Signal Processing Devices Sweden AB
Teknikringen 6
SE-583 30 Linköping, Sweden

ADQ-API

User Guide

Document Number	Revision	Date	Security class	
08-0214	11671	2013-12-18	Open	1(80)

Author

Stefan Ahlqvist

Printed

Contents

1 OVERVIEW

ADQ-API provides a simple and powerful programming interface to ADQ devices. The programming interface handles all communication with the connected ADQ devices with just a few highly abstracted functions.

ADQ-API consists of these classes:

- **ADQControlUnit** – An object that manages connection between the ADQ devices and the host computer. The ADQControlUnit creates objects of type **ADQDSP, DSU, SDR14, ADQ212**, **ADQ412**, **ADQ108, ADQ208, ADQ1600**,**ADQ112**, **ADQ114** and **ADQ214**.
- **ADQDSP**– An instance of this object is connected to a specific ADQDSP device and handles the communication with it.
- **DSU**– An instance of this object is connected to a specific ADQDSP device and handles the communication with it.
- **SDR14** – An instance of this object is connected to a specific SDR14 device and handles the communication with it.
- **ADQ1600** – An instance of this object is connected to a specific ADQ1600 device and handles the communication with it.
- **ADQ412**– An instance of this object is connected to a specific ADQ412 device and handles the communication with it.
- **ADQ212**– An instance of this object is connected to a specific ADQ212 device and handles the communication with it.
- **ADQ108**– An instance of this object is connected to a specific ADQ108 device and handles the communication with it.
- **ADQ208**– An instance of this object is connected to a specific ADQ208 device and handles the communication with it.
- **ADQ112**– An instance of this object is connected to a specific ADQ112 device and handles the communication with it.
- **ADQ114** – An instance of this object is connected to a specific ADQ114 device and handles the communication with it.
- **ADQ214** – An instance of this object is connected to a specific ADQ214 device and handles the communication with it.

Windows:

These classes are hidden in a dll-file and interfaced via a function set where the user specifies which ADQ device to communicate with. The interface consists of three files:

ADQAPI.lib – This file must be linked to the code project for compilation of the program.

ADQAPI.dll – This dynamic linked library must be located in the same directory as the compiled program or have a proper path for it set up.
When SP Devices software development kit (SDK) is installed, this dll is copied to the windows dll directory and will always be accessible for the computer.

ADQAPI.h – A header file that must be linked to the code project for declaration of the ADQ-API function set. This is used for programming in C/C++. For other languages, it must be modified.

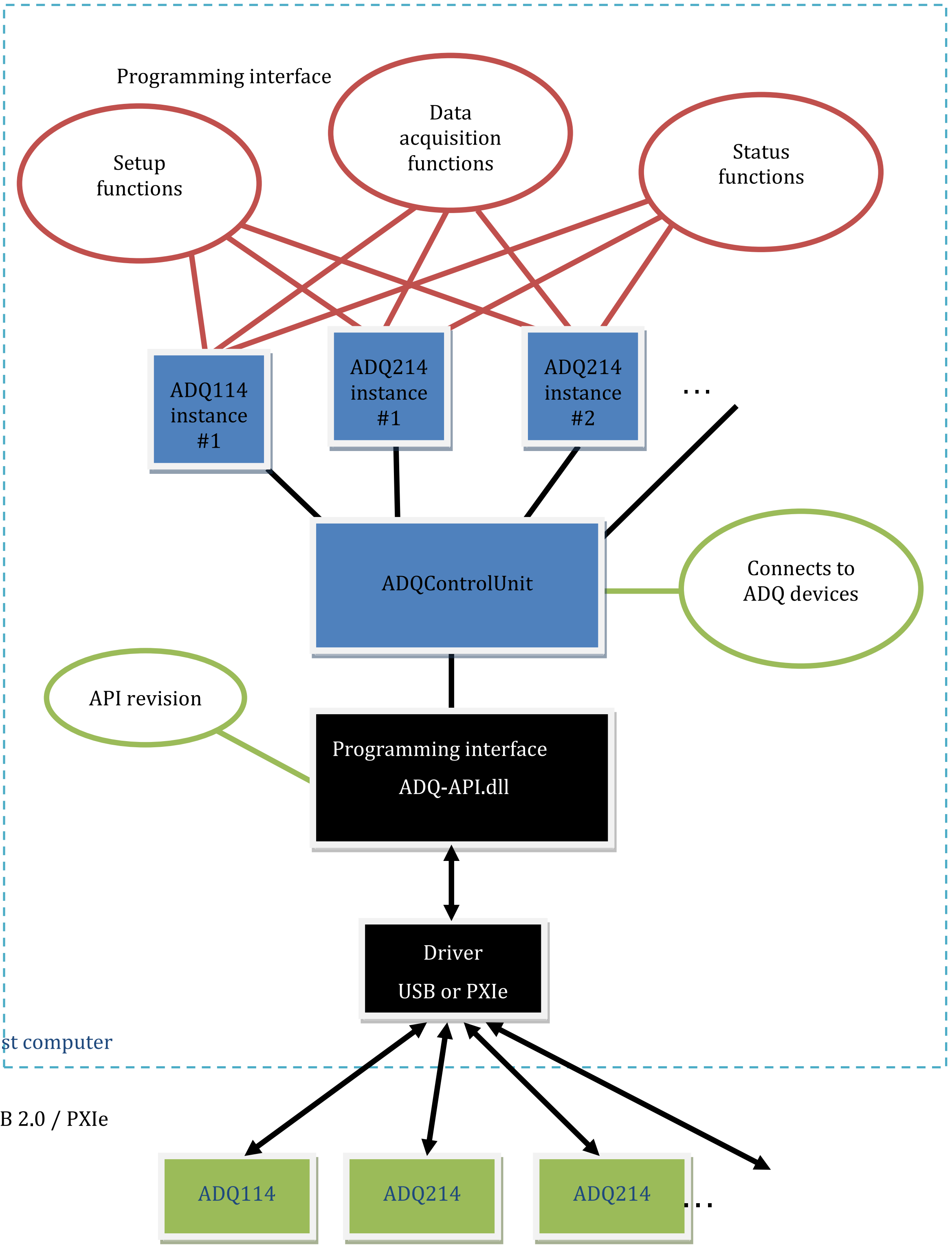
The SDK installation provides three different versions of these files. If the code project is compiled on a on a 32-bit system, the files in the ADQAPI-foldermust be used. If the code project is compiled on a 64-bit system, the files in ADQAPI_64 must be used for 64-bit applications and the files in ADQ_API_32_64 should be used for 32-bit applications. In the latter case, the 32-bit API works for USB but not for e.g. PCIe.

Linux:

The Linux ADQ library providing ADQAPI will follow existing naming conventions and will be called libadq. If installed from a package, the library will be installed in "/usr/lib" and the api header (ADQAPI.h) will be installed in "/usr/include". Instructions on how to install and use the ADQAPI for Linux are found in the installation package. For Linux, the ADQAPI only supports 64-bit systems.

Via the function *"void* CreateADQControlUnit()"* a pointer to an ADQControlUnit object is created and should be used as input to all of the other functions for the API to work properly. Do only call *CreateADQControlUnit**once*** for stable behavior or delete the object with *"void DeleteADQControlUnit(void* adq_cu_ptr)"* before creating another.

2 STRUCTURE


Programming interface
Setup functions
Data acquisition functions
Status functions
ADQ114 instance #1
ADQ214 instance #1
ADQ214 instance #2
…
ADQControlUnit
Connects to ADQ devices
API revision
Programming interface
ADQ-API.dll
Driver
USB or PXIe
Host computer
USB 2.0 / PXIe
ADQ114
ADQ214
ADQ214
…

3 FUNCTIONS OF ADQ-API

The functions of the ADQ-API are categorized into three main sets.

API Specific functions - Purely related to the API itself and not to the operation of digitizers.
ADQControlUnit functions - Interface with the device driver for tasks such as finding and initializing digitizers
ADQ functions - Interface directly with a specific digitizer

In the documentation of the **ADQ functions** it is specified which ADQ device is supported by each function.

The **ADQ functions** are divided into the subcategories:

Setup functions
Data acquisition functions
Status functions.

When standard-C access to the API is desired, all functions except `CreateADQControlUnit` take a `void*` to an ADQControlUnit instance as input. In the following tables it is assumed that only one ADQControlUnit has been created and `adq_cu_ptr` refers to the pointer that points to it .

Via the function `void* CreateADQControlUnit()` a pointer to an ADQControlUnit object is created and should be used as input to all of the other functions for the API to work properly. Only call CreateADQControlUnit **once** for stable behavior.

ADQControlUnit instances may be deleted using:
`void DeleteADQControlUnit(void* adq_cu_ptr)` before creating another.

When C++ access to the API is desired, use the ADQControlUnit_GetADQ function to get a pointer to the ADQInterface object. The ADQInterface object is defined in the ADQAPI.h file. The pointer can then be used directly to call the API functions.

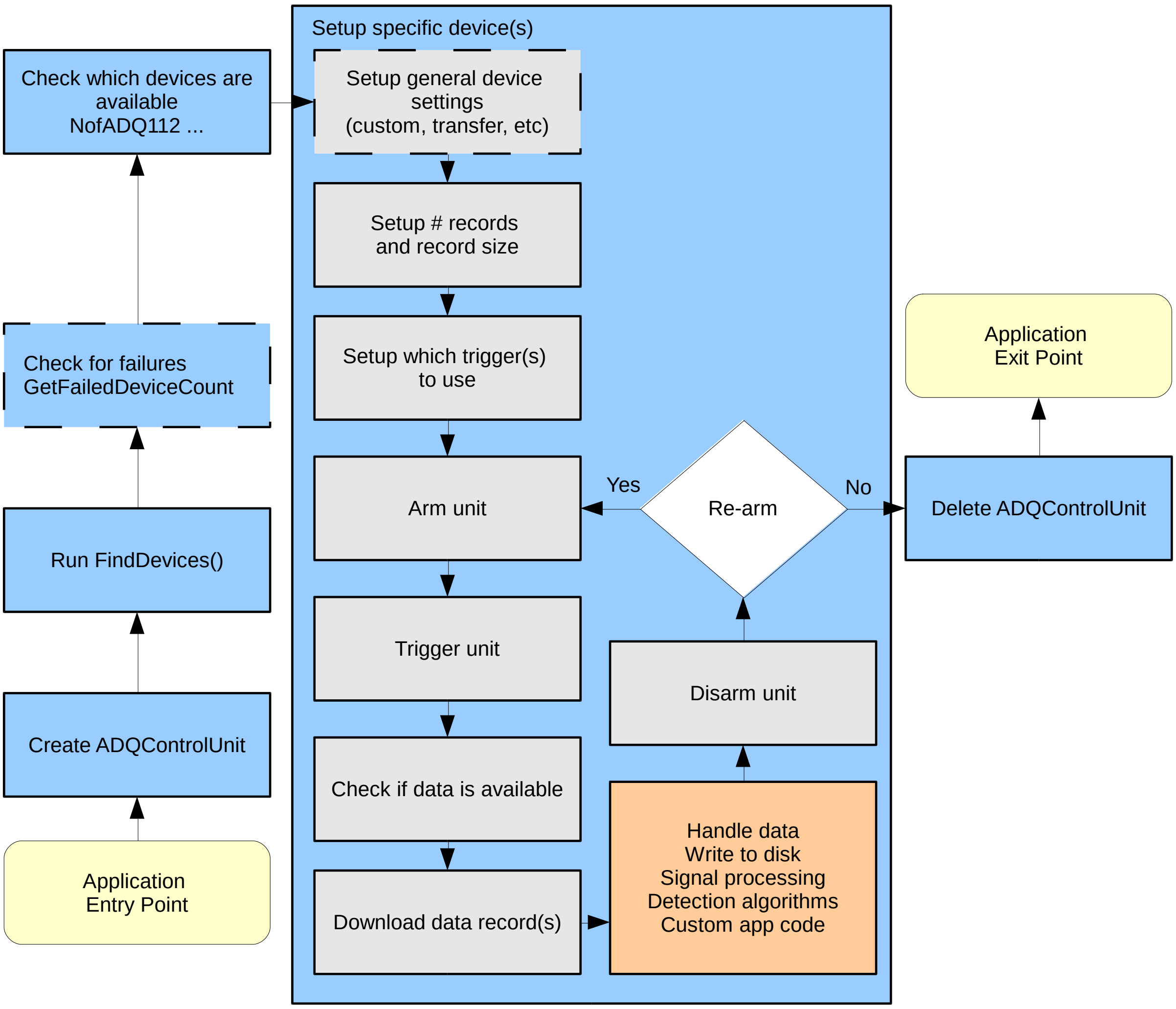
Small C++-style code example:

```
void* ADQCU = NULL;
int TypeOfBoard;
ADQInterface* ADQDevice = NULL;
ADQCU = CreateADQControlUnit();

if((ADQCU != NULL) && (ADQControlUnit_NofADQ(ADQCU) > 0))

{

      ADQDevice = ADQControlUnit_GetADQ(ADQCU, 1);
      ADQDevice->ResetDevice(16);
      TypeOfBoard = ADQDevice->GetADQType();

}

DeleteADQControlUnit(ADQCU);
```

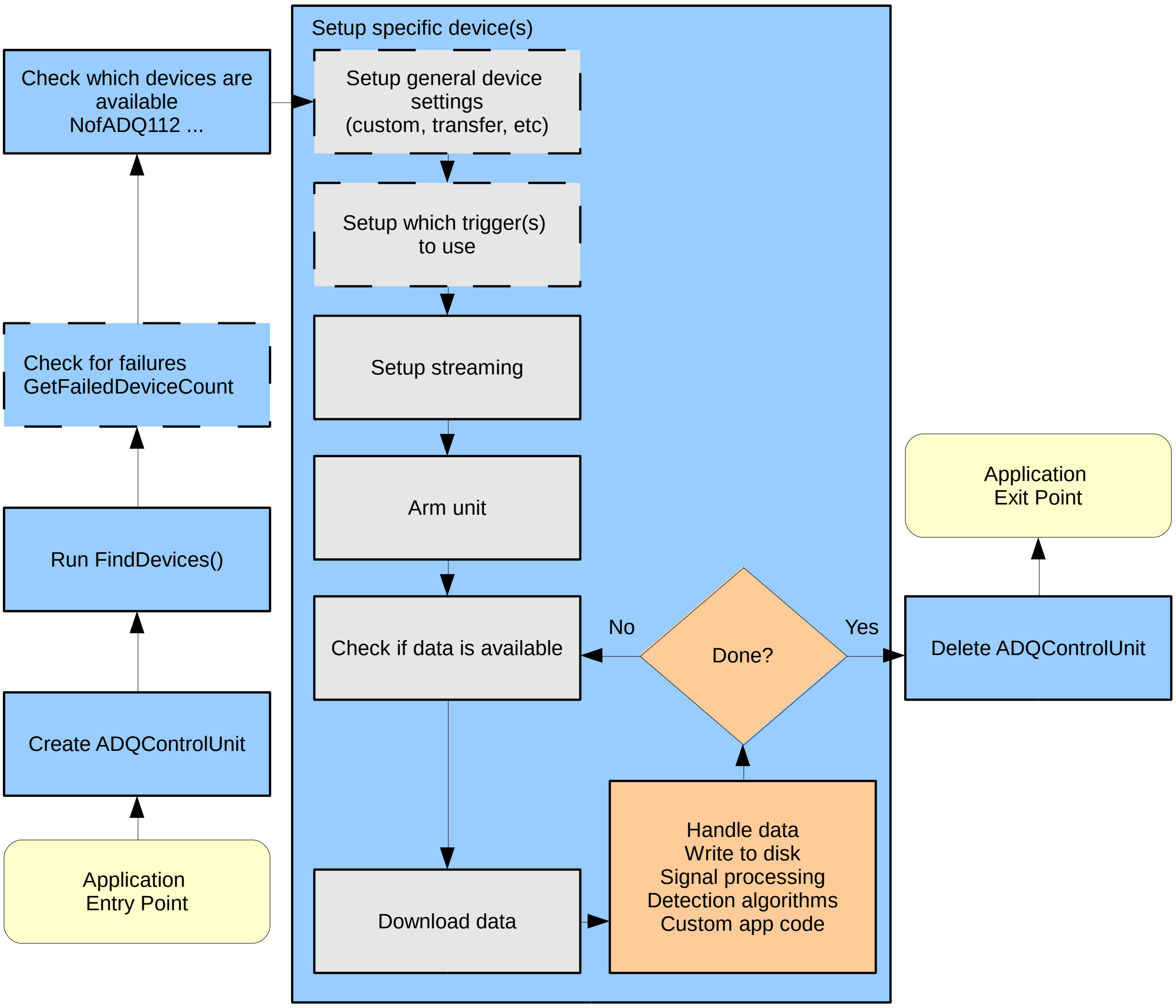
If there are deviations in function naming between the C and C++ API, the naming for the C++ objects retrieved through `ADQControlUnit_GetADQ`, is especially noted in the function document sections as "C++ name".

4 APPLICATION PROGRAMMING FLOWCHART

4.1 Multi-record mode



4.2 Streaming mode


Setup specific device(s)
Check which devices are available NofADQ112 ...
Setup general device settings (custom, transfer, etc)
Setup which trigger(s) to use
Check for failures GetFailedDeviceCount
Setup streaming
Arm unit
Application Exit Point
Run FindDevices()
Check if data is available
No
Done?
Yes
Delete ADQControlUnit
Create ADQControlUnit
Handle data Write to disk Signal processing Detection algorithms Custom app code
Application Entry Point
Download data

5 CODE EXAMPLES

Please see the C_examples and Cpp_examples folders found in the SDK installer directory. There are code examples for, among other things:

- Multi-record mode data collection
- Streaming
- Waveform averaging

6 API REFERENCE

6.1 API Specific Functions

API Specific Function	Description
ADQAPI_GetRevision `int ADQAPI_GetRevision()`	Returns the revision number of the DLL.

6.2 ADQControlUnit Functions

ADQControlUnitFunction	Description
CreateADQControlUnit `void* CreateADQControlUnit()`	Creates an instance of an ADQControlUnit that is capable to find and establish connection to ADQ devices. Returns a pointer to the ADQControlUnit
CreateADQControlUnitWN `void* CreateADQControlUnitWN(HANDLE ReceiverWnd)`	Creates an instance of an ADQControlUnit that is capable to find and establish connection to ADQ devices. Also registers a top window to receive any notifications of device removals. Returns a pointer to the ADQControlUnit
DeleteADQControlUnit `void DeleteADQControlUnit( void* adq_cu_ptr)`	Deletes the instance of ADQControlUnit that *adq_cu_ptr* points to.
ADQControlUnit_FindDevices `int ADQControlUnit_FindDevices( void* adq_cu_ptr)`	Finds all ADQ units connected to the computer and creates/updates a separate list of pointers for all ADQ types. Returns the total number of devices found. Creates new ADQobject(s) if found for the first time. The order of the devices is determined by their USB bus addresses and/or their PXIe address.

ADQControlUnit_ListDevices

```
int ADQControlUnit_ListDevices(
void* adq_cu_ptr, struct ADQInfoListEntry**
retList, unsigned int* retLen)
```

The ListDevices/OpenDeviceInterface/SetupDevice functions are intended as a more versatile replacement for FindDevices.

ListDevices creates a list of available ADQ devices without attempting to boot any firmware or set up any communication channels.

The function requires pointers to a list pointer and a length integer to be provided. The list is then returned as an array which can be indexed from retList[0] to retList[retLen-1], with each entry corresponding to an ADQ device.

The ADQInfoListEntry structure is found in the ADQAPI.h header file and contains all information which can be read non-destructively from the device:

```
struct ADQInfoListEntry
{
  enum ADQHWIFEnum HWIFType;
  enum ADQProductID_Enum ProductID;
  unsigned int VendorID;
  unsigned int AddressField1;
  unsigned int AddressField2;
  char DevFile[64];
  unsigned int DeviceInterfaceOpened;
  unsigned int DeviceSetupCompleted;
};
```

```
enum ADQProductID_Enum {
  PID_ADQ214 = 0x0001,
  PID_ADQ114 = 0x0003,
  PID_ADQ112 = 0x0005,
  PID_SphinxHS = 0x000B,
  PID_SphinxLS = 0x000C,
  PID_ADQ108 = 0x000E,
  PID_ADQDSP = 0x000F,
  PID_SphinxAA14 = 0x0011,
  PID_SphinxAA16 = 0x0012,
  PID_ADQ412 = 0x0014,
  PID_ADQ212 = 0x0015,
  PID_SphinxAA_LS2 = 0x0016,
  PID_SphinxHS_LS2 = 0x0017,
  PID_SDR14 = 0x001B,
  PID_ADQ1600 = 0x001C,
  PID_SphinxXT = 0x001D,
  PID_ADQ208 = 0x001E,
  PID_DSU = 0x001F
};
```

```
enum ADQHWIFEnum {
  HWIF_USB,
  HWIF_PCIE
};
```

ADQControlUnit_OpenDeviceInterface

```
int ADQControlUnit_OpenDeviceInterface(
void* adq_cu_ptr, int ADQInfoListEntryNumber)
```

After running ListDevices and finding an entry of interest in the device list, OpenDeviceInterface is used to open a communications channel towards the device.

The ADQInfoListEntryNumber argument should be the array index of the listdevices entry you want to open, i.e. if you want to open the device corresponding to retList[0], pass 0 to this function.

Using this function will add an ADQ object to the internal lists of the ADQControlUnit. This means that the ADQ will show up when using functions such as ADQControlUnit_GetADQ or ADQControlUnit_NofADQ, etc. Simple tasks such as reading and writing registers can be done at this stage, but data collection and similar requires ADQControlUnit_SetupDevice() to be run also.

Please note that the device number when using GetADQ/NofADQ/etc will not have anything to do with the index number used in this function.

ADQControlUnit_SetupDevice

```
int ADQControlUnit_SetupDevice(
void* adq_cu_ptr, int ADQInfoListEntryNumber)
```

After running ListDevices and having used OpenDeviceInterface to open a communication channel towards a specific device, this function is used to do everything necessary to make the device ready for use, such as initializing API variables, calibrating PLLs, calibrating ADC data interfaces, reseting internal logic, etc. After this, the digitizer is ready for use.

This function takes the same index number as was used with OpenDeviceInterface, i.e. the ListDevices array index corresponding to your device.

Please note that the device number when using GetADQ/NofADQ/etc will not have anything to do with the index number used in this function.

ADQControlUnit_GetFailedDeviceCount

```
int ADQControlUnit_GetFailedDeviceCount(
void* adq_cu_ptr)
```

After a call to ADQControlUnit_FindDevices this function returns the number of units found, which was not possible to start correctly (error reported during start of device)

If zero is returned no devices failed to start.

Cause of failure can be one of:

- Incompatible HW device version
- Power-off during setup phase
- Malfunctioning FPGA code (if used with ADQ Development Kit)

ADQControlUnit_GetLastFailedDeviceError

```
unsigned int
ADQControlUnit_GetLastFailedDeviceError(void*
adq_cu_ptr);
```

Returns the last returned error code from a failing device.

ADQControlUnit_EnableErrorTrace

```
unsigned int
ADQControlUnit_EnableErrorTrace(void*
adq_cu_ptr, int trace_level, const char*
trace_file_dir);
```

Enables log file output from the connected devices, each device opens a separate log file.

trace_level = 0 : No error logging
trace_level = 1 : Error logging
trace_level = 2 : Error and warnings logging
trace_level = 3 : Error, warning, info logging

The log file(s) is opened in the directory specified by the string in the ***trace_file_dir*** argument.

Note: To log errors from a specific device it is often best to disconnect all other ADQ devices to get a single, non-conflicting log file as the result.

ADQControlUnit_GetADQ

C++ only

```
ADQInterface* ADQControlUnit_GetADQ(
void* adq_cu_ptr, int adq_num)
```

Returns a pointer to the ADQInterface object for the corresponding ADQ.Used for C++ interfacing.

ADQControlUnit_NofADQ

```
int ADQControlUnit_NofADQ(
void* adq_cu_ptr)
```

Returns the number of ADQ devices found, any type.

ADQControlUnit_NofADQDSP

```
int ADQControlUnit_NofADQDSP(
void* adq_cu_ptr)
```

Returns the number of ADQDSP devices found.

ADQControlUnit_NofADQ412

```
int ADQControlUnit_NofADQ412(
void* adq_cu_ptr)
```

Returns the number of ADQ412 devices found.

ADQControlUnit_NofADQ212

```
int ADQControlUnit_NofADQ212(
void* adq_cu_ptr)
```

Returns the number of ADQ212 devices found.

ADQControlUnit_NofADQ108

```
int ADQControlUnit_NofADQ108(
void* adq_cu_ptr)
```

Returns the number of ADQ108 devices found.

ADQControlUnit_NofADQ208

```
int ADQControlUnit_NofADQ208(
void* adq_cu_ptr)
```

Returns the number of ADQ208 devices found.

ADQControlUnit_NofADQ112

```
int ADQControlUnit_NofADQ112(
void* adq_cu_ptr)
```

Returns the number of ADQ112 devices found.

ADQControlUnit_NofADQ114

```
int ADQControlUnit_NofADQ114(
void* adq_cu_ptr)
```

Returns the number of ADQ114 devices found.

ADQControlUnit_NofADQ1600

```
int ADQControlUnit_NofADQ1600(
void* adq_cu_ptr)
```

Returns the number of ADQ1600 devices found.

ADQControlUnit_NofSDR14

```
int ADQControlUnit_NofSDR14(
void* adq_cu_ptr)
```

Returns the number of SDR14 devices found.

ADQControlUnit_NofADQ214

```
int ADQControlUnit_NofADQ214(
void* adq_cu_ptr)
```

Returns the number of ADQ214 devices found.

ADQControlUnit_DeleteADQDSP

```
void ADQControlUnit_DeleteADQDSP(
void* adq_cu_ptr, int adqdsp_n)
```

Deletes the ADQDSP object of number adqdsp_n.

Note: This function will rearrange the list of ADQDSPs and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQDSP

ADQControlUnit_DeleteADQ412

```
void ADQControlUnit_DeleteADQ412(
void* adq_cu_ptr, int adq412_n)
```

Deletes the ADQ412 object of number adq412_n.

Note: This function will rearrange the list of ADQ412s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ412

ADQControlUnit_DeleteADQ212

```
void ADQControlUnit_DeleteADQ212(
void* adq_cu_ptr, int adq212_n)
```

Deletes the ADQ212 object of number adq212_n.

Note: This function will rearrange the list of ADQ212s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ212

ADQControlUnit_DeleteADQ108

```
void ADQControlUnit_DeleteADQ108(
void* adq_cu_ptr, int adq108_n)
```

Deletes the ADQ108 object of number adq108_n.

Note: This function will rearrange the list of ADQ108s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ108

ADQControlUnit_DeleteADQ208

```
void ADQControlUnit_DeleteADQ208(
void* adq_cu_ptr, int adq208_n)
```

Deletes the ADQ208 object of number adq208_n.

Note: This function will rearrange the list of ADQ208s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ208

ADQControlUnit_DeleteADQ112

```
void ADQControlUnit_DeleteADQ112(
void* adq_cu_ptr, int adq112_n)
```

Deletes the ADQ112 object of number adq112_n.

Note: This function will rearrange the list of ADQ112s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ112

ADQControlUnit_DeleteADQ114

```
void ADQControlUnit_DeleteADQ114(
void* adq_cu_ptr, int adq114_n)
```

Deletes the ADQ114 object of number adq114_n.

Note: This function will rearrange the list of ADQ114s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ114

ADQControlUnit_DeleteADQ1600

```
void ADQControlUnit_DeleteADQ1600(
void* adq_cu_ptr, int adq1600_n)
```

Deletes the ADQ1600 object of number adq1600_n.

Note: This function will rearrange the list of ADQ1600s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ1600

ADQControlUnit_DeleteSDR14

```
void ADQControlUnit_DeleteSDR14(
void* adq_cu_ptr, int sdr14_n)
```

Deletes the SDR14 object of number sdr14_n.

Note: This function will rearrange the list of SDR14s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofSDR14

ADQControlUnit_DeleteADQ214

```
void ADQControlUnit_DeleteADQ214(
void* adq_cu_ptr, int adq214_n)
```

Deletes the ADQ214 object of number adq214_n.

Note: This function will rearrange the list of ADQ214s and a given number for an ADQ device will maybe no longer refer to the same object as before.

1 <= n <= NofADQ214

6.3 ADQ functions

The ADQfunctions handle the communication to the connected ADQ-devices. All of these functions takes “void* adq_cu_ptr, int adqxxx_num” as input. adq_cu_ptr refers to the ADQControlUnit that was used to connect to this ADQ device. adqxxx_num is the number of the specific ADQ-device to interface. This number corresponds to its place in the ADQControlUnits ADQxxx list.

ADQxxx should here be replaced by the name of the ADQ-device that you are interfacing. This may be ADQDSP, DSU, ADQ412, ADQ212, ADQ108, ADQ208, ADQ1600, SDR14, ADQ112, ADQ114 or ADQ214. In the following part of this document it is assumed that you replace ADQxxx with the proper ADQ-device name.

6.3.1 ADQ Setup Functions

Setup Function	Description
Blink **unsigned int ADQxxx_Blink(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, ADQDSP, DSU, SDR14	Identify board with blinking of the green LED on the front panel.
ValidateDll **unsigned int ValidateDll()**	Function for checking that your application is compiled with the correct ADQAPI.h. Only usable with the C++ API. Please use the macro VALIDATE_DLL(ADQInterface* p) that will exit the application on failure or IS_VALID_DLL(ADQInterface* p)that returns 1 on valid dll and 0 otherwise.
ResetDevice **unsigned int ADQxxx_ResetDevice(void* adq_cu_ptr, int adqxxx_num, int resetlevel)** Valid for: ADQ108, ADQ208, ADQ412, ADQ212, ADQ112, ADQ114, ADQ214, SDR14, ADQ1600	Resets the ADQ device. ***resetlevel = 2 => Soft reset, restores to default power-on state [valid for all devices]*** ***resetlevel = 8 =>Soft reset of communication link [valid for all devices]*** ***resetlevel = 16 => Hard reset (hardware device) [only for USB ADQ V5 versions]*** ***Note:*** *resetlevel 16 is only supported on USB devices.* ***Note:*** *After ResetDevice with resetlevel 16 is issued, hardware must be re-enumerated through the ADQControlUnit by issuing FindDevices. This reset makes the connection between the API and the hardware invalid.* Returns 1 for successful operation and 0 for failure.

ReBootADQFromFlash

unsigned int ADQxxx_ReBootADQFromFlash(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ1600, ADQ412, SDR14

Reads the PCIe configuration header from the ADQ, reboots the ADQ, and then writes the PCIe configuration header back to it.

This effectively power-cycles the FPGA of the ADQ. The ADQ must then be re-enumerated by issuing the FindDevices-command.

Returns 1 for successful operation and 0 for failure.

ResetOverheat

unsigned int ADQxxx_ResetOverheat(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ108, ADQ208, ADQ412, SDR14, ADQ1600

Reset the device from an overheat condition. Device will be initiated to a default configuration.

Returns 1 for successful operation and 0 for failure.

RebootCOMFPGAFromSecondaryImage

unsigned int
ADQxxx_RebootCOMFPGAFromSecondaryImage(void*
adq_cu_ptr, int adq214_num)

Valid for: ADQ214

Reload COM FPGA from secondary image

Note:It takes about 5 seconds to complete.

Returns 13 for successful operation and 0 for failure.

RebootALGFPGAFromPrimaryImage

unsigned int
ADQxxx_RebootALGFPGAFromPrimaryImage(void*
adq_cu_ptr, int adq214_num)

Valid for: ADQ214

Reload COM FPGA from secondary image

Note:It takes about 5 seconds to complete.

Returns 13 for successful operation and 0 for failure.

SetSampleDecimation

unsigned int ADQxxx_SetSampleDecimation(
void* adq_cu_ptr, int adqxxx_num,
unsigned int decimationfactor)

Valid for: ADQ214

Enables decimation.

Decimationfactor = 0 => No decimation
Decimationfactor = 1 => Decimation by 2^1=2
Decimationfactor = 2 => Decimation by 2^2=4
…
Decimationfactor = 34 => Decimation by 2^{34}

Note:For decimationfactor>0, set data format to ***ADQ214_DATA_FORMAT_UNPACKED_32BIT***. (See SetDataFormat)

SetTrigLevelResetValue

int ADQxxx_SetTrigLevelResetValue(
void* adq_cu_ptr, int adqxxx_num,
int resetlevel)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, SDR14, ADQ1600

Sets the offset level for which the level trigger shall arm the trigger for detecting rising or falling edges.

Note:A smaller value results in a more sensitive trigger. A larger value suppresses noise better.

Note:*This setting function should rarely be used, as the default value is usually working best.*

Returns 1 for successful operation and 0 for failure.

SetLvlTrigLevel

```
int ADQxxx_SetLvlTrigLevel(
void* adq_cu_ptr, int adqxxx_num,
int level)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14, ADQ1600

Sets the level for which the level trigger shall trig.

ADQ114/214:

-8192 <= level <= 8191 *(14 bit data)*

ADQ112/ADQ412/ADQ212:

-2048 <= level <= 2047 *(12 bit data)*

ADQ108/ADQ208:

-128<= level <= 127 *(8 bit data)*

Other:

-2^31 <= level <= 2^31-1 *(32 bit data)*

Note:*This setting must be re-set after changing sample width, even if level value is unchanged.*

Returns 1 for successful operation and 0 for failure.

SetLvlTrigEdge

```
int ADQxxx_SetLvlTrigEdge(
void* adq_cu_ptr, int adqxxx_num,
int edge)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14, ADQ1600

Set the edge which the level trigger shall trig for.

edge = 1 => Rising edge
edge = 0 => Falling edge

Returns 1 for successful operation and 0 for failure.

SetLvlTrigChannel

```
int ADQxxx_SetLvlTrigChannel(
void* adq_cu_ptr, int adqxxx_num,
int ChannelCode)
```

Valid for: ADQ412, ADQ212, ADQ214, ADQ208, SDR14

Sets the channel for which the level trigger shall correspond to.Channel C and D are only available for ADQ412.

ChannelCode = 0 => None
ChannelCode = 1 => Channel A
ChannelCode = 2 => Channel B
ChannelCode = 4 =>Channel C
ChannelCode = 8 =>Channel D

To trig on multiple channels add the channel code for each individual channel. Examples:

ChannelCode = 10 =>Any of Channel B and D
ChannelCode = 15 =>Any Channel

Note for ADQ412:

When interleaving, enable level trigger for both channels that are interleaved (that is, use ChannelCode = 0, 3, 12 or 15. This is because channel A&B and C&D are interleaved.

Returns 1 for successful operation and 0 for failure.

SetClockSource

```
int ADQxxx_SetClockSource(
void* adq_cu_ptr, int adqxxx_num,
int source)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Set the clock source for the ADQ device.

ADQ108, ADQ208, ADQ112, ADQ114, ADQ214:

```
source = 0 => Internal clock source,
              Internal 10 MHz reference

source = 1 => Internal clock source,
              External 10 MHz reference

source = 2 => External clock source
```

ADQ108, ADQ208:

```
source = 3 => Internal clock source,
              External 10 MHz reference from
              PXIsync
```

ADQxxx-MTCA:

```
source = 4 => Internal clock source,
              external TCLKA backplane
              reference

source = 5 => Internal clock source,
              external TCLKB backplane
              reference
```

NOTE for ADQ112/114/212/214:
When setting external clock source, do not follow with the command to set the pll freq divider because it will reset the source to internal.

SetClockFrequencyMode

```
int ADQxxx_SetClockFrequencyMode(
void* adq_cu_ptr, int adqxxx_num,
int clockmode)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Set the clock frequency mode for the ADQ device. If internal clock and reference is used, this is handled automatically. If external clock or external reference is used, the boardmust be explicitly set in low-frequency mode if required.

```
source = 1 =>High frequency mode (default)

              (External clock range 240-
550MHz)

source = 0 =>Low frequency mode

              (External clock range 35-240MHz)
```

SetInterleavingMode

int ADQxxx_SetInterleavingMode(void* adq_cu_ptr, int adqxxx_num, char mode)

Valid for: ADQ412, ADQ208, ADQ1600

Sets interleaving mode.

ADQ412:

When enabled ADQ412 will use only 2 of the 4 inputs but at doubled sampling rate.

mode = 0 =>Four channel mode (default)

mode = 1 =>Two channel mode, active inputs A / C

mode = 2 =>Two channel mode, active inputs B / D

mode = 3 =>Two channel mode, all inputs active

ADQ208:

When enabled ADQ208 will use only 1 of the 2 inputs but at doubled sampling rate.

mode = 0 => Two channel mode

mode = 1 => One channel mode (default)

SetPllFreqDivider

int ADQxxx_SetPllFreqDivider(void* adq_cu_ptr, int adqxxx_num, int divider)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Sets the divider in the pll and restarts the pll, and checks if it locks properly.

Note: *This function will call SetClockFrequencyMode if the clock source is internal reference.*

Clock frequency to the ADCs and sample rate is calculated by (internal reference is 10Mhz):

2 <= divider <= 20

ADQ214:

$$f_{adc} = f_s = \frac{F_{ref} * 80}{divider}$$

ADQ114:

$$f_{adc} = \frac{F_{ref} * 80}{divider}, \quad f_s = f_{adc} * 2$$

ADQ112:

$$f_{adc} = \frac{F_{ref} * 110}{divider}, \quad f_s = f_{adc} * 2$$

Returns 1 for successful operation and 0 for failure.

Note: *Dividers 18-20 may sometimes fail to get the PLL locked for ADQ114/ADQ214 devices as this renders a clock out of specification for the clocking circuitry in the FPGA. If you require lower sampling rates, please consider using the sample skip function. The function will then return failure.*

SetPll

```
int ADQxxx_SetPll(
void* adq_cu_ptr, int adqxxx_num,
int n_divider, int r_divider,
int vco_divider, int channel_divider)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Sets the dividers in the pll and restarts the pll.

Not all parameters can be changedon all cards. Please look under the specific card below to see how the sample frequency is set.

ADQ214, ADQ212:

$$f_s = f_{adc} = \frac{F_{ref} * n\_divider}{r\_divider * vco\_divider * channel\_divider}$$

ADQ114, ADQ112:

$$f_s = f_{adc} * 2 = \frac{2 * F_{ref} * n\_divider}{r\_divider * vco\_divider * channel\_divider}$$

ADQ108, ADQ208:

$f_s = 40\,MHz * n\_divider/r\_divider$ (if using a 10 MHz reference clock).

ADQ412:

For ADQ412-1G:

$f_s = 5\,MHz * n\_divider/r\_divider$ (if using a 10 MHz reference clock).

For ADQ412-3G and ADQ412-4G:

$f_s = 10\,MHz * n\_divider/r\_divider$ (if using a 10 MHz reference clock).

ADQ1600:

$f_s = 40\,MHz * n\_divider/r\_divider$ (if using a 10 MHz reference clock).

Note for ADQ114, ADQ214, ADQ112, ADQ212:
The limits for the inputs parameters are:

1<= n_divider<= 262175

1<= r_divider <= 16383

2<= vco_divider <= 6

1<= channel_divider <= 32

Notefor ADQ114, ADQ214, ADQ112, ADQ212:
This function will call SetClockFrequencyMode if the clock source is internal reference.

Note for ADQ114, ADQ214, ADQ112, ADQ212:
Frequencies lower than 100 MHz may sometimes fail to get the PLL locked as this renders a clock out of specification for the clocking circuitry in the FPGA. If you require lower sampling rates, please consider using the sample skip function.

Note for ADQ108 and ADQ208: *Frequencies lower than 6000 MHz may sometimes fail to get the PLL locked as this renders a clock out of specification for the clocking circuitry in the FPGA.*

Note for ADQ412 and ADQ1600: *VCO Frequencies lower than 1400 MHz may sometimes fail to get the PLL locked as this renders a clock out of specification for the clocking circuitry in the FPGA.*

Returns 1 for successful operation and 0 for failure.

SetSampleSkip

```
unsigned int ADQxxx_SetSampleSkip(
void* adq_cu_ptr, int adqxxx_num,
unsigned int skipsamples)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ208, ADQ412

Sets up the sample skip function.

skipsamples = 1 =>No samples skipped
skipsamples = N =>Every N:th sample kept

ADQ214/ADQ212 (allowed N):
N = 2, 4, 6, 8, …, 131072

ADQ112/ADQ114 (allowed N):
N = 2, 4, 8, 12, …, 262140

ADQ208, ADQ412 (allowed N):
N = 2, 4, 8, 16, 32, 64, 128

Returns 1 for successful operation and 0 for failure/unsupported N.

SetTriggerMode

```
int ADQxxx_SetTriggerMode(
void* adq_cu_ptr, int adqxxx_num,
int trig_mode)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Sets how the ADQ device shall be trigged.

All devices:

trig_mode = 1 => Software trigger only mode
trig_mode = 2 => External trigger 1 mode
trig_mode = 3 => Level trigger mode
trig_mode = 4 =>Internal trigger mode
trig_mode = 7 => External trigger 2 mode
trig_mode = 8 => External trigger 3 mode

NOTE: External triggers 2 and 3 are not available on all board hardware.

Returns 1 for successful operation and 0 for failure.

Note: *The software trigger is always enabled regardless of mode.*

SetExternTrigEdge

```
int ADQxxx_SetExternTrigEdge(
void* adq_cu_ptr, int adqxxx_num,
int trig_mode)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600, SDR14

Set the edge which the external trigger shall trig for.

edge = 1 => Rising edge
edge = 0 => Falling edge

Returns 1 for successful operation and 0 for failure.

SetExternalTriggerDelay

```
unsigned int ADQxxx_SetExternalTriggerDelay(
void* adq_cu_ptr, int adqxxx_num,
unsigned char delaycycles)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Sets the delay of the external trigger to match the data path. In default configurations this is setup correctly by the API. If there is additional delay in user configured logic, this API call may be used to compensate correctly.

delaycycles (0-61) => Number of data path clock cycles to delay the external trigger.

Note: 1 data path clock cycle is 4 samples on ADQ112/ADQ114 and 2 samples on ADQ214 and ADQ212.

SetInternalTriggerPeriod

```
int ADQxxx_SetInternalTriggerPeriod(
void* adq_cu_ptr, int adqxxx_num,
unsigned int TriggerPeriodClockCycles)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600, SDR14

Sets the period of the internal trigger.

```
ADQ112/ADQ114:
Period =
TriggerPeriodClockCycles*(1/(fclk/4))

ADQ212/ADQ214:
Period =
TriggerPeriodClockCycles*(1/(fclk/2))

ADQ108/ADQ208:
Period =
TriggerPeriodClockCycles*(1/(fclk/32))

ADQ412:
Period =
TriggerPeriodClockCycles*(1/(fclk/2))

ADQ1600:
Period =
TriggerPeriodClockCycles*(1/(fclk/8))

SDR14:
Period =
TriggerPeriodClockCycles*(1/(fclk/4))
```

Example: A value of 200000 on on ADQ114 sampling at default speed of 800MHz gives a 1kHz internal trigger

Returns 1 for successful operation and 0 for failure.

SetInternalTriggerFrequency

```
unsigned int
ADQxxx_SetInternalTriggerFrequency(
void* adq_cu_ptr, int adqxxx_num,
unsigned int Int_Trig_Freq)
```

Valid for: ADQ214, ADQ208, ADQ1600, ADQ412

Set the frequency for the internal trigger directly in Hertz without needing to manually calculate the trigger period.

Int_Trig_Freq is the frequency in Hz

Beware that the desired frequency will only be **approximated**and the approximation also depends on the current sampling frequency in use!

For manual control of the internal frequency period, the command **SetInternalTriggerPeriod** can also be used. **SetInternalTriggerFrequency** is only meant to make it more convenient.

EnableInternalTriggerCounts

```
unsigned int
ADQxxx_EnableInternalTriggerCounts(void*
adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214

Enable the internal trigger counter block so that the number of triggers can be controlled. A number of triggers must be specified with the function call **SetInternalTriggerCounts** enabling alone will not pass thru any triggers if you have not specified how many triggers that will be allowed to pass thru.

DisableInternalTriggerCounts

```
unsigned int
ADQxxx_DisableInternalTriggerCounts(void*
adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214

Disable the internal trigger counter block. This will let the internal trigger block to run freely and all the triggers generated by the internal trigger block will propagate forward as normal.

SetInternalTriggerCounts

unsigned int ADQxxx_SetInternalTriggerCounts(void* adq_cu_ptr, int adqxxx_num, unsigned int trigger_counts)

Valid for: ADQ214

Set the number of triggers that will be allowed to propagate forward from the internal trigger block. This can be used to control the amount of generated triggers to investigate if a trigger has been missed by the data capturing interface.

trigger_counts is the amount of positive trigger edges that be allowed thru.

ClearInternalTriggerCounts

unsigned int ADQxxx_ClearInternalTriggerCounts(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ214

This function will initiate the counting of the number of triggers outputted by the internal trigger block. Every time the function is called the counting will restart from zero. This can be used to control "bursts of triggers" with a certain time interval between each burst.

MultiRecordSetup

unsigned int ADQxxx_MultiRecordSetup(void* adq_cu_ptr, int adqxxx_num, unsigned int NumberOfRecords, unsigned int SamplesPerRecord)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Setups the memory buffers for multi-record mode (multiple triggers) in the ADQ device.

NumberOfRecords is the amount of records you want to setup the device to collect.

SamplesPerRecords is the size of each record.

Note:*The two parameters apply to all available channels at the same time.*

Returns 1 for successful operation and 0 for failure. Failures include trying to allocate more memory than is available.

MultiRecordSetupGP

unsigned int ADQxxx_MultiRecordSetupGP(void* adq_cu_ptr, int adqxxx_num, unsigned int NumberOfRecords, unsigned int SamplesPerRecord, unsigned int* mrinfo)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Setups the memory buffers for multi-record mode (multiple triggers) in the ADQ device.

NumberOfRecords is the amount of records you want to setup the device to collect.

SamplesPerRecords is the size of each record.

mrinfo is a pointer to an area where the API writes resulting settings, for use with for example MemoryDump/MemoryShadow functions. This area must be preallocated to be 10x32-bit integers. The API will write 10x32-bit integer values into this area. Use NULL if mrinfo should not be used.

Note: *The two parameters apply to all available channels at the same time.*

Fields in **mrinfo**:

```
unsigned int [0] = dram_start_addr
unsigned int [1] = dram_end_addr
unsigned int [2] = dram_addr_per_record
unsigned int [3] = dram_bytes_per_addr
unsigned int [4] = setup_records
unsigned int [5] = setup_samples
unsigned int [6] = setup_padded_samples
unsigned int [7] = max_number_of_records
unsigned int [8] = shadow_size
unsigned int [9] = reserved
```

Returns 1 for successful operation and 0 for failure. Failures include trying to allocate more memory than is available.

GetMaxNofRecordsFromNofSamples

```
unsigned int
ADQxxx_GetMaxNofRecordsFromNofSamples(
void* adq_cu_ptr, int adqxxx_num,
unsigned int NofSamples,
unsigned int* MaxNofRecords)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Returns the maximum number of records that can be used in MultiRecordSetup, given a desired number of samples.

The value is returned in the variable pointed to by the MaxNofRecords-pointer.

Returns 1 for successful operation and 0 for failure.

GetMaxNofSamplesFromNofRecords

```
unsigned int
ADQxxx_GetMaxNofSamplesFromNofRecords(
void* adq_cu_ptr, int adqxxx_num,
unsigned int NofSamples,
unsigned int* MaxNofSamples)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Returns the maximum number of samples that can be used in MultiRecordSetup, given a desired number of records.

The value is returned in the variable pointed to by the MaxNofSamples-pointer.

Returns 1 for successful operation and 0 for failure.

MultiRecordClose

```
unsigned int ADQxxx_MultiRecordClose(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Closes multi-record mode and returns the ADQ device to single record mode.

Returns 1 for successful operation and 0 for failure.

SetStreamStatus

```
int ADQxxx_SetStreamStatus(
void* adq_cu_ptr, int adqxxx_num,
unsigned int status)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600, SDR14

Control streaming mode.

Use the following macros to control streaming mode.

ADQ214 & ADQ212:

ADQ214_STREAM_DISABLED
ADQ214_STREAM_ENABLED_BOTH
ADQ214_STREAM_ENABLED_A
ADQ214_STREAM_ENABLED_B

ADQ114 & ADQ112:

ADQxxx_STREAM_DISABLED
ADQxxx_STREAM_ENABLED

If you want streaming to wait for a trigger after arming – or the above function with the macro ADQxxx_STREAM_WAIT_FOR_TRIGGER

ADQ108, ADQ208, ADQ1600, ADQ412, SDR14:

0x0 (stream disabled)
0x1 (stream enabled)
0x9 (redirect data to DRAM)

Note: When stream status is set to 0x1 and ArmTrigger is executed, data will be streamed immediately and the user application must start emptying with the API command CollectDataNextPage. When stream status is set to 0x9, the DRAM may be emptied using the MemoryDump function after setting stream status to 0x0. Stream mode 0x9 Requires firmware revision 12920 or newer.

Returns 1 for successful operation and 0 for failure.

SetPreTrigSamples

```
int ADQxxx_SetPreTrigSamples(
void* adq_cu_ptr, int adqxxx_num,
unsigned int PreTrigSamples)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Sets the size of the pretrigger buffer.

The granularity of this buffer depends on the product type:

ADQ412:	8 samples (non-interleaved)
	16 samples (interleaved)
ADQ1600:	8 samples
ADQ108:	32 samples
ADQ208:	16 samples (non-interleaved)
	32 samples (interleaved)
SDR14:	8 samples (non-interleaved)
	16 samples (interleaved)
ADQ214:	2 samples (non-interleaved)
	4 samples (interleaved)
ADQ212:	2 samples (non-interleaved)
	4 samples (interleaved)
ADQ114:	2 samples
ADQ112:	2 samples

This means that any pretrigsample size will be rounded UP by the granularity.

For example on ADQ412 in non-interleaved mode; if you set pretrigsamples to 5, it will automatically be rounded up to 8 samples and if you instead set it to 9, it will be rounded up to 16 samples.

0 <= PreTrigSamples <= BufferSize/RecordSize*

Returns 1 for successful operation and 0 for failure.

*BufferSize is the buffer size set by ADQxxx_SetBufferSize. Posttrig data will be filled up in the rest of the buffer.In Multi-Record mode the size is set by RecordSize, i.e. SamplesPerRecord.

*When using this function, TriggerHoldOffSamples is automatically reset to zero.

SetTriggerHoldOffSamples

```
int ADQxxx_SetTriggerHoldOffSamples(
void* adq_cu_ptr, int adqxxx_num,
unsigned int TriggerHoldOffSamples)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Sets the number of samples to wait for acquiring data after the trigger.

***Note:** per channel if applicable (ADQ214).*

0 <= TriggerHoldOffSamples<= 2^31

Returns 1 for successful operation and 0 for failure.

*When using this function PreTrigSamples is automatically reset to zero. All data in the buffer will be acquired after the holdoff.

*For ADQ112/ADQ114 the effective granularity is 4 samples. For ADQ214 the effective granularity is 2 samples. For ADQ108 the effective granularity is 32 samples.

SetDataFormat

unsigned int ADQxxx_SetDataFormat(
void* adq_cu_ptr, int adqxxx_num,
unsigned int format)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14

Sets the sample format.

This function will call ADQxxx_SetNofBits, ADQxxx_SetSampleWidth and ADQxxx_WriteAlgoRegister and set all parameters needed for a sample width and/or alignment change.

Use the following macros for setting a specific sample format:

ADQ214& ADQ114:

ADQxxx_DATA_FORMAT_PACKED_14BIT
ADQxxx_DATA_FORMAT_UNPACKED_14BIT
ADQxxx_DATA_FORMAT_UNPACKED_16BIT
ADQxxx_DATA_FORMAT_UNPACKED_32BIT

ADQ112 & ADQ212:

ADQx12_DATA_FORMAT_PACKED_12BIT
ADQx12_DATA_FORMAT_UNPACKED_12BIT
ADQx12_DATA_FORMAT_UNPACKED_16BIT
ADQx12_DATA_FORMAT_UNPACKED_32BIT

ADQ108& ADQ208:

0 = ADQ108_DATA_FORMAT_PACKED_8BIT
2 = ADQ108_DATA_FORMAT_UNPACKED_16BIT
3 = ADQ108_DATA_FORMAT_UNPACKED_32BIT

ADQ412:

0 = ADQ412_DATA_FORMAT_PACKED_12BIT
1 = ADQ412_DATA_FORMAT_UNPACKED_12BIT
2 = ADQ412_DATA_FORMAT_UNPACKED_16BIT
3 = ADQ412_DATA_FORMAT_UNPACKED_32BIT

ADQ1600 & SDR14:

0 = XXXX_DATA_FORMAT_PACKED_16BIT
1 = XXXX_DATA_FORMAT_UNPACKED_16BIT
3 = XXXX_DATA_FORMAT_UNPACKED_32BIT

The packed format will configure the ADQ to store samples for minimal memory footprint, unpacking after transfer to the host PC is done automatically when using single or multi-record mode. Using streaming mode, unpacking will not be done and is not recommended for use.

Unpacked mode should be used for streaming, this configures the ADQ to store samples padded to 16 bits. 12 & 14 bit modes are stored with sign-extended MSBs. 16 bit mode is stored with zero-padded LSBs.

Unpacked 32 bit mode is used for decimation data, data is stored with zero-padded LSBs.

Returns 1 for successful operation and 0 for failure.

SetDirectionTrig

int ADQxxx_SetDirectionTrig(
void* adq_cu_ptr, int adqxxx_num,
int direction)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600

Sets the direction of the trig connector.

direction = 0 -> Input

direction = 1 -> Output: Data from WriteTrig calls

direction = 5 -> Output: a positive pulse for each trigger accepted (ignore WriteTrig calls) (not valid for ADQ108/ADQ208)

Returns 1 for successful operation and 0 for failure.

SetConfigurationTrig

```
int ADQxxx_SetConfigurationTrig(
void* adq_cu_ptr, int adqxxx_num,
unsigned int mode, unsigned int pulselength,
unsigned int invertoutput)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ1600, ADQ412

Sets the configuration of the trig connector. When issued, this overrides any previous calls to SetDirectionTrig.

Mode is one of:

0x00: Trigger set to input (default)
0x01: WriteTrig() data
0x05: Trigger state (See app note)
0x11: Trigger event *) **)
0x19: Level trigger *) **)
0x41: Internal trigger 50% duty cycle
0x45: Internal trigger *)

*) Pulselength sets the length of the output pulse in nanoseconds when trigger connector is used as output.Minimum is 20ns (14.4ns for ADQ112/ADQ212) and maximum is 5100 ns (3672ns for ADQ112/ADQ212). Default is minimum pulse length.

If invertoutput is 1, the output will be inverted.

If mode is OR:ed with bit 5 (mode | 0x20) the special GPIO trigger block will be activated. Triggers are then blocked with an active high input on GPIO connector pin 5.

**)Wired OR between units, set WriteTrig(1)

Returns 1 for successful operation and 0 for failure.

WriteTrig

```
int ADQxxx_WriteTrig(
void* adq_cu_ptr, int adqxxx_num,
int level)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600

Sets the output level for the trig output.

level = 0-> low

level = 1 -> high

Returns 1 for successful operation and 0 for failure.

SetDirectionGPIO

```
int ADQxxx_SetDirectionGPIO(
void* adq_cu_ptr, int adqxxx_num,
unsigned int direction,
unsigned int mask)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14

ADQ112 ADQ114 ADQ212 ADQ214 Before FPGA#2 revision 3991:

Only GPIO pin#5 has GPIO function. Sets the direction of the GPIO pin#5 by the bits in **direction** and **mask**.

direction[4] = GPIO pin 5

0 = input (default)

1 = output

ADQ112 ADQ114 ADQ212 ADQ214 after and including FPGA#2 revision 3991; ADQ412, ADQ108, ADQ208 all versions:

Sets the direction of the GPIO pins by the bits in **direction** and **mask**.

direction[0] = GPIO pin 1

direction[1] = GPIO pin 2

direction[2] = GPIO pin 3

direction[3] = GPIO pin 4

direction[4] = GPIO pin 5

0 = input (default)

1 = output

***Note:** The mask performs a negative mask, i.e. only the bits that are zero in the mask will be written.*

Returns 1 for successful operation and 0 for failure.

WriteGPIO

int ADQxxx_WriteGPIO(
void* adq_cu_ptr, int adqxxx_num,
unsigned int data,
unsigned int mask)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14

ADQ112 ADQ114 ADQ212 ADQ214 Before FPGA#2 revision 3991:

Sets the state of the GPIO pins with output capability by the bits in **data** and **mask**.

Out pin 4 = data[0]

Out pin 3 = data[1]

GPIO pin 5 = data[4]

ADQ112 ADQ114 ADQ212 ADQ214 after and including FPGA#2 revision 3991;
ADQ412, ADQ108, ADQ208 all versions:

Sets the output of the GPIO pins by the bits in **data** and **mask**.

GPIO pin 0 = data[0]

GPIO pin 1 = data[1]

GPIO pin 2 = data[2]

GPIO pin 3 = data[3]

GPIO pin 4 = data[4]

***Note:** The mask performs a negative mask, i.e. only the bits that are zero in the mask will be written.*

***Note:**Use SetDirectionGPIO to set the pin as output or input.*

Returns 1 for successful operation and 0 for failure.

ReadGPIO

unsigned int ADQxxx_ReadGPIO**(**
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, **ADQ208**, ADQ1600, SDR14

ADQ112 ADQ114 ADQ212 ADQ214 Before FPGA#2 revision 3991:

Returns the state of the GPIO pins.

output[2] = In pin 2

output[3] = In pin 1

output[5] = GPIO pin 5

Where ***output***is the returned value.

ADQ112 ADQ114 ADQ212 ADQ214 after and including FPGA#2 revision 3991;
ADQ412, ADQ108, ADQ208 all versions:

Returns the state of the GPIO pins.

data[0] = GPIO pin 1

data[1] = GPIO pin 2

data[2] = GPIO pin 3

data[3] = GPIO pin 4

data[4] = GPIO pin 5

Where ***output***is the returned value.

EnableClockRefOut

```
unsigned int ADQxxx_EnableClockRefOut(
void* adq_cu_ptr, char enable)
```

Valid for: ADQ108, ADQ208, ADQ412, ADQ1600, SDR14

Enables or disables clock reference output.

enable = 1 =>Clock reference output enabled

enable = 0 => Clock reference output disabled

Returns 1 for successful operation and 0 for failure.

ReadRegister

```
unsigned int ADQxxx_ReadRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Reads a 32 bit word from the FPGA register space. For V5 products, only the Comm FPGA is reachable by this function.

Address space is 32 bits, word length is 32 bits.

Returns the read data.

WriteRegister

```
unsigned int ADQxxx_WriteRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr,
int mask,
int data)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Writes a masked 32 bit word, to the FPGA address space. For V5 products, only the Comm FPGA is reachable by this function.

Address space is 32 bits, word length is 32 bits.

Note: *The mask performs a negative mask, i.e. only the bits that are zero in the mask will be written.*

Returns the answer from the FPGA depending on the register written.

ReadUserRegister

```
unsigned int ADQxxx_ReadUserRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr, unsigned int* retval)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Reads one of the 32-bit user logic output registers.

Returns 1 for success, 0 for failure.

The read data is returned via the retval pointer.

WriteUserRegister

```
unsigned int ADQxxx_WriteUserRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr, int mask, int data,
unsigned int* retval)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Performs a masked write of a value to one of the 32-bit user logic input registers.

Note: *The mask performs a negative mask, i.e. only the bits that are zero in the mask will be written.*

Returns 1 for success, 0 for failure.

The register data is read out again after the write and returned in the retval pointer. You can use a mask of 0xFFFFFFFF to simply check the current value of an input register without overwriting it.

retval may be set to NULL if readout is not desired.

ReadAlgoRegister

```
unsigned int ADQxxx_ReadAlgoRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Reads a 16 bit word from the ADQ algorithm FPGA register space.

Registers are defined by Algo FPGA code, address space is 15 bits with 16 bit word size.

Returns the read data.

WriteAlgoRegister

```
unsigned int ADQxxx_WriteAlgoRegister(
void* adq_cu_ptr, int adqxxx_num,
int addr,
int data)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Writes **data** to the ADQ algorithm FPGA register **addr**.

Registers are defined by Algo FPGA code, address space is 15 bits with 16 bit word size.

Returns the answer from the FPGA depending on the register written.

SetTrigTimeMode

```
int ADQxxx_SetTrigTimeMode(
void* adq_cu_ptr, int adq114_num,
int TrigTimeMode)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Sets which mode the trig timer should work in.

0 => continuous count

1 => activate sync mode, count sync pulses and reset counter.

ResetTrigTimer

```
int ADQxxx_ResetTrigTimer(
void* adq_cu_ptr, int adqxxx_num,
int TrigTimeRestart)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Reset the trig timer. Restarts the timer by default.

0 => Timer waits for start pulse to start

1 => Timer restarts immediately

SetTestPatternMode

```
unsigned int ADQxxx_SetTestPatternMode(
void* adq_cu_ptr, int adqxxx_num,
int mode)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208

Enables and sets which mode the test pattern mux should work in.

ADQ112/ADQ212/ADQ114/ADQ214:

0 => Normal operational mode (direct data)
1 => Test mode with user constant output
2 => Test mode with 16-bit counter
3-6 => Reserved test modes
7 => Mode for merging GPIO with data (unpacked 16-bit modes only)

SetTestPatternConstant

```
unsigned int ADQxxx_SetTestPatternConstant(
void* adq_cu_ptr, int adqxxx_num,
int value)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ208, ADQ108

Sets the 16-bit constant value used for some of the test pattern modes.

SetAfeSwitch

```
unsigned int ADQxxx_SetAfeSwitch(
void* adq_cu_ptr, int adqxxx_num,
unsigned int afemask)
```

Valid for: ADQ214, ADQ212

Sets the AFE relays and PDWN signals to DC-AFE buffers.

Bitmap of afemask:

0: AFE relay ch A, 0 =>AC-AFE, 1 =>DC-AFE
1: AFE relay ch B, 0 =>AC-AFE, 1 =>DC-AFE
2: DC amp ch A PDWN, 0 => amp OFF, 1 => amp ON
3: LF amp ch B PDWN, 0 => amp OFF, 1 => amp ON

Ex:

0x0000 =>AC-AFE both channels
0x0005 => DC-AFE ch A, AC-AFE ch B
0x000A => AC-AFE ch A, DC-AFE ch B
0x000F => DC-AFE both channels

The ADQ device starts up with afe mode ***0x0000***

Returns 1 for successful command transfer and 0 for failure.

SetGainAndOffset

```
unsigned int ADQxxx_SetGainAndOffset(
void* adq_cu_ptr, int adqxxx_num,
unsigned char Channel, int Gain, int Offset)
```

Valid for: ADQ114, ADQ112, ADQ214, ADQ212, ADQ1600, SDR14

Sets the digital gain and offset which is located directly after the sampling circuit. Note, the settings are relative to the factory calibrated settings. To override this relativeness, set bit 7 of the Channel argument to 1.

Maximum allowed values is 32767 and minimum allowed value is -32768.

Gain is scaled by 10 bits i.e. 1024 corresponds to unity gain.

Offset is scaled by codes i.e. 1 corresponds to 1 ADC code (multiplied by current Gain setting)

6.3.2 ADQ Peripheral Functions

Peripheral Function	Description
ADCCalibrate **int ADQxxx_ADCCalibrate(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ108, ADQ208	Tells on-board ADCs to perform calibration immediately. (Not recommended to use in applications.)
ReadADCCalibration **unsigned int ADQxxx_ReadADCCalibration(void* adq_cu_ptr, int adqxxx_num, unsigned char ADCNo, unsigned short* Calibration)** Valid for: ADQ412	Reads out the internal calibration of one ADC and stores it in the user allocated space ***Calibration***. The same data can be written back with WriteADCCalibration to restore a specific state. (Not recommended to use in applications.)
WriteADCCalibration **unsigned int ADQxxx_WriteADCCalibration(void* adq_cu_ptr, int adqxxx_num, unsigned char ADCNo, unsigned short* Calibration)** Valid for: ADQ412	Reads out the internal calibration of one ADC and stores it in the user allocated space ***Calibration***. (Not recommended to use. If ***Calibration*** contains anything but what has been reported by a ReadADCCalibration call, results are unpredictable.)
ReadEEPROM **unsigned int ADQxxx_ReadEEPROM(void* adq_cu_ptr, int adqxxx_num, int addr)** Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14	Reads one byte from the on-board EEPROM. Returns the read byte.
WriteEEPROM **unsigned int ADQxxx_WriteEEPROM(void* adq_cu_ptr, int adqxxx_num, int addr, int data, int accesscode)** Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14	Writes one byte to the on-board EEPROM. The lower 64 pages (256 byte pages => addr < 16384) are reserved for internal use and requires an accesscode to be given. **data** is 8-bit value. Returns the answer from the comm. FPGA.
ReadEEPROMDB **unsigned int ADQxxx_ReadEEPROMDB(void* adq_cu_ptr, int adqxxx_num, int addr)** Valid for: ADQ412, ADQ108, ADQ208, ADQ1600, SDR14	Reads one byte from the on-board, on the daughterboard, EEPROM. Returns the read byte.

WriteEEPROMDB

```
unsigned int ADQxxx_WriteEEPROMDB(
void* adq_cu_ptr, int adqxxx_num,
int addr,
int data,
int accesscode)
```

Valid for: ADQ412, ADQ108,ADQ208, ADQ1600, SDR14

Writes one byte to the on-board, on the daughterboard, EEPROM. The lower 64 pages (256 byte pages => addr < 16384) are reserved for internal use and requires an accesscode to be given.

data is 8-bit value.

SetDACOffsetVoltage

```
unsigned int SDR14_SetDACOffsetVoltage(
unsigned char channel,
float v)
```

Valid for: SDR14

Sets the common-mode voltage for the DAC outputs.

channel = Output channel, 1 or 2
v = CM voltage, -1.0 to 1.0

SetExtTrigThreshold

```
unsigned int ADQxxx_SetExtTrigThreshold(
unsigned int trignum,
double vthresh)
```

Valid for: ADQ1600

Sets the threshold voltage of the specified external trigger input.

trignum = Trigger number, allowed numbers are hardware dependent (some boards only have trig1, others have 1,2,3, etc).

vthresh = Threshold voltage. 0.5V is default.

TrigoutEnable

```
unsigned int ADQxxx_TrigoutEnable(
unsigned int bitflags)
```

Valid for: ADQ1600

Allows the user to select which trigout connectors to send the trigout output signal to.

bitflags = An asserted bit 0 outputs signal on trigout1, asserted bit 1 outputs signal on trigout2, etc.

HasTrigHardware

```
unsigned int ADQxxx_HasTrigHardware(
unsigned int trignum)
```

Valid for: ADQ1600

Returns 1 if the specified external trigger input exists in the board hardware.

trignum = Trigger number

HasTrigoutHardware

```
unsigned int ADQxxx_HasTrigoutHardware(
unsigned int trignum)
```

Valid for: ADQ1600

Returns 1 if the specified external trigger output exists in the board hardware.

trignum = Trigger number

HasVariableTrigThreshold

```
unsigned int ADQxxx_HasVariableTrigThreshold(
unsigned int trignum)
```

Valid for: ADQ1600

Returns 1 if the specified external trigger input supports variation of the trigger threshold voltage (via the SetExtTrigThreshold command).

trignum = Trigger number

6.3.3 ADQ Data Acquisition Functions

Data Acquisition Function	Description
ArmTrigger **int ADQxxx_ArmTrigger(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	This command must be sent before the ADQ device is allowed to be trigged. When the trigger is armed the ADQ device records a *record* of samples whenever the device is trigged until**NofRecords** records is acquired. ***Note:*** *When the ADQ device is busy recording a record of data, the device will ignore trigs.* ***Note:*** *To rearm the device, you must first call ADQxxx_DisarmTrigger, then ADQxxx_ArmTrigger.* Returns 1 for successful operation and 0 for failure.
DisarmTrigger **int ADQxxx_DisarmTrigger(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Disarms the trigger. The ADQ device cannot be trigged when trigger is disarmed. When the trigger is disarmed the memorycounter is reset, so next time *ADQxxx_ArmTrigger* is called and the device records a *record* of data,this record will overwrite the previous first record. Returns 1 for successful operation and 0 for failure.
SWTrig **int ADQxxx_SWTrig(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Trigs the ADQxxx. Always honored regardless of current trigger mode. Returns 1 for successful operation and 0 for failure. ***Note:****The return value does not tell if the device was actually trigged, just that the command was/was not sent ok to the ADQ device.*
GetWaitingForTrigger **int ADQxxx_GetWaitingForTrigger(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Returns 1 if the ADQ device is waiting for a trigger. 0 else.
GetAcquired **int ADQxxx_GetAcquired(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Returns 1 if the ADQ device has been trigged and data has been acquiredfor one or more its records. 0 else.
GetAcquiredRecords **int ADQxxx_GetAcquiredRecords(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ108	Returns the number of records that have been acquired.

GetAcquiredAll

int ADQxxx_GetAcquiredAll(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14

Returns 1 if the ADQ device has been trigged and data has been acquired for all its records. 0 else.

GetTrigPoint

int ADQxxx_GetTrigPoint(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns the position in the data array where the trig occurred.

GetTriggedCh

int ADQxxx_GetTriggedCh(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ208, SDR14, ADQ214, ADQ212

Returns the channel that which the device was trigged by.

***Return value = 0 => None** (if the device was trigged in software trigger mode)*
Return value = 1 => Channel A
Return value = 2 => Channel B
Return value = 4 => Channel C
Return value = 8 => Channel D

The trigged channel value is updated each time *ADQxxx_CollectRecord* is called.

GetOverflow

int ADQxxx_GetOverflow(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns 1 if an overflow has occurred in the most recent collected record. 0 if not.

6.3.4 ADQ Data Transfer Functions

Data Transfer Function	Description
CollectDataNextPage **int ADQxxx_CollectDataNextPage(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Transfers the data from the internal memory buffers of the physical ADQ to the ADQ-object. **ADQxxx_GetSamplesPerPage(..) samples** are collected. The internal page counter in the ADQxxx is counted forward one step. ***1 <= pages <= ADQxxx_GetMaxPages(..)*** Note: If you want to collect all samples stored, a loop that collects "***ADQxxx_GetMaxPages(..)***" of pages should be written. Returns 1 for successful operation and 0 for failure.
CollectRecord **int ADQxxx_CollectRecord(void* adq_cu_ptr, int adqxxx_num, unsigned int record_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	MultiRecord mode only. Transfers data from the internal memory buffers of the ADQxxx device to the ADQxxx-object in the host computer. Collects the *record* specified by **record_num.** **0 <= record_num <= NofRecords-1**Returns 1 for successful operation and 0 for failure. Data is made available in the buffer pointed to by the GetPtrData
GetData **unsigned int ADQxxx_GetData(void* adq_cu_ptr, int adqxxx_num, void** target_buffers, unsigned int target_buffer_size, unsigned char target_bytes_per_sample, unsigned int StartRecordNumber, unsigned int NumberOfRecords, unsigned char ChannelsMask, unsigned int StartSample, unsigned int nSamples, unsigned char TransferMode)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Collects data from the device. Transfers data from the internal memory buffers in the ADQ device directly to the user-assigned buffers pointed to by **target_buffers**.One buffer for each channel of data.**target_buffers** can therefore be an array of pointers, depending on how many channels the capturing device has. This function is meant to be used together with the function **MultiRecordSetup**. **target_buffer_size**is the size of each buffer. This must be equivalent to the total number of samples for all records on each channel**that you want to transfer**. You might have collected a certain number of records with a certain number of samples for each record on the ADQ device. But you may only want to transfer some of these records to the PC. Thus, **target_buffer_size** should always be: **target_buffer_size = NumberOfRecords*nSamples** **target_bytes_per_sample**is the size of each element in the buffers. This parameter will depend on which data format that is currently used and must be big enough to contain the bit width of the sample. If each sample has a bitwidth of, for example, 14 bits then **target_bytes_per_sample** must have a value of 2. Because 2 bytes (16 bits) is the smallest amount of space that can contain a 14 bits sample. Currently used data format can be obtained by using the **GetDataFormat** function. **StartRecordNumber** is the record number to start collecting data from. This value can be set between zero and up to the parameter

NumberOfRecords, which you had previously used to call the **MultiRecordSetup** function with. For example, if you have set up the ADQ device to collect 20 records. But you are only interested in transferring the last 5 records, **StartRecordNumber** should therefore be set to 14 (record index starts from 0).

NumberOfRecords is the number of records to put in the buffers starting from the record number set by **StartRecordNumber.** The sum:

NumberOfRecords + StartRecordNumber

Must always be smaller than the amount of records that you have collected in your ADQ device.

ChannelsMask is a bit-mask providing a set bit for each channel to be fetched. In the case for ADQ214 for example, which has 2 channels, **ChannelsMask** can be set to 0x1 for fetching data only from channel A, or 0x2 for fecting data only from channel B. A value of 0x3 for this parameter will fetch data from both channels.

StartSample is the starting sample of each record to fetch. Just as you can chose a starting point from which record you want to transfer from an array of records, this parameter allows you chose the starting sample within each record that you want to transfer

nSamples is the number of samples to fetch from each record with the starting point set by **StartSample.**

TransferMode is the transfer mode. Please set to 0x00 for normal data fetch operations.

Note: The buffers pointed to by target_buffers is the users responsibility. If these are not allocated correctly, the API will still write in to these addresses.

Note: GetData is the recommended function for fast record data transfers, rather than using CollectRecord.

Returns 1 for successful operation and 0 for failure.

GetDataWH

```
unsigned int ADQxxx_GetDataWH(
void* adq_cu_ptr, int adqxxx_num,
void** target_buffers,
void* target_headers,
unsigned int target_buffer_size,
unsigned char target_bytes_per_sample,
unsigned int StartRecordNumber,
unsigned int NumberOfRecords,
unsigned char ChannelsMask,
unsigned int StartSample,
unsigned int nSamples,
unsigned char TransferMode)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14

Collects data from the device with headers. See documentation for GetData. The difference is only one added argument, the target destination for header data.

target_headersis the memory location where headers will be written. The total amount of data that will be written is 32 bytes times the number of records to fetch specified in **NumberOfRecords**. If set to NULL no headers will be fetched. Header data for record #0 will be in bytes 0-31 and header data for record #1 will be in bytes 32-63 and so forth.

Returns 1 for successful operation and 0 for failure.

GetDataWHTS

Collects data from the device with headers and timestamps. See documentation for GetData. The difference is only two added arguments, the target

```
unsigned int ADQxxx_GetDataWHTS(
void* adq_cu_ptr, int adqxxx_num,
void** target_buffers,
void* target_headers,
void* target_timestamps,
unsigned int target_buffer_size,
unsigned char target_bytes_per_sample,
unsigned int StartRecordNumber,
unsigned int NumberOfRecords,
unsigned char ChannelsMask,
unsigned int StartSample,
unsigned int nSamples,
unsigned char TransferMode)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14

destination for header data and timestamp data.

target_headersis the memory location where headers will be written. The total amount of data that will be written is 32 bytes times the number of records to fetch specified in **NumberOfRecords**. If set to NULL no headers will be fetched.

target_timestampsis the memory location where timestamps will be written. The total amount of data that will be written is 8 bytes (one int64) times the number of records to fetch specified in **NumberOfRecords**. If set to NULL no timestamps will be fetched.

Returns 1 for successful operation and 0 for failure.

MemoryDump

```
unsigned int ADQxxx_MemoryDump(
void* adq_cu_ptr, int adqxxx_num,
unsigned int StartAddress,
unsigned int EndAddress,
unsigned char* buffer,
unsigned int* bufctr,
unsigned int transfersize)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ112, ADQ114, ADQ214, SDR14, ADQ1600, ADQ208

Transfers data from the internal memory buffers in the ADQ device to the buffer **buffer**. It does not parse data into samples, only transfers raw data into the buffer.

StartAddress and **EndAddress** is defined in 128/256/512-bit (product dependent) segment addresses and specifies which part of the memory that shall be transferred.

StartAddress must be a multiple of 32, k*32

EndAddress must be a multiple of 32, (k*32)-1

EndAddress > StartAddress

The number of bytescollected is stored in ***bufctr**.

The memory space **buffer** used must be preallocated for the correct size which is in.

transfersize is the used transfer size over the interface. If set to NULL, default is used.

Returns 1 for successful operation and 0 for failure.

Note: To retrieve product/settings dependent sizes to know which DRAM addresses to read, you may use the MultiRecordSetupGP function. To parse the data at a later stage use the MemoryShadow function of the API, together with GetData.

MemoryShadow

```
unsigned int ADQxxx_MemoryShadow(
void* adq_cu_ptr, int adqxxx_num,
void* MemoryArea,
unsigned int ByteSize)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ112, ADQ114, ADQ214, SDR14, ADQ1600, ADQ208

Sets the API to use a DRAM shadow (in the PC DRAM) for parsing data rather than accessing the device DRAM directly. This is used together with MemoryDump to separate the tasks of transfer and parsing for higher transfer rates, where parsing is possible to perform offline.

MemoryArea pointer to memory area with **ByteSize** allocated bytes. User is responsible for correct allocation/deallocation of this area. If **MemoryArea** is NULL, the shadow function is deactivated.

Returns 1 for successful operation and 0 for failure.

Note: To retrieve product/settings dependent sizes to know which DRAM addresses to read, you may use the MultiRecordSetupGP function. To parse the data at a later stage use the MemoryShadow function of the API, together with GetData.

SetTransferBuffers

```
unsigned int ADQxxx_SetTransferBuffers(
void* adq_cu_ptr, int adqxxx_num,
unsigned int NumberOfBuffers,
unsigned int BufferSize)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14, ADQDSP, DSU

Sets the number and size of data transfer buffers. Can be used to optimize transfer performance for a specific application. Must be given in multiples of 512 bytes.

***Note:* When setting this value, make sure that the cache size (SetCacheSize) is less or equal to transfer buffer size.**

Note: This setting function should rarely be used, as the default value is working best for most applications.

Note: This function allocates memory in the Windows kernel space.

Returns 1 for successful operation and 0 for failure.

SetTransferTimeout

```
unsigned int ADQxxx_SetTransferTimeout(
void* adq_cu_ptr, int adqxxx_num,
unsigned int TimeoutValue)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14, ADQDSP, DSU

Sets the timeout for data transfers. This is used in situations where certain data amounts are expected over the streaming interface at certain update rates. This value should always be significantly higher than the expected data rate, to avoid problems with the communication link.

TimeoutValue is specified in milliseconds, and the default setting is 1000 ms.

Note: *This setting function should rarely be used, as the default value is working best for most applications.*

Returns 1 for successful operation and 0 for failure.

SetCacheSize

```
unsigned int ADQxxx_SetCacheSize(
void* adq_cu_ptr, int adqxxx_num,
unsigned int CacheSizeInBytes)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14

Sets the cache size of transfer of data. Can be used to optimize transfer performance for a specific application. Must be given in multiples of 1024 bytes.

Note:*When transferring small records one at the time, use a small value.*

Note: *This setting function should rarely be used, as the default value is working best for most applications.*

Note: *The cache is not used when ADQ is in streaming mode.*

Returns 1 for successful operation and 0 for failure.

GetStreamOverflow

```
int ADQxxx_GetStreamOverflow(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14

Gets the FIFO overflow flag of the streaming FIFO.

When this is reported true, data is missing from the stream

GetTransferBufferStatus

```
unsigned int
ADQxxx_GetTransferBufferStatus
void* adq_cu_ptr, int adqxxx_num,
unsigned int* filled)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ412, ADQ1600, SDR14, ADQDSP, DSU

Stores the number of buffers available for transferring in *filled. This function enables the host application to balance the streaming read-out to avoid overflows.

***0** => No buffer can be read-out (call to CollectDataNextPage not allowed)*

***1** – (n_of_buffers) => The number of buffers available.*

Returns 1 for successful operation and 0 for failure.

Note: If the number is *n_of_buffers*, all buffers are filled and result will be overflow if the buffers are not read out.

GetPtrStream

```
void* ADQxxx_GetPtrStream(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ214, ADQ112, ADQ114, ADQ412, ADQ108, ADQ208, ADQ1600, SDR14

Returns a pointer to the data array of the stream.

Size of the data array is available using ADQxxx_GetSamplesPerPage(…) after calling ADQxxx_SetStreamStatus(…).

GetPtrData

```
int* ADQxxx_GetPtrData(
void* adq_cu_ptr, int adqxxx_num, int
channel)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, ADQ112, ADQ114, ADQDSP, SDR14, DSU

Returns a pointer to the data array of a specific channel (A=1, B=2, C=3, D=4).

Channel retrieved = channel

Note: ADQDSP C API call is not using the argument channel

GetPtrDataChA

```
int* ADQxxx_GetPtrDataChA(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214, ADQ212

Returns a pointer to the data array for channel A of the most recent collected record.

Size of the data array = SamplesPerPage

GetPtrDataChB

```
int* ADQxxx_GetPtrDataChB(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214, ADQ212

Returns a pointer to the data array for channel B of the most recent collected record.

Size of the data array = SamplesPerPage

6.3.5 ADQ StatusFunctions

Status Function	Description
GetADQType **int GetADQType()** **C++ only** Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU	Returns an integer describing the device. Typically usable only with the C++ API, when the unit type of the ADQInterface* object can be unknown.
GetErrorVector **int ADQxxx_GetErrorVector(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14	Returns 0 if no error has been detected. Otherwise non-zero. **Bold-face marked** conditions are irreversible and needs a power-cycling. Others may affect functionality in different ways, but the ADQ board will continue to operate. **Bit 0: Board turned off - detected overheat condition** Bit 1: Detected broken contact bridge between FPGA #1 and #2 Bit 3: Detected fan fault All detected error conditions will also cause the front panel STATUS LED to flash slowly.
GetLastError **int ADQxxx_GetLastError(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU	Returns 0 if no error in the API has been detected. Otherwise non-zero. Error codes are listed in section 5.
GetLvlTrigLevel **int ADQxxx_GetLvlTrigLevel(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Returns the level for which the level trigger shall trig. ADQ114/214: ***-8192 <= Return value <= 8191*** *(14 bit data)* ADQ112: ***-2048 <= Return value <= 2047*** *(12 bit data)* ADQ108/ADQ208: ***-128 <= Return value <= 127*** *(8 bit data)* Other: ***-2^31 <= Return value <= 2^31-1*** *(32 bit data)*
GetLvlTrigEdge **int ADQxxx_GetLvlTrigEdge(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ412, ADQ212, ADQ108,ADQ208, ADQ112, ADQ114, ADQ214, ADQ1600, SDR14	Returns the edge for which the level trigger shall trig. ***Return value = 1 => Rising edge*** ***Return value = 0 => Falling edge***

GetLvlTrigChannel

int ADQxxx_GetLvlTrigChannel(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ212, ADQ214, ADQ208, SDR14

Returns the channel for which the level trigger trigs on.

ADQ412, ADQ212, ADQ214, ADQ208:

Return value = 0 => None
Return value = 1 => Channel A
Return value = 2 => Channel B
Return value = 4 =>Channel C
Return value = 8 =>Channel D

SDR14:

Return value = 0 => None
Return value = 3 => Channel A
Return value = 12 => Channel B

To trig on multiple channels add the channel code for each individual channel. Examples for ADQ412:

Return value = 10 =>Both Channel B and D
Return value = 15 =>All Channels

GetSampleSkip

int ADQxxx_GetSampleSkip(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ112, ADQ114, ADQ214, ADQ212

Returns the current value of the sample-skip unit. See SetSampleSkip for explanations of the values.

GetSampleDecimation

int ADQxxx_GetSampleDecimation(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ214

Returns the current value of the sample decimation unit. See SetSampleDecimation for explanations of the values.

GetExternTrigEdge

int ADQxxx_GetExternTrigEdge(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108,ADQ208, ADQ1600, SDR14

Returns the edge for which the external trigger shall trig.

Return value = 1 => Rising edge
Return value = 0 => Falling edge

GetOutputWidth

int ADQxxx_GetOutputWidth
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ212, ADQ214, ADQ114, ADQ112, ADQ108,ADQ208, ADQ1600, SDR14

Returns the width of output data in number of bits.

GetNofChannels

int ADQxxx_GetNofChannels(
void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ212, ADQ214, ADQ114, ADQ112, ADQ108,ADQ208, ADQ1600, SDR14

Returns the number of output channels for the device.

GetPllFreqDivider

```
int ADQxxx_GetPllFreqDivider(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns the PLL-divider.

Clock frequency to the ADCs and sample rate is calculated by:

ADQ214:

$$f_{adc} = f_s = \frac{F_{ref} * 80}{divider}$$

ADQ114:

$$f_{adc} = \frac{F_{ref} * 80}{divider}, \qquad f_s = f_{adc} * 2$$

ADQ112:

$$f_{adc} = \frac{F_{ref} * 110}{divider}, \qquad f_s = f_{adc} * 2$$

2 <= Return value <= 20

GetClockSource

```
int ADQxxx_GetClockSource(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108,ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Returns the clock source for the ADQ device.

Return value = 0 => Internal clock source,
Internal 10 MHz reference

Return value = 1 => Internal clock source,
External 10 MHz reference

Return value = 2 => External clock source

GetGainAndOffset

```
unsigned int ADQxxx_GetGainAndOffset(
void* adq_cu_ptr, int adqxxx_num,
unsigned char Channel, int*Gain, int*
Offset)
```

Valid for: ADQ114, ADQ112, ADQ214, ADQ212, ADQ1600, SDR14

Gets the current digital gain and offset which is located directly after the sampling circuit. Note, the returned settings are relative to the factory calibrated settings. To override this relativeness, set bit 7 of the Channel argument to 1.

Gain and Offset are pointers to 32-bit integers where to write the results.

Maximum allowed values is 32767 and minimum allowed value is -32768.

Gain is scaled by 10 bits i.e. 1024 corresponds to unity gain.

Offset is scaled by codes i.e. 1 corresponds to 1 ADC code (multiplied by current Gain setting)

GetAfeSwitch

```
unsigned int ADQxxx_GetAfeSwitch(
void* adq_cu_ptr, int adqxxx_num,
unsigned char Channel, unsigned char*
afemode)
```

Valid for: ADQ214, ADQ212

Gets the setting of the AFE.

Channel A => Channel = 1
Channel B => Channel = 2

afemode is a pointer to an unsigned char where to write the result.

Output values

afemode == 0 => Signal path in AC mode

afemode == 1 => Signal path in DC mode

GetExternalClockReferenceStatus

unsigned int ADQxxx_GetExternalClockReferenceStatus(void* adq_cu_ptr, int adqxxx_num, unsigned int* extrefstatus)

Valid for: ADQ214, ADQ212, ADQ114, ADQ112

When using an external clock reference, this API returns the status of this reference.

Returned in the user-allocated ***extrefstatus*** (unsigned int)

extrefstatus = 1 => External reference available
extrefstatus = 0 => External reference not detected

GetTriggerMode

int ADQxxx_GetTriggerMode(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108,ADQ208,ADQ112, ADQ114, ADQ214, SDR14

Returns the trigger mode of the ADQ device.

All devices:

trig_mode = 1 => Software trigger only mode
trig_mode = 2 => External trigger mode
trig_mode = 3 => Level trigger mode
trig_mode = 4 => Internal trigger mode

ADQ208:

trig_mode = 7 => External DB1 trigger mode

GetUSBAddress

unsigned int ADQ214_GetUSBAddress(void* adq_cu_ptr, int adq214_num)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208,ADQ112, ADQ114, ADQ214, SDR14

Returns the bus address of Windows USBDI stack. If the ADQ device is connected to the host via a PXIe interface, 0 is returned.

GetPCIeAddress

unsigned int ADQxxx_GetPCIeAddress(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208,ADQ112, ADQ114, ADQ214, SDR14

Returns a specific address PXIe address. If the ADQ device is connected to the host via a USB interface, 0 is returned.

GetRevision

int* ADQxxx_GetRevision(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412,ADQ1600, ADQ212, ADQ108, ADQ208,ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Returns the revision of the ADQ device. Fields 0-2 contain information for FPGA #2(Comm FPGA) and fields 3-5 contain information for FPGA #1 (Alg FPGA). The returned field (int* revision) is 6 positions long and contains:

revision[0 and 3] = revision number

revision[1 and 4]: 0 => SVN Managed
1 => Local Copy

revision[2 and 5]: 0 => SVN Updated
1 => Mixed Revision

Where ***revision***is the returned pointer.

GetBoardSerialNumber

char* ADQxxx_GetBoardSerialNumber(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208,ADQ112, ADQ114, ADQ214, SDR14

Returns the serial number of the ADQ device. The returned field (char* serialno) is 16 positions long and contains a null-terminated string.

GetCardOption

```
const char* ADQxxx_GetCardOption(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, SDR14, ADQ208

Returns a null terminated string containing card option.

Example: "-3G" for ADQ412 specifies ADQ412-3G card option.

GetADQDSPOption

```
const char* ADQxxx_GetADQDSPOption(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, SDR14, ADQ108, ADQ208, ADQDSP, DSU

Returns a null terminated string containing motherboard options.

Example: "-PXIe" for a PXIe form-factor motherboard

GetTriggerInformation

```
int ADQxxx_GetTriggerInformation(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns the enhanced trigger accuracy information.

The bits in the returned value holds the information and is decoded as:

output[0:9] = Reserved for future use

output[10:11] = Enhanced trigger accuracy vector

output[12:31] = Reserved for future use

Where ***output***is the returned value.

Note: *This information is only valid if the ADQ device is set to External trigger mode.*

GetTrigTime

```
unsigned long long ADQxxx_GetTrigTime(
void* adq_cu_ptr,int adq114_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Returns the timestamp counter value. The Result depends on the Trig Time Mode.

SYNC_ON=cycles*2^2+start_val+trig_val

SYNC_OFF=syncs*2^42+cycles*2^2+start_val+trig_val

GetTrigTimeCycles

```
unsigned long long
ADQxxx_GetTrigTimeCycles(
void* adq_cu_ptr, int adq114_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Returns the cycle counter value of the time stamp.

GetTrigTimeSyncs

```
unsigned intADQxxx_GetTrigTimeSyncs(
void* adq_cu_ptr, int adq114_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Returns the sync counter value of the time stamp.

GetTrigTimeStart

```
unsigned int ADQxxx_GetTrigTimeStart(
void* adq_cu_ptr, int adq114_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ1600, SDR14

Returns the start pulse value of the time stamp. It is a two bit value of the start pulse

GetMultiRecordHeader

```
unsigned int* ADQxxx_GetMultiRecordHeader(
void* adq_cu_ptr, int adq114_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns the pointer to the Multi Record Header from the record last collected. The Multi Record Header contains 8 unsigned int values.

GetTemperature

```
unsigned int  ADQxxx_GetTemperature(
void* adq_cu_ptr, int adqxxx_num,
int addr)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208,ADQ112, ADQ114, ADQ214, ADQDSP, SDR14, DSU

Reads and returns the current on-board temperatures.Temperatures are returned as the actual temperature in Celsius times 256.

Addressing 112/114/214/212:

addr = 1: Temperature sensor #1 (comm. FPGA)
addr = 2: Temperature sensor #2 (alg. FPGA)

Addressing 108/412/208/SDR14/DSU/1600/DSP:

addr = 0: Sensor controller local temp
addr = 1: Temperature sensor #1 (ADC0)
addr = 2: Temperature sensor #2 (ADC1)
addr = 3: Temperature sensor #3 (FPGA)
addr = 4: Temperature sensor #4 (PCB)

GetStreamStatus

```
int ADQxxx_GetStreamStatus(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ412, ADQ108, ADQ208,ADQ1600, SDR14

Returns the streaming status.

Return value = 0 => Streaming disabled

Return value = 7 => Streaming of all data enabled

Return value = 3 => Streaming of all data on channel A enabled (valid for ADQ214 only)

Return value = 5 => Streaming of all data on channel B enabled (valid for ADQ214 only)

GetDataFormat

```
unsigned int ADQxxx_GetDataFormat(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208,ADQ212, ADQ112, ADQ114, ADQ214, SDR14

Returns the data format set for the device. See SetDataFormat for an explanation of the values.

GetRecordSize

```
unsigned int ADQxxx_GetRecordSize(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212,ADQ112, ADQ114, ADQ214

MultiRecord mode only.

Returns the record size set in the ADQ device. The returned value is given in number of *samples*.

Note: *per channel if applicable (ADQ214).*

GetNofRecords

```
unsigned int ADQxxx_GetNofRecords(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

MultiRecord mode only.

Returns the number of records set in the ADQ device.

Note: *per channel if applicable (ADQ214).*

GetSamplesPerPage

```
unsigned int ADQxxx_GetSamplesPerPage(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ212, ADQ108, ADQ208, ADQ112, ADQ114, ADQ214, SDR14

Returns the number of samples of each page set in the ADQ device.

Used with CollectDataNextPage/CollectRecord to get information on number of samples per call.

Note: *per channel if applicable.This figure may change when altering the acquisition settings.*

GetBcdDevice

```
unsigned int ADQxxx_GetBcdDevice(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Returns the PCB revision of the ADQ device.

IsPCIeDevice

```
int ADQxxx_IsPCIeDevice(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ1600, ADQ412, SDR14

Returns 1 if the ADQ device is configured for PXIe, else 0.

IsUSBDevice

```
int ADQxxx_IsUSBDevice(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214, ADQ108, ADQ208, ADQ1600, ADQ412, SDR14

Returns 1 if the ADQ device is configured for USB, else 0.

IsAlive

```
unsigned int ADQxxx_IsAlive(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Pings the ADQ unit. Returns 1 if the ADQ device is answering the ping request, else 0.

IsStartedOK

```
unsigned int ADQxxx_IsStartedOK(
void* adq_cu_ptr, int adqxxx_num)
```

C++ only

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, ADQ212, ADQ112, ADQ114, ADQ214, SDR14

Checks if the ADQ unit started correctly. Returns 1 if the ADQ device has been started OK, else 0.

GetNGCPartNumber

```
const char* ADQxxx_GetNGCPartNumber(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, SDR14, ADQDSP, DSU

Read out the part number of the framework NGC or NGC package which was used to build the firmware.

This part number cannot be modified from inside an ADQ DevKit (apart from replacing the NGC files).

The result is returned as a pointer to a null-terminated string, consisting of three three-digit numbers followed by a revision letter. For example:

400-200-002-A

Older firmware revisions do not contain part number registers and will always be read out as 000-000-000-A.

GetUserLogicPartNumber

```
const char* ADQxxx_GetUserLogicPartNumber(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, SDR14, ADQDSP, DSU

Read out the part number of the user logic file which was used to build the firmware.

This part number may be modified from inside the DevKit, using either the set_userlogicpartnumber command while building the DevKit, or by modifying the assignment statements to the registers in the user logic module. This allows the DevKit customer to keep track of different firmware types and revisions.

The result is returned as a pointer to a null-terminated string, consisting of three three-digit numbers followed by a revision letter. For example:

400-013-011-A

Older firmware revisions do not contain part number registers and will always be read out as 000-000-000-A.

GetPCIeLinkWidth

```
Unsigned int ADQxxx_GetPCIeLinkWidth(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ108,ADQ208, SDR14, ADQDSP, DSU

Returns the number of lanes used for the PCIe connection between ADQ and host. If the ADQ is not connected through PCIe, this function returns 0.

GetPCIeLinkRate

```
Unsigned int ADQxxx_GetPCIeLinkRate(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, SDR14, ADQDSP, DSU

Returns the generation ofthe PCIe connection between ADQ and host. If the ADQ is not connected through PCIe, this function returns 0.

GetProductFamily

```
Unsigned int ADQxxx_GetPCIeLinkRate(
void* adq_cu_ptr, int adqxxx_num, unsigned
int* family)
```

Valid for: ADQ412, ADQ1600, ADQ108, ADQ208, SDR14, ADQDSP, DSU, ADQ214, ADQ212, ADQ114, ADQ112

Get the product family number for the digitizer. A V6 digitizer returns the number 6, a V5 digitizer returns the number 5.

Pass a pointer to an unsigned int where the number is to be stored, via the "family" argument.

6.4 ADQ Special Block Functions

Special block functions are not available on all units. These functions relate to specific IP blocks which may be added on some unit types.

6.4.1 Waveform Averaging and Triggered Streaming Block Functions

ADQ Special Block Function	Description
WaveformAveragingSetup **unsigned int ADQxxx_WaveformAveragingSetup(void* adq_cu_ptr, int adqxxx_num, unsigned int NofWaveforms, unsigned int NofSamples,unsigned int NofPreTrigSamples, unsigned int NofHoldOffSamples, unsigned int WaveformAveragingFlags)** Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412, ADQ1600	Sets up the waveform averaging block on the ADQ. Please consult the example for waveform averaging to obtain details of the execution flow. **NofWaveforms** is the number of waveforms to average **NofSamples** is the number of samples to average **NofPreTrigSamples** is the number of pretrigger samples to average **NofHoldOffSamples** is the number of samples to hold off after the trigger event. **WaveformAveragingFlags** specify: 0x0001 Compensate data path for external trigger 0x0002 Compensate data path for level trigger 0x0004 Enable fastest readout 0x0008 Enable medium paced readout 0x0010 Enable slow readout 0x0020 Enable data path for using level trigger 0x0040 Enable the waveform get function 0x0080 Enable automatic readout and arm (Used for streaming continuously) 0x0400 Immediate readout mode 0x1000 Chose channel A as input when running WFA in one channel mode (ADQ214) 0x2000 Chose channel B as input when running WFA in one channel mode (ADQ214) ***Note***: When running in one channel input mode (to gain longer record length) only ONE channel can be chosen. Special custom firmware is required to use this mode. If both channels have been set OR no channel has been set when using such a firmware, default channel will be A. On standard firmware without support for this one channel input mode, these two flags will have no meaning. ***Note***: On ADQ114 and ADQ112 the maximum length waveform is 32k samples and maximum waveform count is 64k. Pretrigger, Holdoff and sample length is chosen by 4 sample increments. ***Note:*** If streaming over USB is used, one should preferably choose a sample size of the waveform that equals a packet size of 512 bytes. Each averaged sample is 4 bytes, therefore sample sizes should be chosen as 128 sample increments. ***Note:*** Enabling the waveform get function will change the transfer settings of the device. ***Note:*** The packet streaming block and waveform averaging block cannot be used at the same time. ***Note:*** Immediate readout is only available on ADQ214, ADQ212, ADQ114 and ADQ112 at the moment.

WaveformAveragingArm

unsigned int ADQxxx_WaveformAveragingArm(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412, ADQ1600

Arms the waveform averaging. After this triggers will be accepted. If automatic readout and arm is turned on, readout will occur once average is done and a new average will restart when readout is done.

WaveformAveragingDisarm

unsigned int ADQxxx_WaveformAveragingDisarm(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412, ADQ1600

Disarms the averaging block and puts it in bypass mode.

WaveformAveragingGetWaveform

unsigned int ADQxxx_WaveformAveragingGetWaveform(void* adq_cu_ptr, int adqxxx_num, int* waveformdata)

Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412, ADQ1600

Gets the entire waveform into assigned memory location. Performs all necessary communication with device to get waveform. This can only be done if readout of data is available, which should be checked by the status function.

Data is returned in the signed 32-bit memory space given to the function. The user is entirely responsible for having allocated this space properly.

Please consult the example for waveform averaging to obtain details of the execution flow.

WaveformAveragingGetStatus

unsigned int ADQxxx_WaveformAveragingGetStatus(void* adq_cu_ptr, int adqxxx_num, unsigned char* ready, unsigned int* nofrecords, unsigned char* inidle)

Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412, ADQ1600

Gets the status of the averaging block.

Returns 1 in **ready** if data readout is available. If **ready** is NULL, this flag will not be read.

Returns the number of accumulated records in **nofrecords**. If **nofrecords** is NULL, this flag will not be read.

Returns the in idle status of WFA in **inidle**. If **inidle** is NULL, this flag will not be read.

WaveformAveragingShutdown

unsigned int ADQxxx_WaveformAveragingShutdown(void* adq_cu_ptr, int adqxxx_num)

Valid for: ADQ214, ADQ212, ADQ112, ADQ114, ADQ412

Issues shut down for waveform averaging. Used to gracefully stop the auto automatic readout and arm feature mode.

After issuing shutdown, please monitor and wait for the in_idle signal of the WaveformAveragingGetStatus command to go high before starting again.

TriggeredStreamingSetupV5

unsigned int ADQxxx_TriggeredStreamingSetupV5(void* adq_cu_ptr, int adqxxx_num, unsigned int SamplePerRecord, unsigned int ArmMode, unsigned int ReadOutSpeed, unsigned int Channel)

Valid for: ADQ214

Setup data acquisition using the streaming interface and trigger every record with the available trigger modes. This function is depending on the waveform averaging block as an intermediate storage space and will only work on firmware with the modified waveform averaging block. Please consult the example for Triggered Streaming for details of the execution flow.

SamplePerRecordis the number of samples per record to be collected. If streaming over USB is used, one should preferably choose number of sample perrecord that equals apacket size of 512 bytes. Each sample is 2 bytes (16 bits), therefore record sizes should be chosen with 256 sample increments.

ArmMode:

0 = Manual re-arm and readout

1 = Auto re-arm and readout

Manual re-arm mode will collect a record, signal to the user that a record has been collected and wait for the user to read out the record. Reading out a record manually is done by calling the function **TriggeredStreamingGetWaveform**. After reading out the acquired record the user must also re-arm the triggered streaming block by calling the function **TriggeredStreamingArm**before a new record can be acquired.

Auto re-arm mode will automatically push the acquired record through the streaming interface and re-arm itself to collect the next record. If the user does not read the data or cannot read the data fast enough, the streaming interface will be overflown. Auto re-arm mode still requires to be armed the first time by calling the function **TriggeredStreamingArm**.

ReadOutSpeed:

0 = Slow readout speed

1 = Medium readout speed

2 = Fast readout speed

ReadOutSpeed controls how fast an acquired record is being pushed into the streaming interface. This is useful if a host system or the hardware interface is too slow to take care of the data produced by the triggered streaming block. Using the USB interface for example, it might be difficult to collect long records with moderate trigger rate without causing overflow. Setting the readout speed to slower mode will overcome this issue. This however will also decrease the overall transfer speed.

Channel parameter specifies from which channel the data will be streamed from.

1 = Channel A

2 = Channel B

Note: **Channel** should be set to 0 on standard firmware which does not support one channel mode

Note: Manual re-arm and readout will change the transfer settings of the device.

Note: Packet streaming and triggered streaming cannot be used at the same time.

TriggeredStreamingArmV5

```
unsigned int
ADQxxx_TriggeredStreamingArmV5(void*
adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214

Arms Triggered Streaming. After this command, triggers will be accepted. If automatic re-arm and readoutis turned on, readout will occur once a record is collected and a new record will be collected when readout is done.

TriggeredStreamingDisarmV5

```
unsigned int
ADQxxx_TriggeredStreamingDisarmV5(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ214

Disarms the Triggered Streaming block and puts it in bypass mode.

TriggeredStreamingGetStatusV5

```
unsigned int
ADQxxx_TriggeredStreamingGetStatusV5(
void* adq_cu_ptr, int adqxxx_num, unsigned
char* ready, unsigned int*
nofrecordscompleted, unsigned char*
in_idle)
```

Valid for: ADQ214

Gets the status of the Triggered Streaming block.

Returns 1 in **ready** if data readout is available. If **ready** is NULL, this flag will not be read.

Returns the number of acquired records in **nofrecordscompleted**. If **nofrecordscompleted** is NULL, this flag will not be read.

Returns the in idle status in **in_idle**. If **in_idle** is NULL, this flag will not be read.

TriggeredStreamingGetWaveformV5

```
unsigned int
ADQxxx_TriggeredStreamingGetWaveformV5(void
* adq_cu_ptr, int adqxxx_num, short*
waveform_data_short)
```

Valid for: ADQ214

Gets the entire record into assigned memory location. Performs all necessary communication with device to get waveform. This can only be done if readout of data is available, which should be checked by the status function.

Data is returned in the signed 16-bit memory space given to the function. The user is entirely responsible for having allocated this space properly.

Please consult the example for Triggered Streaming for details of the execution flow.

HasTriggeredStreamingFunctionality

```
unsigned int
ADQxxx_HasTriggeredStreamingFunctionality(
void* adq_cu_ptr,
int adqxxx_num)
```

Valid for: ADQ412

Asks the ADQ whether it has the triggered streaming functionality. This function is always called in **TriggeredStreamingSetup**, and will cause an error in that function if the ADQ-firmware is not compatible.

TriggeredStreamingSetup

```
unsigned int
ADQxxx_TriggeredStreamingSetup(
  void* adq_cu_ptr,
  int adqxxx_num,
  unsigned int NofRecords,
  unsigned int NofSamples,
  unsigned int NofPreTrigSamples,
  unsigned int NofHoldOffSamples,
  unsigned char ChannelsMask)
```

Valid for: ADQ412

Sets up Triggered Streaming, a function used to rapidly trigger short data collections.

Triggering may be done either individually for each channel (with level trigger), or for all channels at the same time (other trigger modes). Data is output one channel at a time. Readout is easiest done with the function **GetTriggeredStreamingRecords**.

NofRecords is the number of times to trigger a data collection for each active channel. From 1 to $2^{32}-2$ records. $2^{32}-1$ is a special value, that enables infinite collection.

NofSamples is the number of samples to collect for each record. A number of these samples will be overwritten by the record header. For ADQ412 this number is 8 in non-interleaved mode and 16 in interleaved mode. For ADQ412 **NofSamples** is maximum 65536 in non-interleaved mode and 131072 in interleaved mode. Must be set in multiples of 32.

NofPreTrigSamples is the number of samples to collect from before the trigger arrives. When using pre-trigger, **NofHoldoffSamples** should be set to 0.

The pre-trigger value is internally rounded downwards to a multiple of a constant factor. For ADQ412 this factor is 8 in non-interleaved mode and 16 in interleaved mode.

NofHoldOffSamples is the number of samples to ignore after the trigger arrives. When this is used, NofPreTrigSamples should be 0. Holdoff affects the rearm time in a negative way. For fast triggering, **NofHoldoffSamples** should be set to 0.

The holdoff value is internally rounded downwards to a multiple of a constant factor. For ADQ412 this factor is 8 in non-interleaved mode and 16 in interleaved mode.

ChannelsMask is used to specify from which channels to collect data. Bit 0 enables channel A, bit 1 channel B and so forth. For example on ADQ412, ChannelsMask = 0xF enables all channels while ChannelsMask = 0x3 enables only channel A and B.

Note: To enable streaming of data over the physical interface, **SetStreamStatus(0x7)** must be called after **TriggeredStreamingSetup**.

Note: To enable storage in the on-board DRAM, **SetStreamStatus(0x9)** must be called after this function. This requires the use of **MemoryDump** to read out the data later on and **MemoryShadow** to tell **GetTriggeredStreamingRecords** to look at the dumped data.

SetTriggeredStreamingHeaderRegister

```
unsigned int
ADQxxx_SetTriggeredStreamingHeaderRegister(
  void* adq_cu_ptr,
  int adqxxx_num,
  char Registervalue)
```

Valid for: ADQ412

Puts the user-defined 8-bit value **Registervalue** in the Triggered Streaming record headers. Useful for keeping track of different measurements and debugging purposes.

SetTriggeredStreamingHeaderSerial

```
unsigned int ADQxxx_
SetTriggeredStreamingHeaderSerial(
  void* adq_cu_ptr,
  int adqxxx_num,
  unsigned int SerialNumber)
```

Valid for: ADQ412

Overwrites the **SerialNumber** field in the Triggered Streaming-header with a user-specified value. Must be called after **TriggeredStreamingSetup** to have an affect.

TriggeredStreamingArm

```
unsigned int ADQxxx_TriggeredStreamingArm(
  void* adq_cu_ptr,
  int adqxxx_num)
```

Valid for: ADQ412

Arms triggered streaming. Must be called after **TriggeredStreamingSetup** in order to enable data collection.

TriggeredStreamingDisarm

```
unsigned int
ADQxxx_TriggeredStreamingDisarm(
  void* adq_cu_ptr,
  int adqxxx_num)
```

Valid for: ADQ412

Disarms triggered streaming.

GetTriggeredStreamingRecordSizeBytes

```
unsigned int
ADQxxx_GetTriggeredStreamingRecordSizeBytes
(
  void* adq_cu_ptr,
  int adqxxx_num)
```

Valid for: ADQ412

Returns the number of bytes needed to store the actual samples (without header) from a record.

GetTriggeredStreamingHeaderSizeBytes

```
unsigned int
ADQxxx_GetTriggeredStreamingHeaderSizeBytes
(
  void* adq_cu_ptr,
  int adqxxx_num)
```

Valid for: ADQ412

Returns the size of the header. This parameter is constant at 16 bytes.

TriggeredStreamingGetStatus

```
unsigned int
ADQxxx_TriggeredStreamingGetStatus(
  void* adq_cu_ptr,
  int adqxxx_num,
  unsigned int* InIdle,
  unsigned int* TriggerSkipped,
  unsigned int* Overflow)
```

Valid for: ADQ412

Returns status parameters at the poiters sent to the function:

InIdle: 1 if no collection is currently being made.

TriggerSkipped: A vector of a bit for each channel, each indicating if a trigger was skipped by the particular channel due to the module not being able to buffer an extra record.

Overflow: Indicates that an overflow occurred in the data buffering, may cause data to be lost.

TriggeredStreamingGetNofRecordsCompleted

```
unsigned int
ADQxxx_TriggeredStreamingGetNofRecordsCompleted(
  void* adq_cu_ptr,
  int adqxxx_num,
  unsigned int ChannelsMask,
  unsigned int* NofRecordsCompleted)
```

Valid for: ADQ412

Reads how many records that have been completed for one or more channels

ChannelsMask: Mask to select which channel(s) to read from. Bit 0 selects channel A, bit 1 selects channel B, and so forth.

NofRecordsCompleted: Pointer to where to store the result. All selected channels are added together.

Example: For an ADQ412, channel A has completed 4 records and all other channels have completed 3 records. Using **ChannelsMask** = 0x7 (read from channels C, B & A) will result in **NofRecordsCompleted*** = 10.

GetTriggeredStreamingRecords

```
unsigned int
ADQxxx_GetTriggeredStreamingRecords(
  void* adq_cu_ptr,
  int adqxxx_num,
  unsigned int NofRecordsToRead,
  void** data_buf,
  void* header_buf,
  unsigned int* NofRecordsRead)
```

Valid for: ADQ412

Collects a number of Trigger-Streaming records from the ADQ and stores the result at user-specified memory spaces. The records are fetched one channel at a time, in the order of collection.

NofRecordsToRead: specifies the number of records to read from the ADQ. The records arrive in the order of collection.

data_buf: pointer to different buffers, one for each channel of the device, where the actual data is output (without headers). If multiple records are collected from a channel, these are simply stored after eachother in the buffer.

The user must allocate these buffers.

header_buf: pointer to a buffer where the headers are stored in order. In level trigger mode this information is needed to determine from which buffer in **data_buf** to read the data, as the channels collect data individually. For other modes, the channel order is always A,B,C,D for ADQ412 if all channels are enabled.

The user must allocate this buffer.

NofRecordsRead: pointer to an integer where the function returns the number of records that were collected.

Note: When streaming data to host, **GetTriggeredStreamingRecords** assumes that the buffer size of the transfer buffers have been set to the size of a record during setup. This is done by calling the function **SetTransferBuffers**. If the total amount of data that is to be collected is small enough, the number of buffers should match the the total number of records to collect. This removes the risk of overflow due to full DMA buffers.

ParseTriggeredStreamingHeader

```
unsigned int
ADQxxx_ParseTriggeredStreamingHeader(
  void* adq_cu_ptr,
  int adqxxx_num,
  void* HeaderPtr,
  unsigned long long* Timestamp,
  unsigned int* Channel,
  unsigned int* ExtraAccuracy,
  int* RegisterValue,
  unsigned int* SerialNumber,
  unsigned int* RecordCounter)
```

Valid for: ADQ412

Reads a Triggered Streaming-header and returns the values.

HeaderPtr: Pointer to the first byte of the header to parse.

Timestamp: Pointer to where to return the value of the internal time counter stored in the header. Useful for knowing when a record was triggered.

Channel: Pointer to where to return the channel that was read. Channel 1 = A, 2 = B, 4 = C, and 8 = D.

ExtraAccuracy: Not currently used.

RegisterValue: Pointer to where to return the register value that was stored in the header. The value may be specified using **SetTriggeredStreamingHeaderRegister**.

SerialNumber: Pointer to where to return the serial number of the board. The value may be overridden using **SetTriggeredStreamingHeaderSerial**.

RecordCounter: Pointer to where to return the record number stored in the header. This value starts at 0 and is then incremented for each record. If infinite streaming is used, this value will wrap back to 0 after 131072 records have been collected for the specific channel.

WaveformAveragingParseDataStream

```
unsigned int

ADQxxx_WaveformAveragingParseDataStream(
  unsigned int samples_per_record,
  int* data_stream,
  int** data_target)
```

Valid for: ADQ412, ADQ1600, SDR14, ADQ214, ADQ212, ADQ114, ADQ112

Parses a buffer filled with a single record of streamed WFA data, and stores it into a set of target buffers, one per channel.

samples_per_record = Number of samples per channel in the buffer

data_stream = Pointer to the buffer containing data to be parse

data_target = Pointer to an array of pointers, which in turn point to an allocated buffer for each channel. Example:

data_target[0] = pointer to a buffer which can hold (samples_per_record) samples of data for channel A

data_target[1] = pointer to a buffer which can hold (samples_per_record) samples of data for channel B

…etc

WaveformAveragingSoftwareTrigger

```
unsigned int

ADQxxx_WaveformAveragingSoftwareTrigger()
```

Valid for: ADQ412, ADQ1600, SDR14

Issue a software trigger to the WFA module. Only valid for V6 digitizers. For V5 digitizers, the SWTrig() command should be used instead.

6.4.2 Packet Streaming Functions

ADQ Special Block Function	Description
PacketStreamingSetup **unsigned int ADQxxx_PacketStreamingSetup(void* adq_cu_ptr, int adqxxx_num, unsigned int PacketSizeSamples, unsigned int NofPreTrigSamples, unsigned int NofHoldoffSamples)** Valid for: ADQ214	Sets up the packet streaming block on the ADQ. **PacketSizeSamples** is the number of samples in each package **NofPreTrigSamples** is the number of samples to keep from before the trigger event. **NofHoldOffSamples** is the number of samples to hold off after the trigger event. ***Note***: ADQ214: Packet size, Pretrig and Holdoff are chosen by 2 sample increments. ***Note:*** If streaming over USB is used, one should preferably choose a sample size of the waveform that equals a packet size of 512 bytes. Each averaged sample is 4 bytes, therefore sample sizes should be chosen as 128 sample increments. Also, best practice is to use SetTransferBuffers to complete each packet independently, i.e. set the transfer buffer size to the expected number of bytes of each packet. ***Note:*** The packet streaming block and waveform averaging block cannot be used at the same time. Returns 1 for successful operation and 0 for failure.
PacketStreamingArm **unsigned int ADQxxx_PacketStreamingArm(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ214	Arms the packet streaming. Packets will be pushed on the data interface for each trigger. Returns 1 for successful operation and 0 for failure.
PacketStreamingDisarm **unsigned int ADQxxx_PacketStreamingDisarm(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ214	Disarms the packet streaming block and puts it in bypass mode. Returns 1 for successful operation and 0 for failure.

6.4.3 Interleaving IP Block Functions

ADQ Special Block Function	Description
ResetInterleavingIP **unsigned int ADQxxx_ResetInterleavingIP(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr);** Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14	Resets the interleaving IP block. Note: ADQ112/ADQ114 contains one single instance addressed by 0. ADQ412 contains two instances addressed by 0 and 1.
GetInterleavingIPCalibration **unsigned int ADQxxx_GetInterleavingIPCalibration(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int* calibration);** Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14	Resets the interleaving IP block. **calibration** is an area of memory to place calibration in. Provide at least 8kbyte for this area, i.e. at least 2048 32-bit integers.
SetInterleavingIPCalibration **unsigned int ADQxxx_SetInterleavingIPCalibration(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int* calibration);** Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14	Resets the interleaving IP block. **calibration** is an area of memory where the calibration to set is stored in. The memory contents must be fetched by GetInterleavingIPCalibration. Placing any other content will cause unpredictable results.
GetInterleavingIPBypassMode **unsigned int ADQxxx_GetInterleavingIPBypassMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int* bypassflag);** Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14	Gets the current mode, whether the IP is bypassed or not. Result is returned in **bypassflag**.
SetInterleavingIPBypassMode **unsigned int ADQxxx_SetInterleavingIPBypassMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int bypassflag);** Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14	Sets the current mode, whether the IP is bypassed or not. Set by input argument **bypassflag**.

GetInterleavingIPEstimationMode

```
unsigned int ADQxxx_GetInterleavingIPEstimationMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int* updatetype);
```

Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14

Gets the current mode, whether the IP is allowed to perform parameter updates (background calibration) or not and what parameter update mode to use.

Result is returned in **updateflag**.

SetInterleavingIPEstimationMode

```
unsigned int ADQxxx_SetInterleavingIPEstimationMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int updatetype);
```

Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14

Sets the current mode, whether the IP is allowed to perform parameter updates (background calibration) or not and what parameter update mode to use.

0 = No updates allowed
1 = Normal mode(default)
2 = Time-domain mode

Set by input argument **updatetype**.

Note: For more information on the different modes and when to use them, please contact SP Devices.

SetInterleavingIPFrequencyCalibrationMode

```
unsigned int ADQxxx_SetInterleavingIPFrequencyCalibrationMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int freqcalmode);
```

Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14

Sets the current mode, whether the IP should use the frequency calibration mode or not.

Set by input argument **freqcalmode**

GetInterleavingIPFrequencyCalibrationMode

```
unsigned int ADQxxx_GetInterleavingIPFrequencyCalibrationMode(void* adq_cu_ptr, int adqxxx_num, unsigned char IPInstanceAddr, unsigned int* freqcalmode);
```

Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14

Gets the current mode, whether the IP are usingthe frequency calibration mode or not.

Result is returned in argument **freqcalmode**

SendIPCommand

```
unsigned int ADQxxx_SendIPCommand(unsigned char IPInstanceAddr, unsigned char cmd, unsigned int arg1, unsigned int arg2, unsigned int* answer);
```

Valid for: ADQ112, ADQ114, ADQ412, ADQ1600, SDR14

SendIPCommand gives the user direct access to the ADX command interface.

IPInstanceAddr selects between different IP instances for products with multiple ADX cores (e.g. ADQ412, SDR14), and is zero-indexed. This parameter should be set to 0 for digitizers with a single ADX core.

The command and arguments are passed in the cmd/arg1/arg2 parameters, while the response is returned via the answer pointer.

Further information regarding commands that may be used, is given in the ADX IP user guide.

6.4.4 Precise-Period Trigger

Precise Period Trigger function	Description
SetPPTActive **unsigned int ADQxxx_SetPPTActive(void* adq_cu_ptr, int adqxxx_num, unsigned int active)** Valid for: ADQ108, ADQ208	Activates or deactivates precise period trigger, synchronizing the external trigger to a precise period. Refer to the PPT user guide and example for more detailed information on how to use the precise period trigger. ***active = 1 =>PPT active*** ***active = 0 =>PPT deactivated*** Returns 1 for successful operation and 0 for failure.
InitPPT **unsigned int ADQxxx_InitPPT(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ108, ADQ208	Initializes the precise period trigger. Returns 1 for successful operation and 0 for failure.
SetPPTInitOffset **unsigned int ADQxxx_SetPPTInitOffset(void* adq_cu_ptr, int adqxxx_num, unsigned int init_offset)** Valid for: ADQ108, ADQ208	Sets the precise period trigger init offset. The init offset should be set to a number of samples from 32 to (2^27)-1. The offset is applied to the period on the first trig after initialization of the precise period trigger. Returns 1 for successful operation and 0 for failure.
SetPPTPeriod **unsigned int ADQxxx_SetPPTPeriod(void* adq_cu_ptr, int adqxxx_num, unsigned intperiod)** Valid for: ADQ108, ADQ208	Sets the precise period trigger period. The period should be set to a number of samples from 32 to (2^27)-1. Returns 1 for successful operation and 0 for failure.
SetPPTBurstMode **unsigned int ADQxxx_SetPPTBurstMode(void* adq_cu_ptr, int adqxxx_num, unsigned int active)** Valid for: ADQ108, ADQ208	Activates or deactivates the precise period trigger burst mode. In burst mode, the device will continue to trigger at each PPT period after the first external trigger event without the need for more external trigger events. ***active = 1 =>Burst mode active*** ***active = 0 =>Burst mode deactivated*** Returns 1 for successful operation and 0 for failure.
GetPPTStatus **unsigned int ADQxxx_GetPPTStatus(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQ108, ADQ208	Returns the status register for the PPT function.

6.4.5 ADQ DSP and DSU Specific Functions

ADQ DSP function	Description
InitTransfer **int ADQxxx_InitTransfer(void* adq_cu_ptr, int adqxxx_num,)** Valid for: ADQDSP, DSU	Initiate and flush the data path. Must be issued before any transfer of data to or from ADQDSP.
GetDSPData **int ADQxxx_GetDSPData(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQDSP, DSU	Start transfer of data from the internal memory buffers of the ADQDSP device to the ADQDSP-object in the host computer.
GetDSPDataNowait **int ADQxxx_GetDSPDataNowait(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQDSP, DSU	Start transfer of data from the internal memory buffers of the ADQDSP device to the ADQDSP-object in the host computer. Use WaitForPCIeDMAFinish before reading data to ensure that data transfer is complete.
SetSendLength **int ADQxxx_SetSendLength(void* adq_cu_ptr, int adqxxx_num, unsigned int length)** Valid for: ADQDSP, DSU	Set the size of data vectors that shall be transferred from ADQDSP in 32 bits words. This length is used by GetData and GatDataNowait.
GetSendLength **unsigned int ADQxxx_GetSendLength(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQDSP, DSU	Returns the value set by SetSendLength.
WaitForPCIeDMAFinish **int ADQxxx_WaitForPCIeDMAFinish(void* adq_cu_ptr, int adqxxx_num, unsigned int length)** Valid for: ADQDSP, DSU	Wait for transaction from ADQDSP to complete. See also GetDSPDataNowait.
WriteToDataEP **int ADQxxx_WriteToDataEP(void* adq_cu_ptr, int adqxxx_num, unsigned int *pData, unsigned int length)** Valid for: ADQDSP, SDR14, DSU	Write data to ADQDSP. Length is number of 32 bit words in pData. Note: This length is not affected by SetSendLength.

TrigOutEn
int ADQxxx_TrigOutEn(void* adq_cu_ptr, int adqxxx_num, unsigned int en)
Valid for: ADQDSP, DSU
	Enable or disable TrigIn to TrigOut propagation.
	En = 0: Disabled
	En = 1: Enabled.

GetPhysicalAddress **unsigned long ADQxxx_GetPhysicalAddress(void* adq_cu_ptr, int adqxxx_num)** Valid for: ADQDSP, DSU	Get the physical DMA address of the unit.

6.4.6 Arbitrary Waveform Generator (AWG)

AWG function	Description
AWGSegmentMalloc **unsigned int ADQxxx_AWGSegmentMalloc(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int dacId,** **unsigned int segId,** **unsigned int length,** **unsigned charreallocate)** Valid for: SDR14	Allocate memory space for an AWG segment. segId selects the segment to allocate for, and length determines the number of samples to be allocated to the segment in memory. The length parameter must be a multiple of 16. The function uses the end address of the preceeding segment internally during allocation, so an allocation loop using this function should always go from segment 1 and upwards sequentially, never the other way around. The reallocate parameter can be used to reallocate the memory mapping of all the segments following the one that is being modified. This is useful if only a few segments are to be reallocated and the user desires the update of the remaining segments to be done automatically. If, however, every segment in the entire AWG is to be reallocated within a loop by the user, the reallocate parameter should be set to 0 in order to avoid wasting computations. dacId: 1 or 2 (select AWG/DAC)
AWGWriteSegment **unsigned int ADQxxx_AWGWriteSegment(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int dacId,** **unsigned int segId,** **unsigned int enable,** **unsigned int NofLaps,** **unsigned int length,** **int *data)** Valid for: SDR14	Writes a segment to the AWG memory allocated by first using AWGSegmentMalloc. The data length must be a multiple of 16 samples. If *data is a null pointer, all the other settings will be set without writing any new data to the DRAM. The input data should be two's complement integers. The *enable* parameter is deprecated and will have no effect on the AWG. Use AWGEnableSegments for setting the number of enabled segments. NofLaps sets the number of laps which the segment should be looped before the AWG continues to the next segment. Bit 31 in the NofLaps integer is used to enable *infinite-laps mode* where the segment loops infinitely (until the AWG is disarmed, or a special trigger mode forces a segment switch, see AWGTrigMode). This means that the maximum number of laps that may be used without infinite looping is 2^31-1. dacId: 1 or 2 (select AWG/DAC)

AWGArm

```
unsigned int ADQxxx_AWGArm(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId)
```

Valid for: SDR14

Arms the AWG. This preloads the first set of data from the DRAM so that the AWG is ready to output data as soon as it is triggered.

dacId: 1 or 2 (select AWG/DAC)

AWGDisarm

```
unsigned int ADQxxx_AWGDisarm(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId)
```

Valid for: SDR14

Disarms the AWG, i.e turns it off. A trigger event will not cause the AWG to output data once it is disarmed.

dacId: 1 or 2 (select AWG/DAC)

AWGEnableSegments

```
unsigned int ADQxxx_AWGEnableSegments(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId,
             unsigned int enableSeg)
```

Valid for: SDR14

Enables the specified amount of segments.

During readout, the AWG will output all segments up to and including this number, before restarting.

dacId: 1 or 2 (select AWG/DAC)

AWGAutoRearm

```
unsigned int ADQxxx_AWGAutoRearm(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId,
             unsigned int enable)
```

Valid for: SDR14

Turns auto-rearm mode on (enable = 1) and off (enable = 0) for the AWG. This mode will rearm the AWG immediately upon a finished readout cycle, to make it ready for a new trigger event.

dacId: 1 or 2 (select AWG/DAC)

AWGContinuous

```
unsigned int ADQxxx_AWGContinuous(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId,
             unsigned int enable)
```

Valid for: SDR14

Turns continuous mode on and off for the AWG. If this mode is turned on (enable = 1), the AWG will start outputting data as soon as it is armed.

dacId: 1 or 2 (select AWG/DAC)

AWGTrigMode

```
unsigned int ADQxxx_AWGTrigMode(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId,
             unsigned int trigmode)
```

Valid for: SDR14

This function allows special triggering modes of the AWG to be enabled.

dacId: 1 or 2 (select AWG/DAC)

The trigmode variable selects the mode to be used, according to the following list:

trigmode = 0
Normal single-shot triggering

trigmode = 1
Requires trigger event before starting each segment lap.

trigmode = 2
Seamless segment switching mode. This mode should be used in conjunction with infinite-laps programmed segments (see AWGWriteSegment description).

Upon being triggered, the AWG will wait until the end of the current segment lap before seamlessly switching to the next segment. This allows the user to loop segments indefinitely, with the trigger acting as break for the loop, and without any junk data being output when the segment switch occurs.

NOTE: If seamless mode is enabled during the very first trigger that starts the AWG, the AWG will immediately seamlessly skip to segment 2. For this reason, always trigger the AWG without seamless mode initially, and then enable it for subsequent triggers.

AWGSetTriggerEnable

```
unsigned int ADQxxx_AWGSetTriggerEnable(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned int dacId,
             unsigned int bitflags)
```

Valid for: SDR14

This function allows selection of which trigger signals may be used to trigger the AWG.

dacId: 1 or 2 (select AWG/DAC)

The bitflags variable should be considered a bit field where each bit enables a trigger if asserted, according to:

bit 0:	Host trigger/software trigger from the data acquisition logic.
bit 1:	External trigger
bit 2:	PXIe port1 trigger
bit 3:	Internal trigger

AWGSetupTrigout

```
unsigned int ADQxxx_AWGSetupTrigout(
              void* adq_cu_ptr,
              int adqxxx_num,
              unsigned int dacId,
              unsigned int trigoutmode,
              unsigned int pulselength,
              unsigned int enableflags,
              unsigned int autorearm)
```

Valid for: SDR14

The AWG may be used to output a trigger signal (to the PXIe backplane, trigger output connector or similar.

dacId: 1 or 2 (select AWG/DAC)

trigoutmode: 0 – off

1 – pulse at the start of each segment

2 – pulse at the end of each segment

pulselength: Sets the trigout pulse length, in number of 200MHz clock cycle periods.

The enableflags variable is a bitfield, where each bit enables output to the following:

bit 0: Trigout connector (not implemented yet)

bit 1: PXIe port1 trigger output

autorearm: 0 - autorearm off (requires manual rearm after every triggered trigout pulse)

1 – autorearm on

AWGTrigoutArm

```
unsigned int ADQxxx_AWGTrigoutArm(
              void* adq_cu_ptr,
              int adqxxx_num,
              unsigned int dacId)
```

Valid for: SDR14

Arms the trigger output of the specified AWG. If the AWG is to be rearmed after having triggered, an AWGTrigoutDisarm command must first be issued.

dacId: 1 or 2 (select AWG/DAC)

AWGTrigoutDisarm

```
unsigned int ADQxxx_AWGTrigoutDisarm(
              void* adq_cu_ptr,
              int adqxxx_num,
              unsigned int dacId)
```

Valid for: SDR14

Disarms the trigger output of the specified AWG.

dacId: 1 or 2 (select AWG/DAC)

6.4.7 MicroTCA-specific functions

MicroTCA function	Description
SetEthernetPllFreq **unsigned int ADQxxx_SetEthernetPllFreq(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned chareth10freq,** **unsigned chareth1freq)** Valid for:ADQ108, ADQ208, ADQ412, ADQ1600 (Only -MTCA products)	Provides a simple way of setting the 10G and 1G Ethernet GTX clocks to predefined values. Currently allowed presets are: ETH10_FREQ_156_25MHZ (156.25 MHz) ETH10_FREQ_125MHZ (125 MHz) ETH1_FREQ_156_25_MHZ (156.25 MHz) ETH1_FREQ_125_MHZ (125 MHz)
SetPointToPointPllFreq **unsigned int ADQxxx_PointToPointPllFreq(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned chareth10freq,** **unsigned chareth1freq)** Valid for:ADQ108, ADQ208, ADQ412, ADQ1600 (Only -MTCA products)	Provides a simple way of setting the point-to-point interface GTX clock to predefined values. Currently allowed presets are: PP_FREQ_330MHZ (330 MHz) PP_FREQ_250MHZ (250 MHz) PP_FREQ_156_25MHZ (156.25 MHz) PP_FREQ_125MHZ (125 MHz)
SetEthernetPll **unsigned int ADQxxx_SetEthernetPll(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned short refdiv,** **unsigned char useref2,** **unsigned char a,** **unsigned short b,** **unsigned char p,** **unsigned char vcooutdiv,** **unsigned char eth10_outdiv,** **unsigned char eth1_outdiv)** Valid for:ADQ108, ADQ208, ADQ412, ADQ1600 (Only -MTCA products)	Provides an advanced way of setting the 10G and 1G Ethernet GTX clocks. See AD9517-1 PLL datasheet for more info on parameters and allowed values. refdiv: Reference divider, 0 – 16383 useref2: Reference selector, 0 = 10MHz TCXO, 1 = output from clockref mux a: VCO feedback parameter A, 0 - 31 b: VCO feedback parameter B, 0 - 4095 p: VCO feedback parameter P, 2,4,8,16 or 32 vcooutdiv: VCO divider: 1-6 eth10_outdiv: 10G clock output divider 0-32 eth1_outdiv: 1G clock output divider 0-32
SetPointToPointPll **unsigned int ADQxxx_SetPointToPointPll(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned short refdiv,** **unsigned char useref2,** **unsigned char a,** **unsigned short b,** **unsigned char p,** **unsigned char vcooutdiv,** **unsigned char pp_outdiv,** **unsigned char ppsync_outdiv)** Valid for:ADQ108, ADQ208, ADQ412, ADQ1600 (Only -MTCA products)	Provides an advanced way of setting the point-to-point interface clock. See AD9517-1 PLL datasheet for more info on parameters and allowed values. refdiv: Reference divider, 0 – 16383 useref2: Reference selector, 0 = 10MHz TCXO, 1 = output from clockref mux a: VCO feedback parameter A, 0 - 31 b: VCO feedback parameter B, 0 - 4095 p: VCO feedback parameter P, 2,4,8,16 or 32 vcooutdiv: VCO divider: 1-6 pp_outdiv: Point-to-point output divider 0-32 ppsync_outdiv: Point-to-point synched clock for 1G Ethernet output divider 0-32

SetDirectionMLVDS

```
unsigned int ADQxxx_SetDirectionMLVDS(
             void* adq_cu_ptr,
             int adqxxx_num,
             unsigned char direction)
```

Valid for: ADQ108, ADQ208, ADQ412, ADQ1600 (Only -MTCA products)

Sets the direction of the eight LVDS pairs connected to the backplane.

The direction parameter is an 8-bit pattern:

{7,…,0} = {T20,R20,…,T17,R17}

where 0 = input, 1 = output. The setting defaults to all inputs.

6.4.9 Peer-to-Peer function

Peer-to-Per function	Description
SetP2pSize **unsigned int ADQxxx_SetP2pSize(void* adq_cu_ptr, int ADQxxx_num, unsigned int bytes)** Valid for: Device with P2P support	This function sets the size of the package in bytes to be sent for each p2p transaction.
GetP2pSize **unsigned int ADQxxx_GetP2pSize(void* adq_cu_ptr, int ADQxxx_num)** Valid for: Device with P2P support	Returns the value set by SetP2pSize.
SendDataDev2Dev **unsigned int ADQxxx_SendDataDev2Dev(void* adq_cu_ptr, int ADQxxx_num, unsigned long PhysicalAddress)** Valid for: Device with P2P support	Configure device for transaction of one package. It can be called before data is available to prepare device.
GetP2PStatus **unsigned int ADQxxx_GetP2PStatus(void* adq_cu_ptr, int ADQxxx_num, unsigned int* pending, unsigned int channel)** Valid for: Device with P2P support	Gets the number of pending DMA transfer for the DMA channel specified by **channel**. The number of pending transfers is returned at the pointer **pending**.

6.4.10 PXIe backplane trigger block

PXIe backplane trigger function	Description
EnablePXIeTriggers **unsigned int ADQxxx_EnablePXIeTriggers(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int port,** **unsigned int bitflags)** Valid for:SDR14	All the various PXIe trigger inputs are summed together into a single PXIe trigger signal. This function allows specific inclusion/exclusion of these inputs. The PXIe trigger block has two ports which can be configured independently, and which are connected to separate parts of the digitizer logic, according to: port 0: Data acquisition logic port 1: AWG (on SDR14) The bitflags variable should be considered a bit field where each bit enables a specific trigger input for the selected port. bit 0: DSTARA bit 1: DSTARB bit 2: PXI_TRIG[0] bit 3: PXI_TRIG[1] (note: PXI_TRIG[2 to 7] are not routed on the digitizer PCB and cannot be used.)

EnablePXIeTrigout **unsigned int ADQxxx_EnablePXIeTrigout(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int port,** **unsigned int bitflags)** Valid for: SDR14	The trigger output of each port may be connected to any/all the available PXIe trigger outputs via this function. port 0: Data acquisition logic port 1: AWG (on SDR14) The bitflags variable should be considered a bit field where each bit enables output of the port trigout signal to a specific PXIe trigger output. bit 0: DSTARC bit 1: PXI_TRIG[0] bit 2: PXI_TRIG[1]
PXIeSoftwareTrigger **unsigned int ADQxxx_PXIeSoftwareTrigger(** **void* adq_cu_ptr,** **int adqxxx_num)** Valid for: SDR14	This function will send out a trigger signal on all of the enabled trigger outputs.
SetPXIeTrigDirection **unsigned int ADQxxx_SetPXIeTrigDirection(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int trig0output,** **unsigned int trig1output)** Valid for: SDR14	This function sets the direction of the two PXI_TRIG I/O pins. 0: Input (default) 1: Output Make sure that no other drivers are connected to the PXI_TRIG bus before setting the trigger pins to outputs, or you may risk damaging the digitizer.
WriteSTARBDelay **unsigned int ADQxxx_WriteSTARBDelay(** **void* adq_cu_ptr,** **int adqxxx_num,** **unsigned int starbdelay)** Valid for: SDR14	This function writes a delay value to be used on the DSTARB trigger input, which is stored in the onboard EEPROM and loaded upon each restart of the digitizer. The allowed range of values are 0 to 31, where each unit corresponds to a 78 ps delay.

6.5 Deprecated functions

Functions documented here are left for backwards compatibility with older applications. Not recommended to use in new or updated applications.

Deprecated function	Description
SetBufferSize **int ADQxxx_SetBufferSize(** **void* adq_cu_ptr, int adqxxx_num,** **unsigned int samples)** Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, SDR14 **Deprecated**	Setups the memory buffers for single record acquisition (single trigger) in the ADQ device. ***Note:*** *per channel if applicable (ADQ214).* Returns 1 for successful operation and 0 for failure. Failures include trying to allocate more memory than is available. (Recommended: Use MultiRecordSetup instead)

SetBufferSizePages **Deprecated**	Do not use
SetBufferSizeWords **Deprecated**	Do not use
GetBufferSizePages **Deprecated**	Do not use
GetBufferSize **Deprecated**	Do not use
SetLvlTrigResetLevel **Deprecated**	(Recommended: Use SetTrigLevelResetValue instead)
USBTrig **Deprecated**	(Recommended: Use SWTrig instead)
SetLvlTrigFlank **Deprecated**	(Recommended: Use SetLvlTrigEdge)
MultiRecordGetRecord **Deprecated**	(Recommended: Use CollectRecord instead)
SetSampleWidth **int ADQxxx_SetSampleWidth(void* adq_cu_ptr, int adqxxx_num, unsigned int NofBits)** Valid for: ADQ212, ADQ112, ADQ114, ADQ214 **Deprecated**	Sets the sample size of inter-FPGA communication in number of bits. ***NofBits = 8, 12, 14, 16** or 32*****(8, 12 or 14 depending of which ADQ-device you are interfacing)* This value must match sample width of inter-FPGA sample data and should normally not be changed. Returns 1 for successful operation and 0 for failure. **Sample width must be set to 16 bits when streaming is active or 32 bits when decimation is active. (Recommended: Use SetDataFormat instead)
SetNofBits **int ADQxxx_SetNofBits(void* adq_cu_ptr, int adqxxx_num, unsigned int NofBits)** Valid for: ADQ212, ADQ112, ADQ114, ADQ214 **Deprecated**	Sets the word size of inter-FPGA communication in number of bits. ***NofBits = 24, 28 or 32*****(24 or 28 depending of which ADQ-device you are interfacing)* This value must match word width of inter-FPGA sample data and should normally not be changed. Returns 1 for successful operation and 0 for failure. **Sample width must be set to 32 bits when decimation or when streaming is active. (Recommended: Use SetDataFormat instead)

SetAlgoStatus

```
unsigned int   ADQxxx_SetAlgoStatus(void*
adq_cu_ptr, int adqxxx_num, int status);
```

Valid for: ADQ112, ADQ114

Deprecated

Set interleaving algorithm in status**status**.

0 => By-pass interleaving algorithm.

1 => (Default) use interleaving algorithm.

(Recommended: Use SetInterleavingIPBypassMode instead)

SetAlgoNyquistBand

```
unsigned
intADQxxx_SetAlgoNyquistBand(void*
adq_cu_ptr, int adqxxx_num, unsigned int
band);
```

Valid for: ADQ112, ADQ114

Deprecated

Set Nyquist band for interleaving algorithm to **band**.

0 => (Default) From 0 Hz to sampling_frequency/2.

1 => From sampling_frequency/2 to sampling_frequency.

(Recommended: No usage necessary)

GetLvlTrigFlank

```
int ADQxxx_GetLvlTrigFlank(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ412, ADQ212, ADQ108, ADQ112, ADQ114, ADQ214

Deprecated

Returns the edge for which the level trigger shall trig.

Return value = 1 => Rising edge
Return value = 0 => Falling edge

(Recommended: Use GetLvlTrigEdge instead)

GetMaxBufferSize

```
unsigned int ADQxxx_GetMaxBufferSize(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, SDR14

Deprecated

Returns the maximum number of samples in the total acquisition buffer in the ADQ device.

Note: *per channel if applicable (ADQ214).This figure may change when altering the acquisition settings.*

(Recommended: Do not use)

GetMaxBufferSizePages

```
unsigned int
ADQxxx_GetMaxBufferSizePages(
void* adq_cu_ptr, int adqxxx_num)
```

Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, SDR14

Deprecated

Returns the maximum number of internal acquisition pages of the device.

Note: *per channel if applicable (ADQ214).This figure may change when altering the acquisition settings.*

(Recommended: Do not use)

SendLongProcessorCommand

```
unsigned
intADQxxx_SendLongProcessorCommand(
void* adq_cu_ptr, int adqxxx_num,
int command, int addr, int mask, int
data)
```

Valid for: ADQ212, ADQ112, ADQ114, ADQ214

Deprecated

Sends commands to the processor in the comm. FPGA.

The available commands are defined in the processor code and are listed in ADQAPI_definitions.h.

Returns the answer from the processor.

GetTrigged

```
int ADQxxx_GetTrigged(
void* adq_cu_ptr, int adqxxx_num)
```

C++ name: GetAcquired()

Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, SDR14

Deprecated

Returns 1 if the ADQ device has been trigged and data has been acquiredfor one or more its records. 0 else.

(Recommended: Use GetAcquired instead)

GetTriggedAll

```
int ADQxxx_GetTriggedAll(
void* adq_cu_ptr, int adqxxx_num)
```

C++ name: GetAcquiredAll()

Valid for: ADQ108, ADQ112, ADQ114, ADQ1600, ADQ208, ADQ212, ADQ214, ADQ412, SDR14

Deprecated

Returns 1 if the ADQ device has been trigged and data has been acquired for all its records. 0 else.

(Recommended: Use GetAcquiredAll instead)

AWGmalloc

```
unsigned int ADQxxx_AWGmalloc(
              unsigned int dacId,
              unsigned int LengthSeg1,
              unsigned int LengthSeg2,
              unsigned int LengthSeg3,
              unsigned int LengthSeg4)
```

Deprecated

Allocate memory space for AWG vectors. LengthSegN defines how many samples to allocate memory for in that specific segment.

(Recommended: Use AWGSegmentMalloc instead)

GetDataMultiRecordSetup

Deprecated

Do not use

6.6 Intentionally undocumented functions

Functions documented here are included in some of the APIs, but are only intended for internal API and debug purposes, internal to SP Devices. Do not use these in applications, as no documentation will be made available and functions may change behavior at any time.

Undocumented function	Description
ParseEEPROMBlock **Internal only**	External documentation not available
SetDelayLineValues **Internal only**	External documentation not available
SetDelayLineValuesDirect **Internal only**	External documentation not available
SetWordsPerPage **Internal only**	External documentation not available
SetPreTrigWords **Internal only**	External documentation not available
SetWordsAfterTrig **Internal only**	External documentation not available
SetTrigMask1 **Internal only**	External documentation not available
SetTrigLevel1 **Internal only**	External documentation not available
SetTrigPreLevel1 **Internal only**	External documentation not available
SetTrigCompareMask1 **Internal only**	External documentation not available
SetTrigMask2 **Internal only**	External documentation not available
SetTrigLevel2 **Internal only**	External documentation not available
SetTrigPreLevel2 **Internal only**	External documentation not available

SetTrigCompareMask2 **Internal only**	External documentation not available
GetPageCount **Internal only**	External documentation not available
ParseSampleData **Internal only**	External documentation not available
RegisterNameLookup **Internal only**	External documentation not available
SPISend **Internal only**	External documentation not available
GetComFlashEnableBit **Internal only**	External documentation not available
FlashUpdate **Internal only**	External documentation not available
CollectDataNextPageWithPrefetch **Internal only**	External documentation not available
SendProcessorCommand **Internal only**	External documentation not available
WriteI2C **Internal only**	External documentation not available
ReadI2C **Internal only**	External documentation not available
WriteReadI2C **Internal only**	External documentation not available
ADQControlUnit_ReadPCIConfigurationSpaceHeader **Internal only**	External documentation not available
ADQControlUnit_WritePCIConfigurationSpaceHeader **Internal only**	External documentation not available

WaveformAveragingStartReadout Internal only	External documentation not available
PllReg **Internal only**	External documentation not available
OffsetDACSpiWrite **Internal only**	External documentation not available
DACSpiWrite **Internal only**	External documentation not available
DACSpiRead **Internal only**	External documentation not available
GetFPGAPart **Internal only**	External documentation not available
GetFPGATempGrade **Internal only**	External documentation not available
GetFPGASpeedGrade **Internal only**	External documentation not available
IsBootLoader **Internal only**	External documentation not available
BootADQFromFlash **Internal only**	External documentation not available
FX2ReadRequest **Internal only**	External documentation not available
FX2WriteRequest **Internal only**	External documentation not available
ProcessorFlashControlData **Internal only**	External documentation not available
ProcessorFlashControl **Internal only**	External documentation not available
GetNofFPGAs **Internal only**	External documentation not available

GetTrigType **Internal only**	External documentation not available
StorePCIeConfig **Internal only**	External documentation not available
ReloadPCIeConfig **Internal only**	External documentation not available

7 MATLAB INTERFACE

Most of the functions of the API will function in the same way via interface_ADQ.m as they do in the C/C++ style API. However, there are some exceptions:

- Functions that return pointers cannot be called using interface_ADQ. An example of this is `ADQxxx_GetPtrStream`.
- Functions that take one or several pointer as input and store data on those addresses instead return the data directly when used via interface_ADQ. An example is `ADQxxx_GetData`
- ADQControlUnit-functions are not supported.

7.1 Using interface_ADQ

To get a list of all ADQs connected to the system, run the mex-file called mex_ADQ. All detected ADQs are then listed together with a board number that is unique for each device.

Functions of the API are then called using this structure:

```
[data_A, data_B, status] = interface_ADQ(functionname, [arg1, … , argN], boardid)
```

Where the input arguments are:

- `functionname`: a string containing the name of the function (in lower case only).
- `[arg1, … , argN]`: the input arguments given in the same order as in the C/C++ style API. Any pointers given in standard API functions are simply skipped.
- `boardid`: either the number of the ADQ received by mex_ADQ, or a string containing the serial number of the ADQ (e.g. `'SPD-01829'`) can be used to specify which device to use.

The data is returned a bit differently compared to the C/C++ style API:

- C/C++ style API functions that return only a success-flag return this flag in both `data_A` and`status` when used via interface_ADQ.
- C/C++ style API functions that return a data value, return that value in `data_A`, while `status` is left empty.
- C/C++ style API functions that fill an address specified by an input pointer with data instead return that data directly in `data_A` when using interface_ADQ. The value returned by the original function (typically a success flag), is returned in `status`. As an example, a call to GetData via interface_ADQ will store all samples from all channels in `data_A`, and return the 'real' return value in `status`.
- `data_B` is used for a few data collection functions for ADQ214 and ADQ212. For these, data from channel A is returned in `data_A` and data from channel B is returned in `data_B`.

As an example, the following command in C:

```
success = ADQ214_WriteRegister(cu_ptr, adq214_num, addr, mask, data)
```

Becomes:

```
success = interface_ADQ('writeregister', [addr, mask, data], boardid)
```

If `boardid` isn't specified, it will be assumed to be '1'. The vector with input arguments may also be omitted for functions that doesn't use input values, but if a `boardid` is specified it must be an empty vector. For example:

```
success = interface_ADQ('isalive', [], 1)
```

isequivalent to:

```
success = interface_ADQ('isalive')
```

7.2 Functions Differing from C/C++ style API

Interface_ADQ call	Description
collectrecord interface_ADQ('collectrecord', record_num, boardid)	Multirecord mode only. Returns the data in the record specified by record_num. For ADQ214 and ADQ212 the data for channel A and B is returned in data_A and data_B, respectively. For all other products the data from all channels is returned as a struct in data_A.
gettemperature interface_ADQ('gettemperature', addr, boardid)	Returns the temperature in Celsius of the sensor specified by addr. The return value of the original API function is scaled by a factor of 256, but interface_ADQ removes this scaling.

7.3 Functions Specific for interface_ADQ.m

Interface_ADQ call	Description
getdatastream interface_ADQ('getdatastream', NofBytesToCopy, boardid)	Calls GetPtrStream() of the API, and copies **NofBytesToCopy** bytes from this address. Returns the raw data bytes as a vector in data_A

8 ERROR CODES

These are the available error codes as reported by GetLastError function:

```
#define ERROR_CODE_NO_ERROR_OCCURRED                                0x00000000
#define ERROR_CODE_ADQAPI_NOT_BUILT_FOR_CORRECT_OS                  0x00000001
#define ERROR_CODE_FUNCTION_NOT_SUPPORTED_BY_DEVICE                 0x00001000
#define ERROR_CODE_CHANNEL_NOT_AVAILABLE_ON_DEVICE                  0x00001001
#define ERROR_CODE_FUNCTION_NOT_SUPPORTED_BY_DEVICE_REVISION        0x00001002
#define ERROR_CODE_GETDATA_BUFFERPOINTERS_NULL                      0x00002001
#define ERROR_CODE_GETDATA_ENDRECORDNUMBER_TOO_HIGH                 0x00002002
#define ERROR_CODE_GETDATA_TARGET_BUFFER_SIZE_SPEC_TOO_SMALL        0x00002003
#define ERROR_CODE_GETDATA_SAMPLE_SETTING_GT_RECORDSIZE             0x00002004
#define ERROR_CODE_GETDATA_TRANSFER_SETTINGS_BAD                    0x00002005
#define ERROR_CODE_DELAY_COMPENSATION_NOT_IN_RANGE                  0x00002100
#define ERROR_CODE_OPEN_READ_VDD_EEPROM_FAILD                       0x00003001
#define ERROR_CODE_OPEN_SET_VDD_FAILD                               0x00003002
#define ERROR_CODE_OPEN_SET_PARAM_1_FAILD                           0x00003003
#define ERROR_CODE_OPEN_SET_PARAM_2_FAILD                           0x00003004
#define ERROR_CODE_OPEN_SET_PARAM_3_FAILD                           0x00003005
#define ERROR_CODE_OPEN_SET_DEF_CLOCK_SOURCE                        0x00003006
#define ERROR_CODE_OPEN_SET_PLL_DEF                                 0x00003007
#define ERROR_CODE_OPEN_RESET_ADC                                   0x00003008
#define ERROR_CODE_OPEN_CALIBRATE_ADC                               0x00003009
#define ERROR_CODE_OPEN_INIT_ADC                                    0x0000300A
#define ERROR_CODE_OPEN_SETPLL_1                                    0x0000300B
#define ERROR_CODE_OPEN_SETPLL_2                                    0x0000300C
#define ERROR_CODE_OPEN_SET_DATA_FORMAT                             0x0000300D
#define ERROR_CODE_OPEN_DRAM_INIT_FAILED                            0x0000300E
#define ERROR_CODE_OPEN_CALIBRATE_PLL                               0x0000300F
#define ERROR_CODE_OPEN_DESKEW_TRIGGERING_FAILED                    0x00003010
#define ERROR_CODE_OPEN_WRONG_DAUGHTERBOARD                         0x00003011
#define ERROR_CODE_SET_PLL_FAILED_BAD_SETTINGS                      0x00003100
#define ERROR_CODE_SET_PLL_FAILED_STAGE_1                           0x00003101
#define ERROR_CODE_REGISTER_NOT_AVAILABLE                           0x00003400
#define ERROR_CODE_IP_NOT_IN_CORRECT_MODE                           0x00004001
#define ERROR_CODE_IP_REPORTS_BAD_SIZE                              0x00004002
#define ERROR_CODE_IP_GENERAL_ERROR                                 0x00004003
#define ERROR_CODE_IP_NO_ANSWER                                     0x00004004
#define ERROR_CODE_WAVEFORMAVERAGING_SETUP_BAD                      0x00005001
#define ERROR_CODE_PACKETSTREAMING_SETUP_BAD                        0x00006001
#define ERROR_CODE_STREAMING_OUT_OF_SYNC_WARNING                    0x00006100
```