No error. Operation successful.
MCA_ERROR_NOT_CONNECTED	Error. No connection has been established to the MCA.
MCA_ERROR_ALREADY_OPEN	Error. MCA already opened by another user or application.
MCA_ERROR_INVALID_SOCKET	Error. Invalid socket.
MCA_ERROR_BIND_FAILED	Error. Binding to the MCA socket failed.
MCA_ERROR_INVALID_PARAMETER	Error. Invalid parameter type passed.
MCA_ERROR_INVALID_ARGUMENT	Error. Invalid argument passed for parameter.
MCA_ERROR_NOT_SUPPORTED	Error. Unsupported command or function.
MCA_ERROR_CONNECTION_FAILED	Error. The connection to the MCA failed.
MCA_ERROR_CONNECTION_CLOSED	Error. The connection to the MCA was closed.
MCA_ERROR_INVALID_RESPONSE	Error. Invalid response received from MCA.
MCA_ERROR_SEND_FAILED	Error. Sending data to MCA failed.
MCA_ERROR_RECEIVE_FAILED	Error. Receiving data from MCA failed.
MCA_ERROR_MCA_FAILED	Error. MCA failed. Unknown error.
MCA_ERROR_INVALID_DEVICE	Error. Connection to the device is invalid.
MCA_ERROR_INVALID_HANDLE	Error. Invalid device handle passed to the function.
MCA_ERROR_OUT_OF_RANGE	Error. The value passed for a parameter is out of range.

Note that the **DIMHW** library does not perform retries in case of communication errors, nor does it automatically recover from links closed or disconnected. Since such requirements may vary according to the specific application, the task is left to the user program.

REFERENCES

[1] bMCA Data Acquisition and Basic Spectrometry Software. Version 1.7. BrightSpec N.V, Sept 2013.

[2] bMCA Device Technical Description. BrightSpec N.V., January 2013.

[3] bMCA-Ethernet. Device Technical Specifications. Version 1.2. BrightSpec N.V., February 2013.

[4] bMCA-USB. Device Technical Specifications. Version 1.5. BrightSpec N.V., February 2013.

[5] BrightSpec N.V. web site http://www.brightspec.be

[6] Topaz-X External Acquisition Control. Application note. BrightSpec N.V., March 2016.