

PROGRAMMER'S GUIDE AGILENT ACQIRIS INSTRUMENTS

Manual Part Number

U1092-90003

Edition

E-Rev: 0, April 2012

The information in this document is subject to change without notice and may not be construed in any way as a commitment by Agilent Technologies, Inc. While Agilent makes every effort to ensure the accuracy and contents of the document it assumes no responsibility for any errors that may appear.

All software described in the document is furnished under license. The software may only be used and copied in accordance with the terms of license. Instrumentation firmware is thoroughly tested and thought to be functional but it is supplied "as is" with no warranty for specified performance. No responsibility is assumed for the use or the reliability of software, firmware or any equipment that is not supplied by Agilent or its affiliated companies.

You can download the latest version of this manual from http://www.agilent.com/ by clicking on Manuals in the Technical Support section and then entering a model number. You can also visit our web site at http://www.agilent.com/find/acqiris. At Agilent we appreciate and encourage customer input. If you have a suggestion related to the content of this manual or the presentation of information, please contact your local Agilent Acqiris product line representative or the dedicated Agilent Acqiris Technical Support (ACQIRIS_SUPPORT@agilent.com).

Acqiris Product Line Information

USA (800) 829-4444

Asia - Pacific 61 3 9210 2890

Europe 41 (22) 884 32 90

© Copyright Agilent 2012

Programmer's Guide Page 2 of 64

CONTENTS

| 1. IN | TRO | DUCTION | 5 | | | | | | |
|-------|---------------------|---|----------|--|--|--|--|--|--|
| 1.1. | Message to the User | | | | | | | | |
| 1.2. | ϵ | | | | | | | | |
| 1.3. | Co | nventions Used in This Manual | 6 | | | | | | |
| 1.4. | Wa | 6 | | | | | | | |
| 1.5. | Warranty | | | | | | | | |
| 1.6. | Wa | arranty and Repair Return Procedure, Assistance and Support | <i>6</i> | | | | | | |
| 1.7. | Sys | stem Requirements | 6 | | | | | | |
| 2. PF | ROGI | RAMMING ENVIRONMENTS & GETTING STARTED | 7 | | | | | | |
| 2.1. | | sual C++ | | | | | | | |
| 2.2. | Lal | oVIEW | | | | | | | |
| 2.2 | 2.1. | LabView 7.x/8 with old LabView programs | 8 | | | | | | |
| 2.2 | 2.2. | AqDx Getting Started VI | 9 | | | | | | |
| 2.2 | 2.3. | AqDx Example Scope VI | 9 | | | | | | |
| 2.2 | 2.4. | AqDx Accumulated Waveform Example VI | 10 | | | | | | |
| 2.3. | MA | ATLAB | 10 | | | | | | |
| 2.4. | Lin | ux | 10 | | | | | | |
| 3. PI | ROGI | RAMMING AN ACQIRIS INSTRUMENT | 11 | | | | | | |
| 3.1. | Pro | ogramming Hints | 11 | | | | | | |
| 3.2. | | vice Initialization | | | | | | | |
| 3.2 | 2.1. | PCI & VXI Identification by Order Found | | | | | | | |
| 3.2 | 2.2. | PCI Identification by Serial Number | | | | | | | |
| 3.2 | 2.3. | PCI Identification by Bus/Slot Number | | | | | | | |
| 3.2 | 2.4. | VXI Identification | 13 | | | | | | |
| 3.2 | 2.5. | PXI VISA & LabViewRT Identification | | | | | | | |
| 3.2 | 2.6. | Firmware initialization (AP-FAMILY/12-bit-FAMILY/AC/SC/TC) | | | | | | | |
| | 2.7. | Automatic Definition of MultiInstruments | | | | | | | |
| | 2.8. | Manual Definition of MultiInstruments | | | | | | | |
| | 2.9. | AqGeo.map file positioning | | | | | | | |
| | 2.10. | Simulated Devices | | | | | | | |
| | 2.11. | Terminating an Application | | | | | | | |
| 3.3. | 2.12. | Reinitializationvice Configuration | | | | | | | |
| | | nfiguring Averagers | 18 | | | | | | |
| | | Basic configuration | | | | | | | |
| | 4.1. 4.2. | Dithering | | | | | | | |
| | 4.2. 4.3. | 'Fixed Pattern' Background Subtraction | | | | | | | |
| | 4.4. | Configuring Noise Suppressed Accumulation (NSA) | | | | | | | |
| 3.5. | | nfiguring SSR Analyzers | | | | | | | |
| 3.5 | 5.1. | Acquisition Parameters | | | | | | | |
| 3.5 | 5.2. | Readout configuration | | | | | | | |
| 3.5 | 5.3. | SSR Time stamps | 23 | | | | | | |
| 3.6. | U1 | 084 Zero-Suppression mode | | | | | | | |
| 3.6 | 6.1. | U1084A Zero-Suppression mode – Acquisition Parameters | 23 | | | | | | |
| 3.6 | 6.2. | U1084 Zero-Supression mode – Threshold configuration | 24 | | | | | | |
| 3.6 | 6.3. | U1084A Zero-Suppression mode – Readout configuration | 25 | | | | | | |
| 3.6 | 6.4. | U1084A Zero-Suppression mode - Data Format | | | | | | | |
| 3.7. | | nfiguring AP family Peak ^{TDC} Analyzers | | | | | | | |
| 3.8. | | nfiguring U1084A Peak ^{TDC} Analyzers | | | | | | | |
| 3.9. | | nfiguring AP101/AP201 Analyzers | | | | | | | |
| 3.10. | . 1 | Data Acquisition | | | | | | | |
| | 10.1. | Starting an Acquisition | | | | | | | |
| | 10.2. | Checking if Ready for Trigger | | | | | | | |
| 2 1 | 10.3 | Waiting for End of Acquisition | 20 | | | | | | |

| | 3.10.4. | Stopping/Forcing a D1-style Acquisition | 30 |
|----|---------|--|----------------|
| | 3.10.5. | Simultaneous multibuffer Acquisition and Readout (SAR) | |
| | 3.10.6. | Analyzer and Peak ^{TDC} Autoswitch mode | 32 |
| | 3.11. | D1-style Data Readout | 34 |
| | 3.11.1. | Reading Digitizer Waveforms with the Universal Read Function | 34 |
| | 3.11.2. | Reading Sequences of Waveforms | 35 |
| | 3.11.3. | Reading Raw Sequences of Waveforms | 36 |
| | 3.11.4. | Averaging Waveforms in a Digitizer | 37 |
| | 3.11.5. | Reading an Averaged Waveform from an Averager | |
| | 3.11.6. | Reading a RT Add/Subtract Averaged Waveform from an Averager | |
| | 3.11.7. | Reading SSR Analyzer Waveforms | |
| | 3.11.8. | Reading AP family Peak TDC Analyzer Data and Histograms | 40 |
| | 3.11.9. | E | |
| | 3.11.10 | e · | |
| | | Trigger Delay and Horizontal Waveform Position | |
| | | Horizontal Parameters in Acquired Waveforms | |
| | 3.14. | Sequence Acquisitions | 48 |
| | 3.15. | Time stamps | 48 |
| | 3.16. | External Clock and Reference | 49 |
| | 3.16.1. | External Reference | 49 |
| | 3.16.2. | External Clock (Continuous) | 50 |
| | 3.16.3. | · · · · · · · · · · · · · · · · · · · | |
| | 3.17. | AS bus Operation | |
| | 3.17.1. | Channel Numbering with AS bus | 54 |
| | 3.17.2. | ee e | |
| | 3.18. | Special Operating Modes | 55 |
| | 3.18.1. | Normal Acquisition (mode 0) | 5 6 |
| | 3.18.2. | Stream Data (mode 1) | |
| | 3.18.3. | Averager (mode 2) | |
| | 3.18.4. | Dual Memory (mode 3) | |
| | 3.18.5. | Peak ^{TDC} (mode 5) | |
| | 3.18.6. | Frequency Counter (mode 6) | |
| | 3.18.7. | SSR – Sustained Sequential Recording (mode 7) | |
| | 3.18.8. | DDC – Digital Down-Conversion (mode 12) | |
| | | Readout of Battery Backed-up Memories | |
| | | Preparations before Power-Off | |
| | 3.19.2. | Recovery after Power-Off | |
| | | Reading the Instrument Temperature | |
| | 3.21. | Building applications that can Hibernate | 60 |
| 4. | APPEN | NDIX A: ESTIMATING DATA TRANSFER TIMES | 61 |
| • | | inciples & Formulas. | |
| | | camples | |
| | | omparison Chart for Typical Transfers | 63 |

1. Introduction

1.1. Message to the User

Congratulations on having purchased an Agilent Technologies Acqiris data conversion product. Acqiris Instruments are high-speed data acquisition modules designed for capturing high frequency electronic signals. To get the most out of the products we recommend that you read the accompanying product User Manual, this Programmer's Guide and the Programmer's Reference manual carefully. We trust that the product you have purchased as well as the accompanying software will meet with your expectations and provide you with a high quality solution to your data conversion applications.

1.2. Using this Manual

This guide assumes you are familiar with the operation of a personal computer (PC) running a Windows 2000/XP/Vista/7 (32/64) or other supported operating system. In addition you ought to be familiar with the fundamentals of the programming environment that you will be using to control your Acqiris product. It also assumes you have a basic understanding of the principles of data acquisition using either, a waveform digitizer, a digital oscilloscope, or other similar instrument.

The **User Manual** that you also have received (or have access to) has important and detailed instructions concerning your Acqiris product. You should consult it first. You will find the following chapters there:

- Chapter 1 *OUT OF THE BOX*, describes what to do when you first receive your new Acqiris product. Special attention should be paid to sections on safety, packaging and product handling. Before installing your product please ensure that your system configuration matches or exceeds the requirements specified.
- Chapter 2 *INSTALLATION*, covers all elements of installation and performance verification. Before attempting to use your Acqiris product for actual measurements we strongly recommend that you read all sections of this chapter.
- Chapter 3 **PRODUCT DESCRIPTION**, provides a full description of all the functional elements of your product.

This Programmer's Guide is divided into 3 separate sections.

- Chapter 1 *INTRODUCTION*, describes what can be found where in the documentation and how to use it.
- Chapter 2 **PROGRAMMING ENVIRONMENTS & GETTING STARTED**, provides a description for programming applications using a variety of software products and development environments.
- Chapter 3 **PROGRAMMING AN ACQIRIS INSTRUMENT**, provides information on using the device driver functions to operate an Acqiris instrument.

The accompanying **Programmer's Reference manual** is divided into 2 sections.

- Chapter 1 **INTRODUCTION,** describes what can be found where in the documentation and how to use it.
- Chapter 2 **DEVICE DRIVER FUNCTION REFERENCE**, contains a full device driver function reference. This documents the traditional Application Program Interface (API) as it can be used in the following environments:

Visual C++, LabVIEW, MATLAB

Programmer's Guide Page 5 of 64

1.3. Conventions Used in This Manual

The following conventions are used in this manual:



This icon to the left of text warns that an important point must be observed.

WARNING Denotes a warning, which advises you of precautions to take to avoid being electrically shocked.

CAUTION Denotes a caution, which advises you of precautions to take to avoid electrical, mechanical, or

operational damages.

NOTE Denotes a note, which alerts you to important information.

Italic text denotes a warning, caution, or note.

Bold Italic text is used to emphasize an important point in the text or a note

mono text is used for sections of code, programming examples and operating system commands.

Certain features are common to several different modules. For increased readability we have defined the following families:

DC271-FAMILY U1061A/U1064A/U1069A DC135/DC135HZ/DC140/DC140HZ/DC211/

DC211A/DC241/DC241A/DC271/DC271A/

DC271AR/DP214/DP235/DP240

AP-FAMILY U1081A/U1082A AP240/AP235/AP100/AP101/AP200/AP201 12-bit-FAMILY U1066A/U1070A DC440/DC438/DC436/DP310/DP308/DP306

10-bit-FAMILY U1062A/U1065A DC122/DC152/DC222/DC252/DC282

U1071A-FAMILY all U1071A variants, DP1400, U1091AD28

U1084A-FAMILY

1.4. Warning Regarding Medical Use

The Agilent Technologies Acqiris cards are not designed with components and testing procedures that would ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of these cards involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user. These cards are *not* intended to be a substitute for any form of established process or equipment used to monitor or safeguard human health and safety in medical treatment.



WARNING:

The modules discussed in this manual have not been designed for making direct measurements on the human body. Users who connect an Acqiris module to a human body do so at their own risk.

1.5. Warranty

Please refer to the appropriate User Manual.

1.6. Warranty and Repair Return Procedure, Assistance and Support

Please refer to the appropriate User Manual.

1.7. System Requirements

Please refer to the appropriate User Manual.

Programmer's Guide Page 6 of 64

2. Programming Environments & Getting Started

Agilent Technologies supplies sample programs as a starting point for the development of user-specific Acqiris applications. For Windows systems there are samples for the Visual C/C++, and MATLAB.

The API has been split into three families:

- Acqrs Generic functions AqBx these can be used for all Acqiris Instruments
- AcqrsD1 Digitizer functions AqDx to be used for Digitizers and Analyzers

All of these functions are still contained in one library called **AgMD1Fundamental**. The LabView interface is split into the two corresponding AqXX parts. The AcqrsD1 section includes redundant copies of the generic functions so that backward calling compatibility can be maintained for existing code.

Preliminary remark: it is assumed in the following that the hardware and Acqiris software installations (see User Manual chapter 2) have already been completed.



NOTE: Visual C/C++ relies on the standard VISA types defined by VXIplug&play Systems Alliance (VISA). The **visatype.h** include file can be found in the include directory created at installation.

2.1. Visual C++

For digitizer users:

- Examine the code of the examples, *.cpp, to identify one which is most relevant
- Open the project file, .vcproj for Visual Studio 2008 of the example and build the project.
- Note that when writing your own application you should insert the line :

```
#include "AgMD1Fundamental.h"
```

at the beginning of every file that will access Acqiris D1 Driver functions.

• The project should link to the **AgMD1Fundamental.lib** or **AgMD1Fundamental_64.lib** file.

2.2. LabVIEW



NOTE: All new software development should use recent platforms, i.e. LV 8.x. Support for LV 7.1 will continue only for the older models (the LLBs are frozen as of Release 3.2). New models/families will only be supported on LV 8.x.

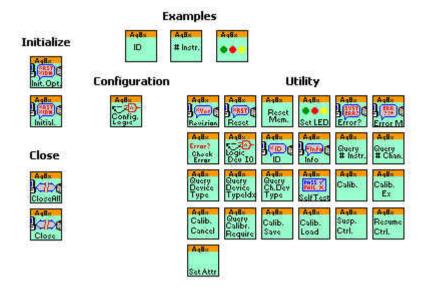
The AqXX (Acqiris Digitizer) driver for LabVIEW conforms to National Instruments' Instrument Driver Standard. If LabVIEW is installed on the target machine when the Acqiris software is installed, the AqXX driver interfaces will be copied to the LabVIEW n\Instr.lib directory. The driver function VI's can then be found on the Functions palette (block diagram), Instrument Drivers subpalette. There is also a Getting Started VI, as well as some example VI's. The standard API help file is available from within LabVIEW. The Revision Query VI gives information on the current version. There are actually many llb files, AqBx.llb, AqTx.llb, and AqDx.llb containing the routines shown below. The AqDx obs.llb has deprecated but still usable routines.

Under LabView8 the two families discussed above have been placed into two projects. Each project contains Examples, Private, and Public Folders. The project folder also contains an Acqiris Xx Readme.html giving version information.

The Example folder contains ready to use potential starting points for your own programs. Individual library functions can be found at the Public level or in one of the 4 categories shown. Query functions are in the Utility folder. The block diagram of the VI Tree.vi can be used to access individual functions. It allows quick and easy access to all needed VI's.

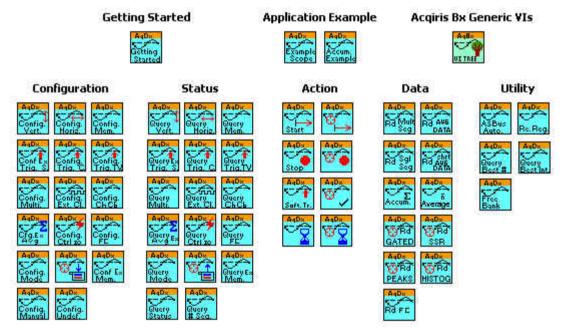
Programmer's Guide Page 7 of 64

The AqBx Generic functions family contains the following vi's.



The AqBx Examples section contains three very simple cases to show some basic functionality.

The AqDx Digitizer functions family contains the following vi's. The diagram also shows the AqBx tree. These AqBx functions are to be preferred over their AqDx equivalents.



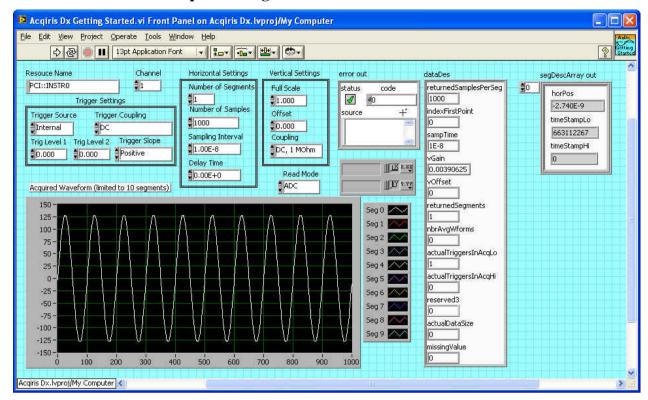
2.2.1. LabView 7.x/8 with old LabView programs

If you have an existing LabView program you can still use it or develop further under LabView 8. You can continue to use the VI's and libraries of the LabView 7.1 environment.

If you are still using the AqDx versions under LabView 7.x they will not immediately be found when you load your vi. LabView will ask you which library should be used; you should enter the name of the library of obsolete functions, $AqDx_obs.llb$.

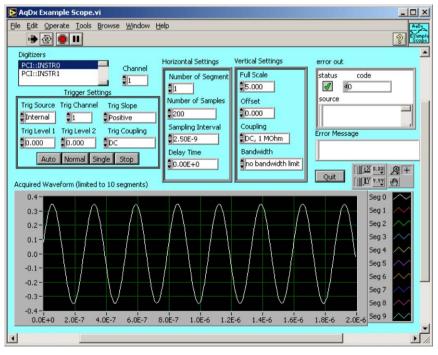
Programmer's Guide Page 8 of 64

2.2.2. AqDx Getting Started VI



This VI demonstrates how to use some of the basic components of the AqDx Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires one waveform. Note that the front panel controls should be set to their desired values before the VI is run.

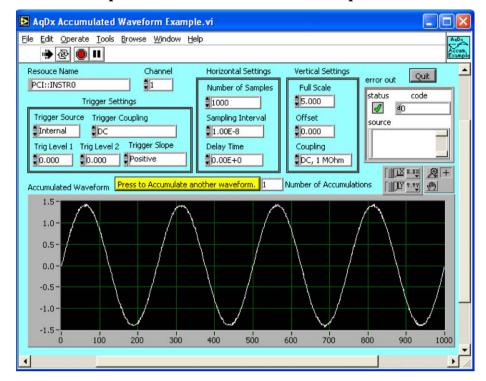
2.2.3. AqDx Example Scope VI



This VI presents a basic, interactive virtual oscilloscope using the AqDx Digitizer Driver. Not all the functionality of the Acqiris digitizers is supported in this program, but most of the most-commonly-used functions are demonstrated in it.

Programmer's Guide Page 9 of 64

2.2.4. AqDx Accumulated Waveform Example VI



This VI demonstrates how to use the Accumulate Waveform function of the Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires and accumulates waveforms as the user requests. Note that the front panel controls should be set to their desired values before the VI is run.

2.3. MATLAB

MATLAB is a very powerful environment to analyze and display data. The MEX interface can be used with MathWorks MATLAB 7.3 or a newer version. The interface from it to the Acqiris products offers simple direct access to the Acqiris driver. An example is in the directory **AcqirisDxRoot>MATLAB**.

To use it,

either set the MATLAB current directory (cd Command) to <AcqirisDxRoot>\MATLAB

or the file from that directory to the directory of your choice.

GetStarted

Filename: **Aq_GetStarted.m**Argument: No Argument

This first example shows you how to perform a single acquisition. The desired configuration is loaded. You then start the acquisition, and read and plot the acquired data.

2.4. Linux

For information on the Linux installation and included examples please refer to the **README** file which is located in the /**Linux** folder in the root of the supplied installation CDROM.

Programmer's Guide Page 10 of 64

3. Programming an Acqiris Instrument

3.1. Programming Hints

When programming an Acqiris instrument it is important to remember that the Acqiris driver must be freshly loaded (this is usually automatic) by any process that uses the modules. This means that each process starts over with a completely clean view of the system and no knowledge of any previously determined calibration constants or settings. Thus a calibration ought to be done, or loaded, before the modules are used for any acquisitions. Of course the system may have to be recalibrated later if the temperature of the modules is changing. Users cannot expect to control Acqiris modules with a succession of process invocations with each one executing a single command.

The Agilent Acqiris AqDrv device driver is *thread-safe*, i.e. its API can be called simultaneously from multiple threads. All of the driver functions are reentrant. The thread safety is guaranteed by using instrument-based mutual exclusion for functions taking an instrument handle as input parameter, and a global lock for the other functions. This means that access to a given instrument is serialized using a lock, ensuring that only one thread reads from or writes to that instrument (and associated variables) at any time. However, calls to functions to *discover*, *initialize* or *close* instruments, while the instrument(s) are still in use in other threads, should be avoided to prevent errors due to concurrent attempts to obtain the global lock. Finally, remember that even though the driver is thread-safe, great care is always required when designing applications using multiple threads to avoid problems such as race conditions or starvation.

Be sure to read the comments on the functions and their parameters in chapter 2, **DEVICE DRIVER FUNCTION REFERENCE** of the **Programmer's Reference Manual**. Another valuable source is the header files: **AgMD1Fundamental.h,** for C-like environments.

The examples below do not check the return value of the **Acqrs_...** functions. In real applications, you should always check the return values of functions.

3.2. Device Initialization

Before any (real or simulated) device can be used, each device must be initialized with a separate call to the function **Acqrs_InitWithOptions**. For real devices, you can also use the slightly simpler function **Acqrs_init**. Both functions return the **instrumentID** (whose value will be different for each device), which must be subsequently used in any other function call. The arguments **IDQuery** and **resetDevice** are currently ignored. The use of the string arguments **resourceName** and **optionsString** are explained with the initialization scenarios in the following sections.

If you use modules that are connected via AS bus, you need to configure them as *MultiInstruments*. This lets you treat them as normal instruments with an increased number of channels. E.g. you can connect 3 DC270's to form a single 12-channel, 1 GS/s digitizer.

If needed, and before initializing the devices, a call to either

Acqrs_getNbrInstruments can be used to learn how many instruments have been found. However, a side-effect of this call will be to select the use of single instruments. This can be manually circumvented as discussed below.

or

AcqrsD1_multiInstrumentAutoDefine can be used to automatically combine as *MultiInstruments* modules that are connected via AS bus and return the total number of instruments found, including individual modules without AS bus connections. It automatically searches for all sets of modules that are connected with AS bus, and configures each such block as a single *MultiInstrument*.

As an alternative to automatic *MultiInstrument* definition, you can initialize each module individually with the function **Acqrs_InitWithOptions**, and then combine some of them with the function **AcqrsD1_multiInstrDefine**. This method provides better control over which modules are combined and in what order, at the expense of careful book keeping of which instrumentID's are available. For details, please refer to the section 3.2.8, **Manual Definition of MultiInstruments**.



NOTE: Slow instrument detection may occur on systems with a VISA installation. To skip the VISA detection, you may edit the "AgMD1Fundamental.ini" file and manually add the string below. Note that VXI instruments use VISA, and that this procedure can therefore not be used. The "AgMD1Fundamental.ini" file will be searched in the directory IVIROOTDIR32\Driver\AgMD1 or IVIROOTDIR64\Driver\AgMD1.

DDRMANAGER_NO_VISA=1

Programmer's Guide Page 11 of 64

3.2.1. PCI & VXI Identification by Order Found

If you don't know which and/or how many Acqiris instruments are present on the machine, use this code fragment:

```
ViSession instrumentID[10];
long nbrInstruments;
ViStatus status;
ViString options = "";
status = Acqrs_getNbrInstruments(&nbrInstruments);
// Initialize the instruments
for (long i = 0; i < nbrInstruments; i++)
{
    char resourceName[20];
    sprintf(resourceName, "PCI::INSTR%d", i);
    status = Acqrs_InitWithOptions(resourceName, VI_FALSE, VI_FALSE, options, &(instrumentID[i]));
}</pre>
```

The resource name must be of the form "PCI::INSTR0", "PCI::INSTR1", etc. This is true in spite of the fact that all PCI, cPCI, and VXI instruments will be found. The integer part of the resource name will be referred to as the **deviceIndex**. The API contains a function, **Acqrs_getDevTypeByIndex**, to allow you to determine the family of an instrument before actually initializing it.

If there are several instruments in the system, the order in which they are found is not obvious. It depends on the Windows 2000/XP/Vista/7 (32/64) Configuration Manager implementation, on the PCI bus topology in your computer, and possibly on the BIOS.

3.2.2. PCI Identification by Serial Number

All Acqiris instruments are labeled with a unique serial number. For PCI digitizers you will find it on the front panel and for CompactPCI instruments it is on the right injector/ejector handle. This same serial number is coded into an on-board EEPROM that is read by the Device Driver upon initialization. You can therefore ask to initialize a specific instrument by specifying its serial number:

Note that the serial number must be contiguous to the keyword SER; leading zeros are accepted.

3.2.3. PCI Identification by Bus/Slot Number

While initialization by serial number is easy to implement, it has the drawback that anytime an instrument is replaced by another one (e.g. if a failure occurred), the program has to be modified. The Acqiris API offers the possibility of specifying the logical position of the device at initialization:

Again, the bus and slot numbers must be contiguous to the keywords BUS and SLOT; leading zeros are accepted.

Unfortunately, it is not obvious at all by simple inspection, which bus and slot number a given PCI device occupies. One way to find out is to use AcqirisLive and to observe the bus/slot numbers that can be found under the Help menu selection in Instrument Information. Another way is to use the auto-identification initialization method and then to interrogate each device with:

```
ViSession instrumentID;
char name[20];
long serialNbr, busNbr, slotNbr;
Acqrs_getInstrumentData(instrumentID, name, &serialNbr, &busNbr, &slotNbr);
```

Programmer's Guide Page 12 of 64

3.2.4. VXI Identification

Instruments in IX20x VXI Carrier modules will also be found by the driver. The resource name will be in the form "VXI[board]::[logical_addr]::INSTR" like "VXI0::1::INSTR".

3.2.5. PXI VISA & LabViewRT Identification

The driver can also be used in the VISA environment. In this case the resource name has the following allowed forms:

```
"PXI<br/>bus>::<device>::INSTR"
"PXI0::<br/>bus>-<device>::INSTR"
"PXI0::CHASSIS<chassis>::SLOT<slot>::INSTR"
```

3.2.6. Firmware initialization (AP-FAMILY/12-bit-FAMILY/AC/SC/TC)

In these modules the on-board FPGA's (field-programmable gate arrays) contain processor logic needed to efficiently execute several crucial functions. For Windows and Linux users, they will be automatically programmed at startup before calibration. The standard initialization using **Acqrs_init** can be used.

The name for the FPGA program file is a synthesis of model, FPGA destination, and option information. The file name suffix is always ".bit". The automatic initialization mentioned above will load the FPGA files as follows:

- For the first time initialization of a module needing an FPGA file, the desired file will be searched for in the working directory.
- If the above doesn't succeed then the working directory will be searched for a file "AgMD1Fundamental.ini"
- Finally the directories IVIROOT32\Driver\AgMD1 and IVIROOT64\Driver\AgMD1 will be searched for a file "AgMD1Fundamental.ini"
- The "AgMD1Fundamental.ini" file should contain the name of a directory which will also be searched for the appropriate FPGA files. Here is a typical example of its contents:

```
[Acqiris]
fpgaPath=C:\Program Files\Acqiris\firmware
GeoMapPath=C:\Program Files\Acqiris\bin
```

The GeoMapPath entry will be described later in this chapter.

• The final path used will be remembered and used for all subsequent demands for this module. In particular this applies if **AcgrsD1 configMode** is used to change functionality.

3.2.7. Automatic Definition of MultiInstruments

The function **AcqrsD1_multiInstrumentAutoDefine** automatically searches for all sets of modules that are connected with AS bus, and configures each such block as a single *MultiInstrument*. It then reports the total number of instruments found, including individual modules without AS bus connections. You still need to retrieve the **instrumentID** for each instrument by calling the function **Acqrs_InitWithOptions** afterwards, as shown below:

```
ViSession instrumentID[10];
long nbrInstruments;
ViStatus status;
ViString options = "";
status = AcqrsD1_multiInstrAutoDefine(options, &nbrInstruments);
// Retrieve the instrument identifiers
    for (long i = 0; i < nbrInstruments; i++)
    {
        char resourceName[20];
        sprintf(resourceName, "PCI::INSTR%d", i);
        status = Acqrs_InitWithOptions(resourceName, VI_FALSE, VI_FALSE, options, &(instrumentID[i]));
    }
}</pre>
```

Programmer's Guide Page 13 of 64

The calls to **Acqrs_InitWithOptions** are needed to obtain the **instrumentID**'s. The physical digitizers were already initialized when **AcqrsD1_multiInstrumentAutoDefine** was called.

The digitizers within a single *MultiInstrument* are numbered from 0 to (NbrModulesInInstrument – 1). In Acqiris CC10x compactPCI crates, the module 0 is always closest to the controller slot, i.e. module numbers increase with increasing front panel slot numbers. This statement is also applicable to CC121 crate configurations with both an Acqiris acquisition module in one of the last 7 slots and with the PC running under Windows 2000/XP/Vista/7 (32/64). Users of other systems or crates may need to provide AqGeo.map files to give the driver needed information. The GeoMapper application described in the User Manuals can create this file. For details on channel and trigger source numbering, please refer to section 3.16, *AS BUS OPERATION*.

The master module is automatically chosen, according to these rules:

AS bus

- If modules of different memory lengths are combined, only modules with the shortest memory length can be
 master modules
- The master module is chosen as near as possible to the center module, in order to minimize propagation delays.

AS bus 2 U1065A 6U digitizers

• The master module is chosen as near as possible to the center module, in order to minimize propagation delays. There can be at most 5 modules in the *MultiInstrument*.

AS bus 2 U1062A 3U and U1071A PCI digitizers

• The master module will be the rightmost of up to 3 modules.

The function **Acqrs_getInstrumentData** will return the information about the master module. If you want control over which module is the master, and in which order they should appear, use the manual definition, described in the next section.

3.2.8. Manual Definition of MultiInstruments

The function **AcqrsD1_multiInstrDefine** permits a 'manual' definition of how to combine multiple digitizers with the AS bus. It cannot be used for AS bus 2 instruments.

Use a code fragment like the following one for the manual combination of digitizers:

```
ViSession instrumentID[10], idList[6];
long nbrInstruments;
ViStatus status;
ViString options = "cal=0"; // since calibration will be performed explicitly
status = Acqrs_getNbrInstruments(&nbrInstruments);
// Initialize the digitizers
for (long i = 0; i < nbrInstruments; i++)</pre>
     char resourceName[20];
     sprintf(resourceName, "PCI::INSTR%d", i); status =
     Acgrs_InitWithOptions(resourceName, VI_FALSE, VI_FALSE, options,
     &(instrumentID[i]));
// Now combine the first 3 digitizers (in inverse order)
idList[0] = instrumentID[2];
idList[1] = instrumentID[1];
idList[2] = instrumentID[0];
ViSession multiInstrID;
ViSession masterIndex = instrumentID[1];
long nbrInList = 3;
status = AcqrsD1_multiInstrDefine(idList, nbrInList, masterIndex,
&multiInstrID);
status = Acqrs_calibrate(multiInstrID);
```

Programmer's Guide Page 14 of 64

The first part of the code above finds and initializes all individual digitizers, as shown in section 3.2.1. Of course, you could also use one of the other 2 methods of identifying individual digitizers.

After **AcqrsD1_multiInstrDefine** has executed successfully, the **instrumentID's** in the list **idList** become unavailable for further operations. You must use the returned **multiInstrID** to refer to the newly defined *MultiInstrument*.

You are responsible for making sure that

- all participating digitizers are physically connected with AS bus bridges
- the modules are of the same model type
- the master module has no more memory than any other participating digitizer
- an AcqGeo.map file is available if needed by the driver

If the master module has more memory than any other digitizer, the combined instrument will work as long as you never request more memory than that available in the unit with the shortest memory.

The digitizers within the 'manually' defined *MultiInstrument* are numbered from 0 to (nbrInList -1), exactly as presented in the digitizer list **idList**. For details on channel and trigger source numbering, please refer to section 3.16, *AS BUS OPERATION*.

3.2.9. AqGeo.map file positioning

For AS bus MultiInstruments in some systems the driver will need additional information about the physical ordering of the modules. This information is stored in a file named AqGeo.map which the driver will load when an AS bus instrument is defined. The driver will search for the AqGeo.map file as follows:

- First the file will be searched for in the working directory of the application.
- Then the working directory will be searched for a file "AgMD1Fundamental.ini"
- Finally the IVIROOT32\Driver\AgMD1 and IVIROOT64\Driver\AgMD1 will be searched for a file "AgMD1Fundamental.ini"
- The "AgMD1Fundamental.ini" file should contain the name of a directory which will also be searched for the AgGeo.map file. Here is a typical example of its contents:

```
[Acqiris]
GeoMapPath=C:\Program Files\Acqiris\bin
```

3.2.10. Simulated Devices

If you want to work with simulated devices, none of the methods above are applicable. Many digitizer modules supported by the driver can be simulated; analyzers and averagers generate data in digitizer mode. Any memory option available for the module can be used for a call to **Acqrs_setSimulationOptions** as shown in this code fragment:

The first function call sets the instrument options that you want to obtain, e.g. "M2M" for the long memory option of a DC110. In the second call, you must specify "simulate=TRUE" (without any spaces!). The device driver creates a simulated device of your choice. The resource name string is always of the form "PCI::aannn", where aannn is a valid instrument module name.

The simulation options will apply to all subsequent calls to $Acqrs_InitWithOptions$, until they are reset with $Acqrs_setSimulationOptions$ ("").

Programmer's Guide Page 15 of 64

3.2.11. Terminating an Application

For an orderly shut down of your application, we recommend the following sequence:

```
// Stop the instruments
for (long i = 0; i < nbrInstruments; i++)
{
    status = AcqrsD1_stopAcquisition(instrumentID[i]);
}
Acqrs_closeAll();</pre>
```

Stopping the acquisition of all instruments ensures that there is no further activity that could, for example, generate an interrupt. The function **Acqrs_closeAll** shuts down the driver components in the correct order, and thus helps avoid crashes of the application during closing

3.2.12. Reinitialization

If the operating system of the computer goes into hibernate or stand-by then the driver will return an error, ACQIRIS_ERROR_IO_DEVICE_OFF, when the application starts to use the hardware again. In this case the device will not be useable until an Acqrs Init... routine is called.

3.3. Device Configuration

As a general rule it should be remembered that new values to be used by the modules, as set with the AcqrsXX_config functions, are remembered by the driver software but not immediately acted upon. They will only really be loaded into the instrument's registers at the beginning of an acquisition when AcqrsXX_acquire is called. At that time all necessary changes will be made and, depending on the type of changes, the driver will force itself to wait the appropriate settling time before it starts the acquisition. This is done to ensure that the acquisition will occur in the desired state. The program does not have to include *ad hoc* waits to allow the analog hardware to settle. The settling times vary from none in the case of no change, to ~0.5 ms in the case of offset changes and to ~5 ms for relay changes or changes between normal operation and External Clock Reference. In the special case of switching from DC to AC coupling, the settling time is 100 ms. Time base setting changes also have associated settling times.



NOTE: The special case of **making transitions from low to high impedance** is treated immediately to avoid the risk of damaging the front-end circuitry of the digitizer. **When making transitions from high to low impedance you must ensure that large voltages are not applied before the change has really happened**. Therefore it is recommended to wait ~5 ms after having asked for an impedance change and before applying any large voltages. Impedance changes can also affect other devices in the signal path.

Unneeded calls to the AcqrsXX_config functions should be avoided because they can delay the start of the next acquisition.



NOTE: The AcqrsXX_get functions return the configuration values to be used for the next acquisition.

Use the following short code fragment for a device configuration:

```
// Configure
double sampInterval = 1.e-9, delayTime = 0.0;
long nbrSamples = 10000, nbrSegments = 1;
long channel = 1, coupling = 1, bandwidth = 0;
double fullScale = 2.0, offset = 0.0;
long trigCoupling = 0, trigSlope = 0;
double trigLevel = 20.0; // in % of vertical Full Scale !
AcqrsDl_configHorizontal(instrID, sampInterval, delayTime);
AcqrsDl_configMemory(instrID, nbrSamples, nbrSegments);
AcqrsDl_configVertical(instrID, channel, fullScale, offset, coupling, bandwidth);
AcqrsDl_configTrigClass(instrID, 0, 0x00000001, 0, 0, 0.0, 0.0);
AcqrsDl_configTrigSource(instrID, channel, trigCoupling, trigSlope, trigLevel, 0.0);
```

Programmer's Guide Page 16 of 64

Comments:

- Channel numbers run from 1 to nbrChannels, not from 0! Segment numbers, however, run from 0 to (nbrSegments 1).
- If the desired sampling interval implies that multiple converters/channel will be needed a call to **AcqrsD1 configChannelCombination** will be needed *before* the call to **AcqrsD1 configHorizontal**.
- Specifying more than 1 segment in **AcqrsD1_configMemory** implies the use of Sequence mode. The 10-bit-FAMILY, U1071A-FAMILY and U1084A-FAMILY instruments offer additional functionality through **AcqrsD1_configMemoryEx**. This includes the Simultaneous multibuffer Acquisition and Readout (SAR) mode to allow a higher loss-less trigger rate. For such units with extended memory there is the possibility of restricting memory use to the internal memory to reduce the maximum dead time between segments of a sequence acquisition.
- The 5 main configuration functions are protected against illegal or incoherent values. Thus, the system might adapt the values you ask for. There are 5 'query' counterparts to these functions, AcqrsD1_getHorizontal, AcqrsD1_getMemory, AcqrsD1_getVertical, AcqrsD1_getTrigClass, and AcqrsD1_getTrigSource, which you can interrogate.
- The function AcqrsD1_configTrigClass configures the trigger class control parameters of the digitizer. For most Acqiris products, the edge trigger class is the only class available. For this class, the available source patterns are Channel 1 through 4 or the external triggers. The AcqrsD1_configTrigSource function configures the source parameters coupling, slope, and level as shown in the example above. Notice that the functions AcqrsD1_configTrigClass and AcqrsD1_configTrigSource must always be used together in order to complete the setup of the trigger configuration. Refer to chapter 2, DEVICE DRIVER FUNCTION REFERENCE of the Programmer's Reference Manual for a detailed description of these two functions.
- The helper functions AcqrsD1_bestSampInterval and AcqrsD1_bestNominalSamples are sometimes useful for deciding on the nominal number of data points and the sampling interval to use for a given time window to cover. If you ask for a nominal number of samples, the system actually needs some additional samples for reasons of data alignment, acquisition stop-time overhead and other reasons. In some cases, the additional 'invisible' samples can exceed the number of 'visible' ones. The helper functions take such memory overheads into account when advising you on the recommended sampling interval and number of samples. You are free to ignore the advice, but the system is likely to adapt your setup values if the requested number of samples does not fit the available memory.
- Specifying the value 0 for **delayTime** sets the trigger point to the beginning of the waveform. A negative value corresponds to pre-trigger, a positive one to post-trigger. Refer to the section 3.11, TRIGGER DELAY AND HORIZONTAL POSITION for a detailed explanation of the use of **delayTime**.
- For DC coupling the trigger levels in %FS as needed by **AcqrsD1_configTrigSource** can be calculated as follows:
 - TriggerLevelPercent = 100*(TriggerLevelVolts + vOffset)/Fsrange;
- The granularity of a trigger value setting is limited by the hardware that uses an 8-bit DAC covering somewhat more than the desired range.
- To set the external trigger range for a DC271-FAMILY, a 10-bit-FAMILY, or a U1071A-FAMILY module, or an AP240/AP235 signal analyzer platform, add a call to **AcqrsD1_configVertical** with channel = -1 before the call to **AcqrsD1_configTrigSource**.
- The 10-bit-FAMILY, U1071A-FAMILY and U1084A dual source pattern triggers can be activated by calling
 the function AcqrsD1_configTrigClass with an appropriate TrigClass and a sourcePattern indicating the two
 desired sources.

Programmer's Guide Page 17 of 64

3.4. Configuring Averagers

3.4.1. Basic configuration

The averagers have 2 operational modes, digitizer and averager, controlled with the function:

```
AcqrsD1_configMode(instrID, mode, 0, 0);
```

The value *mode* can be set to 0 (digitizer) or 2 (averager).

The averager mode uses a number of additional configuration parameters, which describe the requested averaging conditions.

Use the following short code fragment to configure an average of 1000 waveforms of 20'000 data points, with a dithering range of \pm 15 ADC LSB's and a start delay of 128 samples for an AP100:

```
// Configure
long channelNbr = 0, nbrSamples = 20000, nbrWaveForms = 1000;
long ditherRange = 15, trigResync = 1;
long startDelay = 128, stopDelay = 0;
AcgrsD1 configAvgConfig(instrID, channelNbr,
                        "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "NbrWaveforms", &nbrWaveForms);
AcgrsD1_configAvgConfig(instrID, channelNbr,
                        "DitherRange", &ditherRange);
AcgrsD1_configAvgConfig(instrID, channelNbr,
                        "TrigResync", &trigResync);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "StartDelay", &startDelay);
AcgrsD1 configAvgConfig(instrID, channelNbr,
                        "StopDelay", &stopDelay);
```

Comments:

- The value *channelNbr* is usually 0. However, for multi-channel Signal Analyzer platforms (AP240/AP235, U1084A) it can take on the value of the desired channel for some of the parameters.
- When in averager mode, the following digitizer parameters are ignored:
 - delayTime of the function AcqrsD1_configHorizontal is replaced by "StartDelay" and "StopDelay"
 - nbrSamples and nbrSegments of the function AcqrsD1_configMemory are replaced by "NbrSamples" and "NbrSegments"
- For the AP family of averagers, the values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'averaging block size', which is *always* 16 in the AP100 or AP240/AP235 Dual-channel mode, and 32 in the AP200 or AP240/AP235 Single-channel mode. If the supplied value is not an integer multiple of the 'averaging block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) on an AP100, and to 224 (7 * 32) on an AP200. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- For the U1084A Averager, the values startDelay and stopDelay must be integer multiples of the 'averaging block size', which is 16 in Dual-channel mode, and 32 in Single-channel mode. The value nbrSamples must be a multiple of 256 in Dual-channel mode or 512 in Single-channel mode. You can query the actual value with the function **AcqrsD1_getAvgConfig**
- The value *startDelay* controls the time between the trigger and the first data sample that is to be added to the averager sum.
- The *stopDelay* permits the addition of an extra delay to the dead time between the averaging of subsequent waveforms. Its minimum value is zero.
- The *ditherRange* value may be between 0 (no dithering) and 15 (max dithering). Please refer to the next section for further explanations.

Programmer's Guide Page 18 of 64

3.4.2. Dithering

Dithering reduces the effect of non-ideal differential non-linearity of the analog-to-digital converter, by adding or subtracting small offsets to the input signal. The offset is constant during the acquisition of a single waveform, and then modified to another value during the next waveform.

The dithering range, N is programmable between 0 (no dithering) and 15, with the function

Dithering reduces the range of the ADC by N levels at the top and another N levels at the bottom. In order to avoid any undesirable effects, you should make sure that the signal range of interest is within the reduced ADC range.

To use dithering with the U1084A, it must be explicitly enabled in addition to setting the dither range. Use the following API call to enable dithering on the U1084A:

3.4.3. 'Fixed Pattern' Background Subtraction

If an averaging operation is executed while the input is open or no signal is applied, the averaged waveform should tend to a constant value with a standard deviation $\sigma_{avg} = \sigma / \sqrt{N}$, where N is the number of waveforms in the average, and σ is the standard deviation of a single waveform.

In reality, only random noise sources are averaged out, while those that are coherent with the sampling clock are not reduced. The open-input averaged waveform thus represents the 'fixed pattern' background of the averager. Subtracting this waveform from each subsequently acquired averaged waveform should result in more precise data.

In order to facilitate the acquisition of the 'fixed pattern' background, the Averager modules offer the following possibilities:

- disconnection of the input with the value *coupling* = 0 in the function **AcqrsD1_configVertical**
- For the AP family averagers only: acquisition of an averaged waveform *without* a trigger signal with the value 'TrigResync' = 2 (free run) in the function **AcqrsD1_configAvgConfig**. However, you get a better measurement of the 'fixed pattern' background if you acquire with the same trigger conditions as the averaged waveforms that will be corrected. Typically, it is better to continue using an external trigger signal, rather than 'TrigResync' = 2.

The 'fixed pattern' background should be acquired in the same conditions as the averaged waveforms that will be corrected. In particular, the dithering range and the number of waveforms should be the same.

Use the following code fragment to acquire a 'fixed pattern' background with a free- running trigger (AP family averagers only):

Programmer's Guide Page 19 of 64

```
long nbrPoints = ???;  // Should be the 'current' number of points!
long timeStampLo, timeStampHi, nbrReturnedSamples;
double horPos, sampTime;

// Read the Waveform directly to the Background buffer
double bckGndWform[nbrPoints];
AcqrsDl_readRealWform(instrID, 1, 0, 0, nbrPoints, bckGndWform,
&nbrReturnedSamples, &horPos, &sampTime, &timeStampLo, &timeStampHi);

// Restore the settings of the averager
AcqrsDl_configVertical(instrID, 1, fsr, offset, coupl, bwidth);
AcqrsDl_configAvgConfig(instrID, channelNbr, "TrigResync", &reSync);
```

Use the following code fragment to acquire a 'fixed pattern' background, assuming that the external trigger can be used and is already set:

The examples above assume that the background and the averaged waveforms are read in Volts. In this case, the background data points are simply subtracted from the averaged waveform.

However, if you read the background and the averaged waveforms as 32-bit sums, with the function

you must correct the average as follows:

```
corrWform[i] = waveformArray[i] - bckGndWform[i] + 128*nbrAvgWforms;
```

The last term corrects for the fact that the 32-bit data are unipolar and that for display purposes the corrected waveform should be in the middle of the vertical range if the averaged waveform is the same as the background.

3.4.4. Configuring Noise Suppressed Accumulation (NSA)

As discussed in the User Manual Family of Averagers the module can be configured to only accept data above a fixed threshold and, if desired, to shift the data in that case. Since these two values are expressed in Volts and used as ADC counts they have to be converted before use. The User Manual describes this transformation that depends on whether Data Inversion has been enabled. The NSA threshold functionality must be enabled and a threshold defined. If this has been done the NSA base subtraction can also be enabled and will be activated using the defined base value. The order of the calls to AcqrsD1_configAvgConfig is not important since the final decision is taken when the acquisition is started. However, the correct full scale should be selected with AcqrsD1_configVertical before setting the NSA levels. Here is an example:

Programmer's Guide Page 20 of 64

3.5. Configuring SSR Analyzers

3.5.1. Acquisition Parameters

The AP235/AP240 SSR analyzers have 2 operational modes, normal and SSR, controlled with the function:

```
AcqrsD1_configMode(instrID, mode, 0, 0);
```

The value *mode* can be set to 0 (normal) or 7 (SSR mode).

The SSR mode requires a number of additional configuration parameters that describe the requested acquisition and readout conditions.

Use the following short code fragment to configure a buffered acquisition sequence of 800 waveforms of 5000 data points, with a start delay of 128 samples:

```
// Common Configure
long nbrSamples = 5000, nbrSegments = 800;
long startDelay = 128, stopDelay = 0;
AcqrsD1_configMode(instrID, 7, 0, 0);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSegments", &nbrSegments);
AcqrsD1_configAvgConfig(instrID, 0, "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrID, 0, "StopDelay", &stopDelay);
```

Comments:

- The value of the third and fourth arguments to **AcqrsD1** configMode must always be 0.
- When in SSR mode, the following digitizer parameters are ignored:
 - delayTime of the function AcqrsD1_configHorizontal is replaced by "StartDelay" and "StopDelay"
 - *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments" in the function **AcqrsD1_configAvgConfig**.
- The values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'block size', which is *always* 16 in the AP240/AP235 Dual-channel mode, and 32 AP240/AP235 Single-channel mode. If the supplied value is not an integer multiple of the 'averaging block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) for a Dual-channel acquisition, and to 224 (7 * 32) for a Single-channel acquisition. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- The value *startDelay* controls the time between the trigger and when the first digitized data sample is stored. It should also be noted that when *startDelay* is 0, the first few data points, 5 in the case of Dual-channel mode and 10 in the Single-channel mode, will always be 0.
- The *stopDelay* permits the addition of an extra delay to the dead time before the acquisition of subsequent waveforms. Its minimum value may be zero.

Programmer's Guide Page 21 of 64

- Although not shown here, a call to **AcqrsD1_configControlIO** can be made in order to set a trigger veto time to be respected after the receipt of a *Prepare for Trigger* signal on a Control I/O connector. This feature is for AP101/AP201 analyzers only.
- Also not shown, is a call to the function **AcqrsD1_configAvgConfig** to set a timeout value for the automatic completion of a segment in case the real trigger never arrives. This feature is for AP101/AP201 analyzers only.

3.5.2. Readout configuration

There are two possible ways of reading the data when in the SSR mode: user gates, and threshold gates. In all cases the entire acquisition must be read; you cannot ask for fewer segments or points. If you want to read all of the data you should define the appropriate gate. These setting are also controlled through the **AcqrsD1_configAvgConfig** routine and therefore must be prepared before the acquisition is started. They can be set independently for each channel if desired.

For user gate readout you have to define the groups of data samples that you want to read for each segment. If needed new values for the gates can be defined during the acquisition process. They will become effective after the next call to **AcqrsD1_processData** or **AcqrsD1_acquire**. Here is some sample code:

```
long g_gateLengthSum[3];
long g_lastGate[3];
long channel = 1, gate =1;
AcqrsD1_configAvgConfig(instrID, channel, "GateType", &gate);
// you can define up to 4095 gates,
// GatePos and GateLength must both be multiples of 4
AqGateParameters configSetupData[100];
long configObj
                 = SSR_Default;
long gateSize = 1000;
// this will be the size we want to read
g_gateLengthSum[channel] = 0;
q lastGate[channel] = 1; // a very simple example
for(int q=0;q<q_lastGate[channel];q++)</pre>
 // the first gate starts with the first point
 configSetupData[g].GatePos
 configSetupData[g].GateLength = gateSize;
 q_gateLengthSum[channel] += configSetupData[g].GateLength + 8;
status = AcqrsD1_configSetupArray(instrID, channel, configObj,
                            g_lastGate[channel], configSetupData);
```

For threshold gate readout you have to define the threshold value in volts. Data values greater than this will be selected for readout. If desired you can use **AcqrsD1_configAvgConfig** with the "InvertData" parameter to choose data values less than the threshold. In addition you can define the number of data values before and after each selected value that you always want to see. This number is in the range 0 to 16. However, the value will always be rounded up to the next highest multiple of 4. If two consecutive selected values are 32 or more samples apart a new gate block will be generated. Otherwise, the current block will be continued. In all cases the data transferred will always be a multiple of 4 samples and it will start on a sample whose time position is a multiple of 4. Alternatively the number of data values before and the total number of values can be selected. Furthermore, a limit on the maximum number of gates per segment can be set.

Programmer's Guide Page 22 of 64

3.5.3. SSR Time stamps

The 'On-board' 10 MHz reference clock is used to increment a counter. The value of the counter is stored after the trigger of each new segment. The value of the counter can be read by the software as shown here:

```
double SSRtimeStamp;
Acgrs_getInstrumentInfo(instrID, "SSRTimeStamp", &SSRtimeStamp);
```

Since each channel is controlled by its own FPGA the time stamps for the same segment are not necessarily the same for the two channels. The command above works with the stamp of Channel 2.



In this release if the function is called before the first acquisition has been started the value returned will be 0!

It is possible to reset the time stamp when starting an acquisition using a call like

```
status = AcgrsD1_acquireEx(instrID,0,4,0,0);
```

or with a hardware signal on the P1 or P2 connectors. This can be done with a call like:

```
Long TSReset = 1;
status = AcqrsD1_configAvgConfig(instrID, channel, "P1Control",&TSReset);
```

3.6. U1084 Zero-Suppression mode

3.6.1. U1084A Zero-Suppression mode – Acquisition Parameters

The U1084A has two modes of operation, Normal and Zero-Suppress, controlled with the function:

```
AcqrsD1_configMode(instrID, mode, 0, flag);
```

The value *mode* can be set to 0 (*Normal*) or 7 (*Zero-Suppress*). The *flag* value 10 enables SAR mode, and is the only valid flag value for this mode.

The Zero-Suppress mode requires a number of additional configuration parameters that describe the requested acquisition and readout conditions.

Use the following short code fragment to configure a buffered acquisition sequence of 800 waveforms of 4096 data points, with a pre-sample of 128 samples:

```
// Common Configure
ViInt32 nbrSamples = 4096, nbrSegments = 800;
ViInt32 InvertData = 0;
ViInt32 PreSamples = 128, PostSamples = 0;

AcqrsD1_configMode(instrID, 7, 0, 10);
AcqrsD1_configAvgConfigInt32(instrID, 0, "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfigInt32(instrID, 0, "NbrSegments", &nbrSegments);
AcqrsD1_configAvgConfigInt32(instrID, 0, "PreSamples", &PreSamples);
AcqrsD1_configAvgConfigInt32(instrID, 0, "PostSamples", &PostSamples);
AcqrsD1_configAvgConfigInt32(instrID, 0, "PostSamples", &PostSamples);
AcqrsD1_configAvgConfigInt32(instrID, 0, "InvertData", &InvertData);
```

Programmer's Guide Page 23 of 64

Comments:

- The value of the third argument to **AcqrsD1_configMode** must always be 0.
- When in Zero-Suppress mode, the following digitizer parameters are ignored:
 - *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments" in the function **AcqrsD1_configAvgConfig**.
- The value *nbrSamples* defines the target number of samples to acquire in each segment (trigger) before zero-suppression is applied. The value of *nbrSamples* must be integer multiples of 2048 for U1084A in dual-channel mode, and 4096 in single-channel mode. If the supplied value is not an integer multiple of the above values, it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 12000 will be truncated to 10240 for a Dual-channel acquisition, and to 8192 for a Single-channel acquisition. You can query the actual value with the function **AcqrsD1 getAvgConfig**.
- The value *PreSamples* guarantees a number of samples before the first one satisfying the threshold condition. The value must be a multiple of 16 in Dual-channel mode, and a multiple of 32 in Single-channel mode.
- The value *PostSamples* guarantees a number of samples after the last one satisfying the threshold condition. The value must be a multiple of 16 in Dual-channel mode, and a multiple of 32 in Single-channel mode.
- The value *MaxSamplesPerSegment* determines the maximum number of actual data samples that will be stored per segment after the zero-suppression is applied, per channel. Only samples that are actually retained are counted. Any data above this index will be truncated. The actual number of samples stored in each segment will depend on the signal, the threshold configuration and the value of *maxSamplesPerSegment*. This parameter may be used to limit the amount of data stored in each segment (gate), which may be useful in tuning the acquisition to avoid skipped triggers due to the time required to perform the data readout.

3.6.2. U1084 Zero-Supression mode – Threshold configuration

The U1084A Zero Suppress mode supports up to 128 different thresholds which are applied in sequence. For each threshold, a sample position can be given from where to start applying this threshold.

Use the function **AcqrsD1_configSetupArray**() to configure the thresholds, passing a setupType value of 2, and a pointer to an array of **AqThresholdGateParametersU1084A** structures in the setupData parameter. A different set of threshold can be configured for each channel.

The AqThresholdGateParametersU1084A structure has the following fields:

| Name | Type | Description |
|------------------|----------|--|
| threshold | ViReal64 | The threshold value to use, in Volts. |
| nextThreshSample | ViUInt32 | The index of the sample in the segment, at which the Zero Suppression system should switch to the next threshold in the array. |
| Reserved | ViInt32 | Reserved field, must be 0 |

The thresholds are applied in the order given in the setup array, each for a given duration. For example, if setupData is defined as follows:

AgThresholdGateParametersU1084A setupData[4];

Then in each segment, setupData[0].threshold is applied from sample 0 (the first sample of the segment) up to sample number setupData[0].nextThreshSample-1. Starting at sample number setupData[0].nextThreshSample, the threshold setupData[1].threshold is used up to sample number setupData[1].nextThreshSample-1, etc.

The last entry in the setup array must have a value of 0xffffffff (2^32 - 1) for nextThreshSample, to indicate that it will be applied up to the end of the segment. Except for this special value, all values of nextThreshSample must be a multiple of 16 (for dual channel / non-combined operation) or 32 (for single-channel / combined x2 operation). In addition, the value of nextThreshSample of each array element must be strictly greater than the value of nextThreshSample of the preceding array element.

Programmer's Guide Page 24 of 64

3.6.3. U1084A Zero-Suppression mode – Readout configuration

The function AcqrsD1_readData() may be used to read the acquired data. The entire acquisition must be read, reading partial acquisitions is not supported. There are no out-of-band segment descriptors in this mode, the segment descriptors are embedded in the acquisition data. The AqReadParameters structure should be set up as follows:

readMode: ReadModeSSRW

• firstSegment: must be 0

• nbrSegments: equal to the value set with AcqrsD1_configAvgConfigInt32("NbrSegments")

• firstSampleInSeg: must be 0

• nbrSamplesInSeg: equal to the value set with AcqrsD1_configAvgConfigInt32("NbrSamples")

• segmentOffset: unused, set to 0

• dataArraySize: the size in bytes of the user-allocated buffer to which the data is transferred

segDescArraySize: must be 0

• flags: must be 0

• reserved, reserved2, reserved3: must be 0

The total size of the data available for readout may vary between acquisitions. The data buffer passed in the dataArray parameter must be big enough to accommodate the worst case, which is:

maxDataSize = nbrSegments*(nbrSamples + segmentHeaderSize + gateHeaderSize) + padding

where:

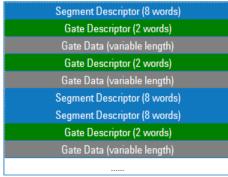
- *nbrSegments* is the value set with AcqrsD1 configAvgConfigInt32("NbrSegments")
- *nbrSamples* is the value set with AcqrsD1_configAvgConfigInt32("NbrSamples")
- *segmentHeaderSize* is 32 bytes (see also U1084A Zero-Suppression mode Data Format below)
- gateHeaderSize is 8 bytes (see also U1084A Zero-Suppression mode Data Format below)
- padding is 32 bytes

Check the actualDataSize field of the AqDataDescriptor structure to find out how many bytes have actually been transferred.

3.6.4. U1084A Zero-Suppression mode - Data Format

When operating in Zero-Suppress mode, the waveform readout contains digitized data and meta-data. The SAR mode is supported so that the user can start another acquisition and simultaneously read the previously acquired data.

In multi-segment mode, all of the segments must be read at once with a single read command. Digitized data and meta-data will be organized as follows:



Data readout structure

Notes:

- All segments will be read sequentially from the first to the last acquired segment.
- Each segment always starts with an 8-word (256-bit) Segment Descriptor.
- The Segment Descriptor will be followed by the zero-suppressed digitized data. The zero-suppress readout contains only portions of the waveform that are above the programmed threshold, each portion is called a 'Gate'.

Programmer's Guide Page 25 of 64

- Each Gate will start with a Gate Descriptor which describes the length of the gate and its position relative to the trigger of the segment.
- -If a segment does not contain any valid data, there will not be any Gate Descriptor or Gate Data, the next Segment Descriptor will follow immediately.

3.6.4.1. Segment Descriptor

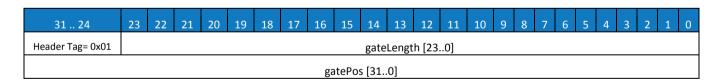
The Segment Descriptor is always 8 words (256 bits) and has the following format:

| 31 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------------------|--|----|----|----|----|----|----|----|-----|------|----|-------|------|------|-------|-----|---|---|---|---|---|---|---|---|
| Header Tag= 0x00 | Header Tag= 0x00 reserved | | | | | | | | | | | | | | | | | | | | | | | |
| | reserved | | | | | | | | | | | | | | | | | | | | | | | |
| | reserved | | | | | | | | | | | | | | | | | | | | | | | |
| | reserved | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | nbr | ates | | | | | | | | | | | | | | |
| reserved | reserved reserved D C T reserved reserved reserved | | | | | | | | | | | | | | | | | | | | | | | |
| reserved | | | | | | | | | | | Т | imeSt | tamp | High | า [15 | 50] | | | | | | | | |
| | TimeStampLow [310] | | | | | | | | | | | | | | | | | | | | | | | |

| Item | bits | Description |
|-------------------------------|------|--|
| Header Tag | 8 | A value of 0x00 indicates this is a segment descriptor of type1. A segment descriptor of type 1 is always 8 words (256-bits). |
| nbrGates | 32 | Indicate the exact number of gates that has been found in the segment. The end of each segment can only be determined by counting the number of gates read (<i>nbrGates</i>). |
| TimeStampHigh TimeStampLow | 48 | TimeStamp is the value of the time counter when the trigger occurs. The time counter can be reset to zero through the software user interface. The resolution of this counter will depend on which model and sampling rate is being used, please refer to the product datasheet for details. |

3.6.4.2. Gate Descriptor

Each Segment Descriptor is followed by zero or more Gate Descriptors, as indicated in the Segment Descriptor's *nbrGates* field. The Gate Descriptor has the following format:



| Item | bits | Description |
|------------|------|--|
| Header Tag | 8 | A value of 0x1 indicates this is a gate descriptor of type1. A gate descriptor of type 1 is always 2x4 bytes. The content is as described in this table. |
| gateLength | 24 | Number of samples that follow the gate descriptor. Possible values are 0x10 modulo 0x10 when the module is operating in 2 channel mode (2 x 2 GS/s non-combined) and will be 0x20 modulo 0x20 when the module is operating in single channel mode (1 x 4 GS/s combined). |
| gatePos | 32 | Absolute position of the first sample of the pre-gate data relative to the trigger of the segment, indicated in number of samples, this value may be used to reconstruct the data. |

Programmer's Guide Page 26 of 64

3.6.4.3. Gate Data

Each gate header is followed by gateLength samples of acquisition data. i.e. the next Gate Descriptor can be found by, next Gate Descriptorr = current header position + gateLength +2. The data is in 8-bit digitized raw unsigned format.

| Word | [3124] | [2316] | [158] | [70] |
|------|-----------|-----------|-----------|-----------|
| 1 | Sample 4 | Sample 3 | Sample 2 | Sample 1 |
| 2 | Sample 8 | Sample 7 | Sample 6 | Sample 5 |
| 3 | Sample 12 | Sample 11 | Sample 10 | Sample 9 |
| 4 | Sample 16 | Sample 15 | Sample 14 | Sample 13 |
| Etc | | | | |

3.7. Configuring AP family Peak TDC Analyzers

Since Peak^{TDC} processing can be viewed as an additional form of processing after SSR acquisition please refer to the discussion in 3.5 **Configuring SSR Analyzers** for that part of process. In addition you will have to

```
// Configure peak detection
long numberOfTriggersPerSeg = 50, numberOfSegments = 1;
double startDelta=0.1,validDelta = 0.2;
AcqrsD1_configMode(instrID, 5, 0, 0);
AcqrsD1_configAvgConfig(instrID, 0, "NbrRoundRobins", &numberOfTriggersPerSeg);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSegments", &numberOfSegments);
AcqrsD1_configAvgConfig(instrID, 0, "StartDeltaPosPeakV", &startDelta);
AcqrsD1_configAvgConfig(instrID, 0, "ValidDeltaPosPeakV", &validDelta);
// Configure histogram
long tdcMode = 1, tdcDepth = 1, tdcIncr = 2, tdcType = 1;
AcqrsD1_configAvgConfig(instrID, 0, "TdcHistogramMode", &tdcMode);
AcqrsD1_configAvqConfig(instrID, 0, "TdcHistogramDepth", &tdcDepth);
AcqrsD1_configAvgConfig(instrID, 0, "TdcHistogramIncrement", &tdcIncr);
AcqrsD1_configAvqConfig(instrID, 0, "TdcProcessType", &tdcType);
// if the acquisition has segments to be histogrammed independently
long tdcOverlay = 0;
AcqrsD1_configAvgConfig(instrID, 0, "TdcOverlaySegments", &tdcOverlay);
```

3.8. Configuring U1084A Peak^{TDC} Analyzers

In addition to the normal Digitizer setup, such as the trigger setup and full scale, you should configure the following settings when using the U1084A in mode:

```
// Configure peak detection
ViInt32 numberOfSamplesPerSeg = 4096;
ViInt32 numberOfSegments = 1;
ViInt32 numberOfTriggersPerSeg = 50;
ViReal64 startDelta=0.1,validDelta = 0.2; // Peak slope thresholds in Volts
AcqrsD1_configMode(instrID, 5, 0, 0);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSamples", &numberOfSamplesPerSeg);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSegments", &numberOfSegments);
AcqrsD1_configAvgConfig(instrID, 0, "NbrWaveforms", &numberOfTriggersPerSeg);
AcqrsD1_configAvgConfig(instrID, 0, "StartDeltaPosPeakV", &startDelta);
AcqrsD1_configAvgConfig(instrID, 0, "ValidDeltaPosPeakV", &validDelta);
```

If you wish to use peak interpolation, the following additional settings should be configured:

Programmer's Guide Page 27 of 64

```
// Configure peak interpolation
ViInt32 enableInterpolation = 1;
ViInt32 horzReso = 2;  // Increase in horizontal resolution, in bits
ViInt32 vertReso = 2;  // Increase in vertical resolution, in bits
AcqrsD1_configAvgConfig(instrID, 0, "InterpEnable", &enableInterpolation);
AcqrsD1_configAvgConfig(instrID, 0, "TdcHistogramHorzRes", &horzReso);
AcqrsD1_configAvgConfig(instrID, 0, "TdcHistogramVertRes", &vertReso);
```

The values *horzReso* and *vertReso* specify the number of bits to use for the fractional part of the peaks' interpolated position and amplitude, respectively.



NOTE: Enabling vertical interpolation will change the bit depth of each histogram bin, which affects the maximum possible number of triggers per segment and the interpretation of the bin values. Enabling horizontal interpolation will change the number of histogram bins per sample and thus the number of data points available for readout. See 3.10.9 Reading U1084A PeakTDC Waveforms and Histograms for more information.

3.9. Configuring AP101/AP201 Analyzers

The models AP101/AP201 have 2 operational modes, *normal* and *buffered* (also called *dual-memory*), controlled with the function:

```
AcgrsD1_configMode(instrID, mode, 0, flags);
```

The value mode can be set to 0 (normal) or 3 (dual-memory). In mode = 3, the parameter flags sets the memory bank into which to acquire (0 or 1).

The buffered mode uses a number of additional configuration parameters that describe the requested buffered acquisition conditions.

Use the following short code fragment to configure a buffered acquisition sequence of 800 waveforms of 5000 data points, with a start delay of 128 samples, into memory bank 1:

```
// Configure
long nbrSamples = 5000, nbrSegments = 800;
long startDelay = 128, stopDelay = 0;
AcqrsD1_configMode(instrID, 3, 0, 1);

AcqrsD1_configAvgConfig(instrID, 0, "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrID, 0, "NbrSegments", &nbrSegments);
AcqrsD1_configAvgConfig(instrID, 0, "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrID, 0, "StopDelay", &stopDelay);
```

Comments:

- The value of the second argument to **AcqrsD1_configAvgConfig** must always be 0.
- When in *buffered* mode, the following digitizer parameters are ignored:
- *delayTime* of the function **AcqrsD1_configHorizontal** is replaced by "StartDelay" and "StopDelay"
- *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments" in the function **AcqrsD1_configAvgConfig**.
- The values *nbrSamples*, *startDelay* and *stopDelay* must be integer multiples of the 'acquisition block size', which is *always* 16 in the AP101, and 32 in the AP201. If the supplied value is not an integer multiple of the 'acquisition block size', it is truncated to the next lower integer multiple. Thus, *nbrSamples* = 250 will be truncated to 240 (15 * 16) on an AP101, and to 224 (7 * 32) on an AP201. You can query the actual value with the function **AcqrsD1_getAvgConfig**.
- The value *startDelay* controls the time between the trigger and when the first digitized data sample is stored. The *stopDelay* permits the addition of an extra delay to the dead time between the acquisition of subsequent waveforms. Its minimum value may be zero.
- Although not shown here, a call to **AcqrsD1_configControlIO** can be made in order to set a trigger veto time to be respected after the receipt of a *Prepare for Trigger* signal on a Control I/O connector.
- Also not shown, is a call to the function **AcqrsD1_configAvgConfig** to set a timeout value for the automatic completion of a segment in case the real trigger never arrives.

Programmer's Guide Page 28 of 64

3.10. Data Acquisition

Instrument operation is preceded by configuring the instrument parameters and then starting the acquisition sequence. New settings are only loaded into the module when the acquisition is started; there is one exception to this rule as discussed for analyzer user gate definition in section 3.5.2 **Readout configuration**.

Similarly, you initiate an averaging operation by configuring the instrument parameters, including those that control the averaging, and then starting the combined acquisition/averaging sequence. The Averager module resets the accumulation buffers and then acquires the requested number of waveforms, each preceded by a front-panel trigger signal, without any software intervention. The **AcqrsD1_acquireEx** function allows an AP100/AP200 Averager to acquire additional data without resetting the accumulation.

Until the operation is terminated, your application is free to execute other tasks. There are several methods of detecting when the acquisition/averaging operation has ended. Finally, you read the averaged waveform with the function **AcqrsD1_readData** as described below.

If you want to acquire several (averaged) waveforms under the same conditions, there is no need to call the **AcqrsXX_config...** functions again. It is sufficient to execute a loop over the "start, wait, read" functions. In principle a subsequent start will happen considerably faster than the first one that was required to load the full configuration.

3.10.1. Starting an Acquisition

Use the following line of code for starting an acquisition in a D1-style instrument:

```
AcgrsD1_acquire(instrID); // start the acquisition
```

One such command is required for each module in use. However, if several digitizers are combined to a single *MultiInstrument* with AS bus, only a single command is needed for the combined instrument.

3.10.2. Checking if Ready for Trigger

If many modules are being used it may be useful to know when they are all ready to accept a trigger. This can be done by verifying that they are all finished with their pre-trigger phase (PreTrigger = 0) by using the call below to all instruments (or the last instrument started):

```
Acgrs getInstrumentInfo(instrID, "IsPreTriggerRunning", &PreTrigger);
```

3.10.3. Waiting for End of Acquisition

Usually data cannot be read from the instrument until the acquisition is terminated. The application may wait for an acquisition to end either by *polling* or by *waiting for interrupt*.

(A) Simple Polling: use the following code fragment for polling the interrupt status:

```
ViBoolean done = 0;
long timeoutCounter = 100000;
while ((!done) && (--timeoutCounter > 0))
    AcqrsD1_acqDone(instrID, &done); // poll for status or
if (timeoutCounter <= 0) // timeout, stop acquisition
    STOP ACQUISITION</pre>
```



NOTE: The code above has the disadvantage of wasting CPU time while checking the instrument status during the entire acquisition period. In addition, the timeout counter value should be set according to the expected acquisition time, but the loop time depends on the CPU speed.

Programmer's Guide Page 29 of 64

(B) More Efficient Polling: use this code fragment to release the polling thread for short periods:

This code puts the polling thread to sleep for periods of 1 ms at a time, letting other threads of the application or other applications use the CPU time. Setting the timeout counter to 100 means that a total timeout period of 100 ms is expected.



NOTE: This method still has some drawbacks:

- depending on the operating system, the 'Sleep' method often has a granularity of 10 ms or more, rounding any smaller number up to this minimum value
- the response time of the application to the end of acquisition is 50% of the sleep time, on average. With a granularity of 10 ms, the mean latency is therefore 5 ms. Thus, no more than 200 waveforms per second could be acquired, because the application wastes time waiting for the acquisition to terminate.

(C) Waiting for Interrupt:

```
ViStatus status;
long timeOut = 100; // in ms
status = AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);
if (status == ACQIRIS_ERROR_ACQ_TIMEOUT) // timeout, stop
    STOP ACQUISITION
```

This method combines low CPU usage with very good response time:

The function enables the instrument's 'end-of-acquisition' interrupt and sets up a semaphore that waits for this interrupt. It then releases the thread by 'going to sleep', thus letting other threads of the application or other applications use the CPU time. The function returns as soon as the interrupt occurs or when the timeout expires. Note that the timeout asked for will be clipped to a maximum value of 10 seconds.



We recommend using AcqrsXX_waitForEndOfAcquisition since it is the most efficient method. The interrupt latency is of the order of several μs, and no CPU time is wasted.

3.10.4. Stopping/Forcing a D1-style Acquisition

The previous section shows a case where an ongoing acquisition must be stopped, typically because there is no trigger. Also, in some situations you may want to use the digitizer to generate a system trigger under software control.

If you still would like to have a valid snapshot of the current input signal, you should generate a trigger signal by software, with the function AcqrsD1_forceTrig or AcqrsD1_forceTrigEx. Typically, the acquisition does not stop immediately, since the digitizer may continue acquiring some additional data, depending on the delayTime and the data acquisition time that were initially configured. Thus, the application should again wait for the acquisition to terminate. Forcing a trigger does not make sense for averagers and analyzers and should not be done. AcqrsD1_forceTrigEx allows you to generate a trigger out signal which can be synchronized with the sampling clock if desired.

Use the following code fragment to replace STOP ACQUISITION in the previous section:

Programmer's Guide Page 30 of 64

Note that no timeout should ever occur when waiting for a 'forceTrig' to terminate, provided that the **timeOut** value was made large enough. If a timeout does occur, this would indicate a failure in the digitizer or the entire system.

For users generating triggers under software control it may be desirable to do the data readout in a way that just gives the acquired data points and ignores the correction of the data gotten from the **horPos** measurement of the time from the trigger to the next data sample. This can be done using the **flags** parameter of the **AqReadParameters** structure.

3.10.5. Simultaneous multibuffer Acquisition and Readout (SAR)

The U1071A-FAMILY and the 10-bit-FAMILY allow the internal memory's dual-port structure to be exploited. Data acquisition and read out can be done simultaneously. This requires the use of a slightly different programming model.

Configure:

In addition to the usual configuration parameters, the application must enable the SAR mode by calling **AcqrsD1_configMemoryEx** with 'nbrBanks = 2' (or more, 3 is a good choice).

Start:

AcqrsD1 acquire must be called to start the acquisitions.

If valid triggers are provided, from the hardware or software (**AcqrsD1_forceTrigEx** with 'forceTrigType = 1'), then the desired segments will be filled for the current and all the following available banks.

This continues until all 'banks' are full. If a bank is freed (see below) the process can continue by reusing that bank.

Wait for data:

To know if data is available for read out, the application must poll, using **AcqrsD1_acqDone**, or wait for end of an acquisition, using **AcqrsD1_waitForEndOfAcquisition** as usual. If

If either of the above operations successfully returns, it means that at least one bank has been acquired and is ready to be read.

Read:

When data is available, it can be read using the usual **AcqrsD1_readData** functions. The data remain available for multiple reads.

Free the bank and continue:

After all the desired data has been read the bank can be marked for reuse by calling **AcqrsD1_freeBank**. Further calls to **AcqrsD1_acquire** are not required. It is important to free banks as soon as possible in order to ensure that the subsequent triggers are accepted.

Configuration changes:

If the configuration settings are changed, they will only be loaded at the next acquisition restart. Thus, **AcqrsD1_stopAcquisition** must be called and all desired data should be read out. Then the new conditions can become effective with the *Start* of a new acquisition sequence.

Programmer's Guide Page 31 of 64

3.10.5.1. U1084A Averager and SAR

The U1084A Averager and Digitizer supports a simplified version of the Digitizer SAR mode. To enable it:

In Averager mode: Call **AcqrsD1_configMode** with 'mode = 2' (Averaging mode) and 'flags = 10' (SAR mode).

In Digitizer mode: Call **AcqrsD1_configMode** with 'mode = 0' (Normal mode) and 'flags = 10' (SAR mode).

Control of the Averager in this mode is essentially the same as with SAR mode for Digitizers, except that the number of banks is fixed at two.

3.10.6. Analyzer and Peak^{TDC} Autoswitch mode



If with the **Peak**^{TDC} mode the **TdcHistogramMode** parameter has been used to enable histogramming the desired number of acquisitions will be taken automatically. As usual the acquisition must be initialized with a call to **AcqrsD1_acquire(instrID)**. When the acquisition has terminated the histogram data and the peak or gate data of the last acquisition will be available for readout. The routines **AcqrsD1_acqDone** or **AcqrsD1_waitForEndOfAcquisition** must be used. With the U1084A Peak^{TDC}, histogramming is always enabled and this is the only operating mode available.



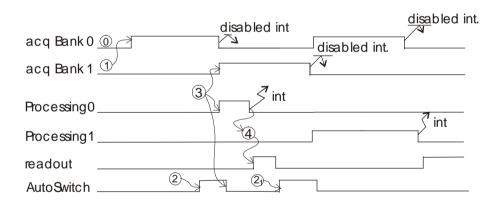
For all other cases the AP Analyzers implement an *SSR Autoswitch* mode that allows the dead time between acquisitions to be reduced to the minimum consistent with the readout of the data. As usual the first acquisition must be initialized with a call to **AcqrsD1_acquire(instrID)**. To allow the second acquisition to start as soon as possible a call to **AcqrsD1_processData** (instrD,processType,1) follows immediately. Thereafter data can be read as soon as the processing is terminated and then the go ahead for the next acquisition can be given as desired.

3.10.6.1. Sequence of actions for SSR mode with event readout

The SSR Autoswitch semaphore is set by the software and cleared by the FPGA.

If the readout process is longer than the acquisition process, the *AutoSwitch* occurs directly after the software raises the *AutoSwitch* semaphore.

At the moment that the ProcessingEnd interrupt occurs, the FPGA has already cleared the AutoSwitch.



| S | Description | Software implementation |
|---|---|--|
| 0 | The software configures the mode, the acquisition parameters, and the readout. | AcqrsD1_configMode(); AcqrsD1_configAvgConfig(); |
| 1 | The software starts the first acquisition. | AcqrsD1_acquire(instrID); |
| 2 | The software sets the <i>AutoSwitch</i> semaphore. To ensure the shortest response time this arming function should be done on the order of 10-20 µs before the expected acquisition end. | AcqrsD1_processData(instrID, processType, 1); processType = 0 is used for SSR & Peak ^{TDC} = 1,2,3, or 4 are for AP101/AP201 or Peak ^{TDC} |
| 3 | When the acquisition has finished, the FPGA | |

Programmer's Guide Page 32 of 64

| | automatically switches the banks, starts a new acquisition, a new processing and clears the <i>AutoSwitch</i> semaphore. | |
|---|--|--|
| 4 | Once the software receives the ProcessingEnd interrupt, it can start the readout. | status = AcqrsD1_waitForEndOfProcessing(instrID, timeout); AcqrsD1_readData(instrID, channel, &readParams, waveformArray, &wfDesc, &segDesc); |
| | Go to 2 to continue. | |

Here is a sample bit of code showing this principle:

To keep the same interface for the AP240 as was the case for the AP101/AP201, the processing step is kept but the "dummy" processing value is used. Actually the software knows which processing is needed from the setup values sent to **AcqrsD1_configAvgConfig**. Thus, the software must wait for the end of processing even if the "processing mode" is set to NO_PROCESSING.

3.10.6.2. What happens when the *AutoSwitch* semaphore is not set

After the "processing" of an acquisition, if the semaphore is not set, the FPGA waits for further instructions. This feature ensures that the software has finished with the old buffer and gives full compatibility with older software implementations. If you make a call to **AcqrsD1_stopAcquisition** you shouldn't try to read the last acquisition's data.

3.10.6.3. Changing acquisition settings while acquiring and reading events

If you want to change any of the acquisition settings you must

o terminate the current acquisition sequence

o configure the instrument for the new values and start the new set of acquisitions

```
// go back to step 0 in the table above
AcqrsD1_configAvgConfig(...); ...
AcqrsD1_acquire(instrID); ...
```

Programmer's Guide Page 33 of 64

3.11. D1-style Data Readout

For the reading of standard waveforms the **AcqrsD1_readData** routine should be used. The following older routines will remain available but will no longer be discussed:

You should use the function **AcqrsD1_readData** for all new programs. The older functions will not give support for new instruments or new functionality. All variables of the **AqReadParameters** structure should be initialized; 0 can be used for the reserved words.

For the readout of the averager data the read function AcqrsD1_readData described in 3.10.1 Reading Digitizer Waveforms with the Universal Read Function should be used with readMode = 2 (ReadModeAvgW). For reading data from analyzers please refer to 3.10.7 Reading SSR Analyzer Waveforms, 3.10.8 Reading AP family PeakTDC Analyzer Data and Histograms, 3.10.9 Reading U1084A PeakTDC Waveforms and Histograms, or 3.10.10 Reading AP101/AP201 Analyzer Waveforms.

3.11.1. Reading Digitizer Waveforms with the Universal Read Function

For the general case, which includes the reading of more complex waveforms, we provide a universal read function **AcqrsD1_readData**.

Control of the read parameters is passed through the input structure **AqReadParameters**. For the description of the output data an array of segment descriptors, **AqSegmentDescriptor**, and a waveform descriptor, **AqDataDescriptor**, are returned. These structures are defined in the header file **AcqirisDataTypes.h**.

The following parameter setting can be used for reading a single waveform segment in 8-bit representation.

```
static long nbrSegments = 1; // readMode = 0 requires this value
const long nbrPoints = 1000;
char dataArray[nbrPoints+32];
AgReadParameters
                     *readPar
                                = new AqReadParameters;
AqDataDescriptor
                                 = new AqDataDescriptor;
                     *dataDesc
AgSegmentDescriptor
                     *seqDesc
                                 = new AgSegmentDescriptor[nbrSegments];
readPar->dataType = 0; // 0 = byte
readPar->readMode = 0; // 0 = standard waveform
readPar->nbrSegments = nbrSegments;
readPar->firstSampleInSeg = 0;
readPar->segmentOffset = 0;
                                 // unused parameter
readPar->firstSegment = 0;
readPar->nbrSamplesInSeg = nbrPoints;
readPar->dataArraySize = sizeof(dataArray);
readPar->seqDescArraySize = sizeof(AqSeqmentDescriptor)*nbrSeqments;
readPar->flags = 0;
readPar->reserved = 0;
readPar->reserved2 = 0.0;
readPar->reserved3 = 0.0;
status = AcgrsD1 readData(instrID, channel, readPar, dataArray ,
                     dataDesc, seqDesc);
```

Comments:

• The segment numbers run from 0 to **nbrSegments**-1.

Programmer's Guide Page 34 of 64

- The value of **segDesc->horPos** is the time interval in seconds between the first data point and the nominal time origin of the trigger delay. It is always in the range [-sampTime, 0]. It is useful for a very precise positioning, to a fraction of the sampling interval, of the waveform. In many applications, it can be ignored. Refer to section 3.12, *HORIZONTAL PARAMETERS IN ACQUIRED WAVEFORMS*, for a detailed explanation of **horPos**. Return values have to be interpreted in the same way as for the other readout functions.
- Refer to the section 3.13, SEQUENCE ACQUISITIONS for detailed explanations on the interpretation of segDesc->timeStampLo/Hi.
- It is important to zero the unused parameters at the end of the readPar structure. An incorrect value of flags can be very confusing.

3.11.2. Reading Sequences of Waveforms

In certain situations, see *APPENDIX A: ESTIMATING DATA TRANSFER TIMES*, it can be more efficient (in time) to read Sequence Waveforms with readMode = 1 (ReadModeSeqW). This mode transfers all of the data from the digitizer to the local memory in a single DMA as opposed to calling **AcqrsD1_readData** many times thus using a transfer per segment. The price to be paid is a higher memory requirement. It can also be used to transfer blocks of segments in the case of very large memories.

For dataType = 0 or 1, the amount of memory needed (in bytes) is

```
arraySize \ge (nbrSamplesNom + currentSegmentPad) * (nbrSegments+1) * (dataType + 1)
```

and, for users with segmentOffset > nbrSamplesInSeg,

```
arraySize \ge segmentOffset * (nbrSegments+1) * (dataType + 1), with segmentOffset \ge, nbrSamplesInSeg, where
```

• the *currentSegmentPad* depends on the acquisition configuration and can be determined using the following call,

• the *nbrSamplesNom* is the nominal number of samples to record and may be different than what was asked for! It can be determined using the following call,

```
AcqrsD1_getMemory(instrID, &nbrSamplesNom, &nbrSegments);
```

• the *nbrSegments* can either be taken from the result given just above or it can be freely reduced to a smaller value in the case of a partial read of the data.

You have to make sure that you ask for this information after the acquisition configuration has been established. This is the case after an acquisition has been completed with the new configuration.

Similarly, for dataType = 3 the amount of memory needed (in bytes) is

```
arraySize \geq 8 * segmentOffset * (nbrSegments), with segmentOffset \geq, nbrSamplesInSeg, and arraySize \geq (nbrSamplesNom + currentSegmentPad) * (nbrSegments+1) * (dataTypeADC + 1)
```

where dataTypeADC is 0 for the 8-bit instruments and 1 otherwise.

The following code can be used for reading a waveform sequence in 8 bit representation.

Programmer's Guide Page 35 of 64

```
long nbrSegments = 10;
long nbrPoints = 1000;
char *dataArrayP;
long currentSegmentPad;
long nbrSamplesNom, nbrSegmentsNom;
AgReadParameters
                     *readPar
                                = new AgReadParameters;
AqDataDescriptor
                     *dataDesc = new AgDataDescriptor;
AgSegmentDescriptor
                               = new AgSegmentDescriptor[nbrSegments];
                     *seqDesc
readPar->dataType = 0; // 0 = byte
readPar->readMode = 1; // 1 = sequence waveform
readPar->nbrSegments = nbrSegments;
readPar->firstSampleInSeg = 0;
readPar->segmentOffset = nbrPoints;
readPar->firstSegment = 0;
readPar->nbrSamplesInSeg = nbrPoints;
readPar->flags = 0;
readPar->reserved = 0;
readPar->reserved2 = 0.0;
readPar->reserved3 = 0.0;
status = Acgrs getInstrumentInfo (instrID, "TbSegmentPad",
                     &currentSegmentPad);
// in this case the next call doesn't have any surprises
status = AcgrsD1 getMemory(instrID,
                     &nbrSamplesNom, &nbrSegmentsNom);
readPar->dataArraySize =
               (nbrSamplesNom+currentSegmentPad)*(1+nbrSegments);
// here we show the malloc explicitly
dataArrayP = (char *)malloc(readPar->dataArraySize);
readPar->segDescArraySize = sizeof(AqSegmentDescriptor)*nbrSegments;
status = AcqrsD1_readData(instrID, channel, readPar, dataArrayP,
                     dataDesc, segDesc);
```

Comments:

- The explicit malloc call will normally not be repeated for every acquisition. Obviously, a larger than needed allocation is perfectly acceptable. Also, any space allocated this way ought to be returned to the heap at some point.
- It is possible to allocate dataArray space that is not aligned on a 32-bit boundary. This is not acceptable for some of our modules and the AcqrsD1 readData routine will return an error in such a case.
- Models U0165A, U1071A & U1084A require alignment of the readout buffer in 64-bits when reading sequences of waveforms. The error ACQIRIS_ERROR_BUFFER_NOT_64BITS_ALIGNED will be returned if this is not the case.

3.11.3. Reading Raw Sequences of Waveforms

In the example given above the driver software in the PC does the work of reordering the raw data sent by the digitizer so that it can be conveniently used from the dataArrayP vector. Readout time can be slightly reduced even further by postponing the reorder until later at a less time critical moment. This functionality is offered with the readMode = 11, raw sequence waveform read. The segment descriptor is modified to include the information needed by the user to the post-acquisition reordering; this requires more space. The dataArray storage space requirements are the same as for readMode = 11. A typical piece of code to do the reorder could look as follows:

Programmer's Guide Page 36 of 64

3.11.4. Averaging Waveforms in a Digitizer

The driver includes 4 functions provided to improve performance when averaging waveforms.

The first pair of functions, AcqrsD1_averagedData for any digitizer (and the older AcqrsD1_averagedWform for 8-bit digitizers only), are meant *only* for single channel, single segment operation. They average a predefined number of waveforms, taking care of the acquisition loop internally. The client must supply a working array (dataArray or waveformArray, for internal use) and an accumulation array (sumArray). The accumulation array is reset automatically inside the function at the beginning of each call. When the function returns successfully, the accumulation array contains the sample-by-sample sum of the waveforms. To get the average values, the array elements must be divided by the number of acquisitions nbrAcq. If, for each acquisition, the trigger does not arrive within the requested timeout after the beginning of the acquisition, the function returns with an error code.

The second pair of functions, AcqrsD1_accumulateData for any digitizer (and the older AcqrsD1_accumulateWform for 8-bit digitizers only) can be used for multi-channel operation and can be called for each acquisition the user wants to accumulate. It reads the waveform in the module, and performs a sample-by-sample accumulation in the client array. Here again, the client must supply a working array (dataArray or waveformArray, for internal use) and an accumulation array (sumArray). The client controls the acquisition, and must reset the accumulation array appropriately.

In both cases, the allocation of the memory for the working array (**dataArray** or **waveformArray**) has been left to the client for performance reasons. Its size must be at least the requested number of samples **nbrSamples** + 32, for reasons of data alignment. The content of this working array is not meant to be used by the client.

Please note that in both cases, sub-sample timing information (i.e. horPos, see section 3.12, **Horizontal Parameters in Acquired Waveforms**) is not taken into account.

3.11.5. Reading an Averaged Waveform from an Averager

Averaged waveforms can be read out either in Volts, or as 32-bit accumulated sums. In either case, we recommend the use of the general-purpose read function **AcqrsD1_readData**, rather than the obsolete function **AcqrsD1_readRealWform**.

3.11.5.1. Averaged Waveforms in Volts

You should use the general-purpose function **AcqrsD1_readData**. As long as the *mode* is still set to *averager*, either function automatically divides the accumulated waveform sum by the number of acquired waveforms, and returns the result in Volts. They also return zero into the variables *horPos*, *tStampLo* and *tStampHi*, since they are irrelevant in the context of an averaged waveform.

Use this code fragment for the general-purpose function:

Programmer's Guide Page 37 of 64

```
AgReadParameters
                     readParams; // Read Definitions
AqDataDescriptor
                       wfDesc;
                                    // Returned (common) waveform values
AqSegmentDescriptorAvg segDesc;
                                     // Returned segment values
long channel = 1, nbrSamples = 20000;
double waveformArray[20000];
readParams.dataType = ReadReal64;
                                           // Request Volts
readParams.readMode = ReadModeAvgW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;
                                           // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.segDescArraySize = sizeof(AqSegmentDescriptorAvg);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;
AcqrsD1_readData(instrID, channel, &readParams, waveformArray, &wfDesc,
                     &segDesc);
```

Note: If you call a readout function while the acquisition *mode* is set to *digitizer*, it will return the last acquired single waveform, possibly with some unpredictable results.

Note: The 'raw' sums can be read directly with a different function call (see next section). The relationship between Volts and the raw sum is expressed by the following formula:

```
for Averager mode data
```

```
sum[i] = (volts[i] + offset + FS/2.0) * 256 * nbrWforms / FS
or for Inverted Averager mode data
sum[i] = (FS * 127./256. - volts[i] - offset) * 256 * nbrWforms / FS
```

with the following definitions:

```
sum[i] 32-bit integer sum at position i, unipolar (i.e. 0 or positive)
volts[i] floating point voltage at position i, as returned by the code fragments above
offset offset in Volts, as set with AcqrsD1_configVertical
FS full scale range in Volts, as set with AcqrsD1_configVertical
nbrWforms number of summed waveforms
```

The value of 'nbrWforms' must be known, i.e. if the averaging process was interrupted before reaching the requested number of waveforms, the formula above yields wrong results. As a check that the correct value of 'nbrWforms' was used, the value of 'sum[i]', before conversion to an integer, must already be very close to an integer.

Use this code fragment for the 'legacy' function:

```
long channel = 1, segmentNumber = 0, nbrSamples = 20000;
long returnedSamples, tStampLo, tStampHi;
double waveformArray[20000], horPos, sampTime;
AcqrsD1_readRealWform(instrID, channel, segmentNumber, 0, waveformArray, &returnedSamples, &horPos, &sampTime, &tStampLo, &tStampHi);
```

Programmer's Guide Page 38 of 64

3.11.5.2. Averaged Waveforms as 32-bit Sums

You must use the general-purpose function AcqrsD1_readData.

Use this code fragment:

```
AgReadParameters
                  readParams;// Read Definitions
AqDataDescriptor
                   wfDesc;
                            // Returned (common) waveform values
AqSegmentDescriptorAvg segDesc;
                                   // Returned segment values
long channel = 1, nbrSamples = 20000;
long waveformArray[20000];
readParams.dataType = ReadInt32;
                                           // Request 32-bit sums
readParams.readMode = ReadModeAvgW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;
                                            // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.segDescArraySize = sizeof(segDesc);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;
AcgrsD1_readData(instrID, channel, &readParams, waveformArray, &wfDesc,
                     &segDesc);
```

The returned data values in *waveformArray* are unipolar, i.e. the raw ADC values are coded as values between 0 and 255, so that the summed data values may run between 0 and 255*N (N= number of waveforms in the sum).

3.11.6. Reading a RT Add/Subtract Averaged Waveform from an Averager

This case is significantly different than the normal averager case described above.

The 'raw' sums now have to be considered as signed values. The relationship between Volts and the raw sum is expressed by the following formula:

```
sum[i] = volts[i] * 256 * nbrWforms / FS
```

with the same definitions as before. However, the user has to understand if the final result corresponds to the desired signal or just half of it.

3.11.7. Reading SSR Analyzer Waveforms

3.11.7.1. SSR Mode Readout Data Format

In all cases data values are returned in the range [-128, +127]. The relationship between Volts and the raw data is expressed by the following formula:

```
data[i] = (volts[i] + offset) * 256 / FS
```

with the following definitions:

```
data[i] 8-bit signed ADC value at position i
volts[i] floating point voltage at position i, as returned by the code fragments above
offset offset in Volts, as set with AcqrsD1_configVertical
FS full scale range in Volts, as set with AcqrsD1_configVertical
```

In all cases you must readout the entire acquisition. You cannot ask for a reduced number of segments.

Programmer's Guide Page 39 of 64

3.11.7.2. Raw data

The complete data should only be read out using the gated data mode described below. An appropriate User Gate can be defined to access all of the data.

3.11.7.3. Gated data

Data can be read for both user and threshold gate operation using readMode = 7 (ReadModeSSRW).

The waveform descriptor structure contains the value actualDataSize giving the total number of data bytes read. A time stamp block, measuring the trigger time, will mark the beginning of each segment. Segment time stamps are mixed in with the data and not available through the usual segDesc array. The entire time stamp is a 56-bit integer counting in units of 100 ns.

Here is the Time Stamp format:

Marker block

| 3124 (8 bits) | 230 (24 bits) | | |
|------------------------|----------------|--|--|
| Flag = 00000100 (0x04) | Time Stamp MSB | | |
| Time Stamp LSB | | | |

The time stamp may be followed by a variable number of blocks of data with the following format:

Gate block

| 3124 (8 bits) | 230 (24 bits) | | | | | | |
|-------------------------|-------------------------------|---|-------|--|--|--|--|
| Flag = 00000000 | Gate position from the orig | Gate position from the origin of the acquisition (not the segment!) | | | | | |
| 310 | | | | | | | |
| Gate length (number of | of Data bytes, always a multi | iple of 4) | | | | | |
| 3124 | 2316 | 158 | 70 | | | | |
| Data3 Data2 Data1 Data0 | | | | | | | |
| | | | Data4 | | | | |

When reading such data you should carefully check that you terminate correctly and do not read beyond the end of the transmitted data nor generate unphysical time coordinates for the data.

3.11.7.4. Waveform storage requirements

When using the routine AcqrsD1_readData you must allocate waveform storage and inform the driver about the number of bytes available.

Raw data readout requires exactly the number of bytes corresponding to the number of segments times the number of data points per segment.

User gate readout for each segment requires 8 bytes for the time stamp and an overhead of 8 bytes for each gate. This must be added to the total number of samples in all of the gates to get the required length for each segment and multiplied by the number of segments to get the waveform array length.

For threshold gate readout the program should allocate the space needed for the worst case. This means 8 bytes for the time stamp, 8 bytes for a gate block header and space for the total number of samples/segment. This must be multiplied by the number of segments to get the waveform array length. It should be noted that the FPGA will generate a single gate block if there are less than 32 data points "below" threshold between two desired data points.

3.11.8. Reading AP family Peak^{TDC} Analyzer Data and Histograms

3.11.8.1. Reading the gated data

The gated data of the current event can be read out as described in 3.10.7.3 **Gated data**. This is the only output format that gives access to the segment time stamps.

3.11.8.2. Reading the data in the peak regions

The data of the peak regions in the current event can be read out using readMode = 10 (ReadModePeakPic).

The waveform descriptor structure contains the value actualDataSize giving the total number of data bytes read. There will be a variable number of blocks of data with the following format:

Programmer's Guide Page 40 of 64

Peak region block 8 points

| 3124 (8 bits) | 2316 (8 bits) | 158 (8 bits) | 70 (8 bits) | | | | |
|------------------------|----------------|----------------|----------------|--|--|--|--|
| Flag 00010010 | 0x00 | Valid Left | Valid Right | | | | |
| 310 (32 bits) | | | | | | | |
| Peak Position (= tmax) | | | | | | | |
| 3124 (8 bits) | 2316 (8 bits) | 158 (8 bits) | 70 (8 bits) | | | | |
| Sample(tmax) | Sample(tmax-1) | Sample(tmax-2) | Sample(tmax-3) | | | | |
| Sample(tmax+4) | Sample(tmax+3) | Sample(tmax+2) | Sample(tmax+1) | | | | |

Peak region block 16 points

| T tuni Tegron oroth | | | |
|------------------------|----------------|----------------|----------------|
| 3124 (8 bits) | 2316 (8 bits) | 158 (8 bits) | 70 (8 bits) |
| Flag 00010001 | 0x00 | Valid Left | Valid Right |
| 310 (32 bits) | | | |
| Peak Position (= tmax) | _ | | |
| 3124 (8 bits) | 2316 (8 bits) | 158 (8 bits) | 70 (8 bits) |
| Sample(tmax-4) | Sample(tmax-5) | Sample(tmax-6) | Sample(tmax-7) |
| Sample(tmax) | Sample(tmax-1) | Sample(tmax-2) | Sample(tmax-3) |
| Sample(tmax+4) | Sample(tmax+3) | Sample(tmax+2) | Sample(tmax+1) |
| Sample(tmax+8) | Sample(tmax+7) | Sample(tmax+6) | Sample(tmax+5) |

The peak position is counted relative to the beginning of the first segment of the acquisition. Thus the position also gives the segment number of the peak.

The sample data value is the raw data value.

Valid Left and Valid Right give the number of valid data points < tmax and > tmax respectively. Sample(tmax-validLeft) and Sample(tmax+validRight) are the first and last valid points, respectively.

If you need to know the segment time stamps you can read the gated data. If the user or threshold gate parameters are set appropriately the amount of unwanted data can be minimized.

3.11.8.3. Reading the peaks

The results for all of the peaks in the current event can be read out using readMode = 4 (ReadModePeak).

The waveform descriptor structure contains the value actualDataSize giving the total number of data bytes read. There may be a variable number of blocks of data with the following format:

Peak block

| 3124 (8 bits) | 23 | 194 (16 bits) | 30 (4 bits) |
|----------------------|----------------|---|--|
| Flag 00010000 (0x10) | Unused | Peak amplitude with ADC resolution | Interpolated fractional part of amplitude (1/16 LSB) |
| 294 (26 bits) | | | 30 (4 bits) |
| Peak Position wi | th sample rate | Interpolated fractional part of position (1/16 sample interval) | |

The peak position is counted relative to the beginning of the first segment of the acquisition. Thus the position also gives the segment number of the peak.

The peak amplitude is the value acquired after baseline subtraction.

When reading such data you should carefully check that you terminate correctly and do not read beyond the end of the transmitted data nor generate unphysical time coordinates for the data.

If you need to know the segment time stamps you can read the gated data. If the user or threshold gate parameters are set appropriately the amount of unwanted data can be minimized.

Programmer's Guide Page 41 of 64

3.11.8.4. Reading the histogram

The accumulated histogram can be read out using readMode = 9 (ReadModeHistogram). The dataType and dataArraySize must be selected to correspond to the size of the histogram and its bins.

The waveform descriptor structure contains the value actualDataSize giving the total number of data bytes read. The individual histogram bins will be able to contain accumulated sums of either 2**32, as ViUInt32, or 2**16, as ViUInt16.

| Histogram | 17111 | 15 | 1/. | 11115 | with |
|-----------|-------|----|-----|-------|--------|
| 111000 | | | | 0100 | ****** |

| 310 | | |
|------|--|--|
| Bin0 | | |
| | | |
| | | |

Histogram bin is 16 bits wide

| 3116 | 150 |
|-------|-------|
| Bin 1 | Bin 0 |
| | |

3.11.9. Reading U1084A Peak^{TDC} Waveforms and Histograms

3.11.9.1. Reading the histogram

Use 'readMode' = 9 (ReadModeHistogram), 'dataType' = 2 (ReadInt32) to read out the accumulated histogram. The nbrSamples must be selected to correspond to the size of the histogram and its bins. Note that the value for 'nbrSamples' should correspond to the number of bins in the histogram to read, which may be different from the configured number of samples in the acquisition if interpolation is enabled. For example, if you have configured '**NbrSamples'** to 1024 and '**TdcHistogramHorzRes'** to 3 with **AcqrsD1_configAvgConfig**, there will be $2^3 = 8$ bins per sample, for a total of $1024 * 2^3 = 8192$ bins available for readout.

If vertical interpolation is used, the values of the bins should be interpreted as fixed point fractional values. This means that, if '**TdcHistogramVertRes'** is set to N, the actual amplitude of the bin is the raw unsigned integer value divided by 2^N .

Because the data array and transfer size may have to be realigned, the data array should be at least 16 bytes (4 samples) longer than the histogram data to read. To find the first valid sample in the data array, the data descriptor's 'indexFirstPoint' field must be used, which indicates the offset in samples of the first valid sample from the beginning of the data array.

3.11.9.2. Reading the last acquired waveform

To facilitate the visualization and interpretation of the histogram data, the U1084A permits to read the last raw waveform which contributed to the histogram, as 8 bit samples. This is done using readMode = 0 (ReadModeStdW) with dataType = 0 (ReadInt8). The procedure is the same as the one described in 3.10.1 Reading Digitizer Waveforms with the Universal Read Function.

3.11.10. Reading AP101/AP201 Analyzer Waveforms

3.11.10.1. Reading a Buffered Waveform Sequence

This section concerns AP101/AP201 Analyzers ONLY. In *normal* mode, you may read the acquired waveform(s) in the same way as with any other digitizer, as described in the section 3.11, **D1-style Data Readout**, in the Programmer's Guide.

In *buffered* mode, you must use the functions **AcqrsD1_readData** to read out the accumulated waveform sequence, as a single data record. E.g. if you configured nbrSamples = 5000 and nbrSegments = 800, you should specify segmentNumber = 0 and nbrSamples = 4'000'000.

Before reading the buffered data, you *must* switch to the other memory bank. Typically, you also would start a new acquisition before readout, but it is not required. This is done automatically in the *SSR* mode as a consequence of the call to **AcqrsD1_processData** with a non-zero flag value. It can also be done with a call to AcqrsD1_configMode

The read-function returns zero into the dataDesc variables *horPos*, *tStampLo* and *tStampHi*, since they are unavailable in the context of a buffered waveform sequence.

Programmer's Guide Page 42 of 64

With the function AcqrsD1_readData, use this code fragment:

```
AqReadParameters
                     readParams; // Read Definitions
                     dataDesc; // Returned waveform values
AqDataDescriptor
                                     // Returned segment values
AgSegmentDescriptor
                     segDesc;
long channel = 1, nbrSamples = 4000000;
char waveformArray[4000000];
readParams.dataType = ReadInt8;
readParams.readMode = ReadModeStdW;
readParams.nbrSeqments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;
                                           // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.seqDescArraySize = sizeof(seqDesc);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;
memoryBank = (memoryBank+1)&0x1;
                                     // switch to other bank
AcqrsD1_configMode(instrID, 3, 0, memoryBank);
AcgrsD1 acguire(instrID);
                                 //essential!!
AcgrsD1_readData(instrID, channel, &readParams, waveformArray, &dataDesc,
                     &segDesc);
```

The returned data array contains the acquired waveforms as a contiguous array. E.g. if you configured nbrSamples = 5000, the data points 'waveformArray[0...4999]' correspond to the first waveform, the data points 'waveformArray[5000...9999]' correspond to the second waveform etc.

3.11.10.2. Reading Gated Waveforms

For reading gated waveforms the actual desired gates should be set with the setup function **AcqrsD1_configSetupArray**. This function should be called before **AcqrsD1_acquire** is invoked to acquire any data that needs to be read using these gates. To read back the gate values, **AcqrsD1_getSetupArray** has to be used. An example for the two routines is shown in the following code:

The condition GateLength >= 4 is required. Both GateLength and GatePos must be multiples of 4.

You can read the gate parameters back, with this code:

Programmer's Guide Page 43 of 64

Make sure to use a pointer to the last argument, since it returns the number of gates. *lastGate* cannot exceed the number of gates being written.

To read the gated waveforms, use the function AcqrsD1_readData.

```
readParams; // Read Definitions
AgReadParameters
                     dataDesc; // Returned waveform values
AgDataDescriptor
AgSegmentDescriptor
                     segDesc;
                                    // Returned segment values
long channel = 1, nbrSamples = 20000;
char waveformArray[20000];
readParams.dataType = ReadInt8;
readParams.readMode = ReadModeGateW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;
                                           // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
readParams.dataArraySize = sizeof(waveformArray);
readParams.segDescArraySize = sizeof(segDesc);
readParams.flags = 0;
readParams.reserved = 0;
readParams.reserved2 = 0.0;
readParams.reserved3 = 0.0;
AcqrsD1_readData(instrID, channel, &readParams, waveformArray, &dataDesc,
                     &segDesc);
```

The returned data array contains the acquired waveforms as a contiguous array of **dataDesc- returnedSamples** bytes.

Note: Make sure that the *waveformArray* is large enough to hold the sum of all *GateLength's* times the *nbrSegments*. As a rule, the *waveformArray* has to have as a minimum size the sum of all the gate sizes times *nbrSegments* ($waveformArray > (\sum GateLength)* nbrSegments$).

Note: If, for each gate, the sum (GatePos + GateLength) exceeds the nbrSamplesInSeg, GatePos is reduced to satisfy (GatePos + GateLength) <= nbrSamplesInSeg. If this is not sufficient, GateLength is shortened to satisfy that condition.

3.11.10.3. Data Processing before Readout

In *buffered* mode, the AP101 offers the capability of processing the acquired data before readout. This operation must be explicitly requested by the application, after the data acquisition has terminated.

Depending on the processing algorithms used, you may have to prepare the data processing by setting the appropriate parameters with the function **AcqrsD1_configSetupArray**.

In the 'peak-detect' mode, the AP101 will return for each gate exactly 2 peaks, first the positive and then the negative one (some of which might be marked as 'invalid' if no valid peak exists!). Thus, you should define the gates in the same way as described in the previous section.

A typical acquisition/processing/readout sequence in autoswitch buffered mode would be:

Programmer's Guide Page 44 of 64

- 1. Configure the APXXX for appropriate channel, timebase, trigger, and gate parameters.
- 2. Start the first acquisition.
- 3. Give the order to switch banks and start the next acquisition and data processing on the current acquisition as soon as possible. The processing can overlap with the data acquisition since it automatically deals with the memory bank that is not selected for acquisition. If you need to read the original data choose the "no processing" option.
- 4. Wait for the processing to be terminated, read the processed result. Note that the processing will not destroy the originally acquired data.
- 5. You may now do any additional processing in your computer. However, you cannot read the original data at this point without perturbing the acquisition.
- 6. Now that you have finished all work with the current data you can loop to (3) above.

A typical acquisition/processing/readout sequence in explicit buffered mode would be:

- 7. Configure the APX01 for appropriate channel, timebase, trigger, and gate parameters.
- 8. Start the first acquisition.
- 9. Wait for the first acquisition to terminate.
- 10. Switch the memory bank and start a new acquisition in the second bank. Note that you must start the new acquisition to make the memory bank switch happen.
- 11. Start data processing. This can overlap with the data acquisition since it automatically deals with the memory bank that is not selected for acquisition.
- 12. After processing has terminated, read the processed result. Note that the processing will not destroy the originally acquired data.
- 13. You may now do any additional processing in your computer. However, you cannot read the original data at this point without perturbing the acquisition.
- 14. Wait for the new acquisition to terminate.
- 15. If you need to read the original data, e.g. for diagnostics on the processing algorithm, you may now do so.
- 16. Loop to (4.) above.

You need to implement a method to interrupt the infinite loop whenever required. However, you should make sure that you leave the acquisition in a well-determined state for future operation.

The explicit acquisition/processing/readout sequence described above is shown in the following code:

```
// Read Definitions
AgReadParameters
                        readParams;
AqDataDescriptor
                                      // Returned waveform values
                     dataDesc;
long channel = 1, segmentNumber = 0;
                                 // nbrGates is defined by user
long nbrPeaks = 2 * nbrGates;
long waveformArray[2 * nbrPeaks];
long memoryBank = 0, timeout = 5000; // timeout = 5 seconds
// Insert whatever is required for Vertical
        and Trigger configuration
//
AcqrsD1_configMode(instrID, 3, 0, memoryBank);
AcgrsD1 acquire(instrID);
                                 // Acquire into bank 0
AcqrsD1_waitForEndOfAcquisition(instrID, timeout);
// At this point, you should check the return value!
```

Programmer's Guide Page 45 of 64

```
bool finished = false;
while(!finished)
    memoryBank = (memoryBank + 1)&0x1;// switch to other bank
    AcgrsD1_configMode(instrID, 3, 0, memoryBank);
    AcgrsD1 acquire(instrID); // start new acquisition
    AcgrsD1 processData(instrID, 0, 0);// start processing
    AcgrsD1 waitForEndOfProcessing(instrID, timeout);
    // At this point, you should check the return value!
    readParams.dataType = ReadReal64;
    readParams.readMode = ReadModePeak;
    readParams.nbrSegments = 1;
    readParams.firstSampleInSeg = 0;
    readParams.segmentOffset = 0;
    readParams.firstSegment = 0;
                                           // Read first segment
    readParams.nbrSamplesInSeg = 2* nbrPeaks; // pos and neg peak
    readParams.dataArraySize = 2 * sizeof(double) * nbrPeaks;
    readParams.segDescArraySize = 0;
    readParams.flags = 0;
    readParams.reserved = 0;
    readParams.reserved2 = 0.0;
    readParams.reserved3 = 0.0;
    AcgrsD1 readData(instrID, channel, &readParams, waveformArray,
                     &dataDesc, NULL);
    //... analyze and store data
    AcgrsD1_waitForEndOfAcquisition(instrID, timeout);
    //...read original data if desired
    //...check on loop termination conditions and set `finished'
```

The returned **waveformArray** contains exactly 2*nbrGates peaks, each of which is described by 2 double precision floating-point values. The first pair of doubles contains the positive peak position, within the gate (in units of samples), and the amplitude (in codes). The second pair of doubles contains the negative peak position and its amplitude. The peak amplitude is a signed number in the range [-128.0, +127.0] with the two extreme values indicating underflow/overflow conditions. The peak position is normally positive. Negative values are used for warnings. In particular, the value -DBL_MAX indicates that no peak was found. The #define of DBL_MAX can be found in the float.h include file. Each peak contains the following two words:

```
0-63
Peak Position (in samples from start of gate)
```

```
0 – 63
Peak Amplitude (in ADC units)
```

Use the following code to obtain the peak positions for the i'th gate:

```
double *positivePeakPos = &waveformArray[0+8*i];
double *negativePeakPos = &waveformArray[4+8*i];
```

Programmer's Guide Page 46 of 64

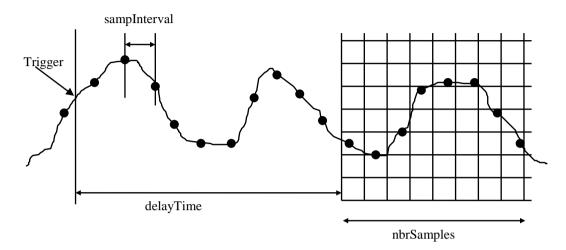
3.12. Trigger Delay and Horizontal Waveform Position

When using a digitizer the user has 3 instrument setup variables with which to position the acquired waveform in time:

• sampInterval: the sampling interval (inverse of the sampling frequency)

• **nbrSamples**: the number of samples to acquire

• **delayTime**: the nominal trigger delay



By convention, the nominal trigger delay is taken relative to the beginning of the trace, i.e. relative to the left edge of a real or virtual display grid. It can be interpreted as the time from the trigger to the start of waveform recording. If this number is positive, recording starts *after* the trigger (post-trigger acquisition). If it is negative, recording starts *before* the trigger (pre-trigger acquisition). In reality, the acquisition always runs before any trigger occurs, and **delayTime** controls the time between the trigger and the stopping of the acquisition:

| delayTime | Time until Acquisition Stop | Comments | | | |
|--------------------------------|--|--|--|--|--|
| - sampInterval * nbrSamples | 0 | Trigger point is at the right edge of grid, i.e. at the end of the nominal waveform (100 % pre-trigger) | | | |
| < 0 | sampInterval * nbrSamples + delayTime | Trigger point is at the desired point within the grid | | | |
| 0 | sampInterval * nbrSamples | Trigger point is at the left edge of grid, i.e. at the beginning of the nominal waveform (0 % pre-trigger) | | | |
| >0 | sampInterval * nbrSamples + delayTime | Trigger point is to the left of the grid, i.e. before the beginning of the nominal waveform (post-trigger) | | | |

Note that **delayTime** is not allowed to become more negative than - **sampInterval** * **nbrSamples**, because it is impossible to stop the acquisition *before* the trigger occurs.

3.13. Horizontal Parameters in Acquired Waveforms

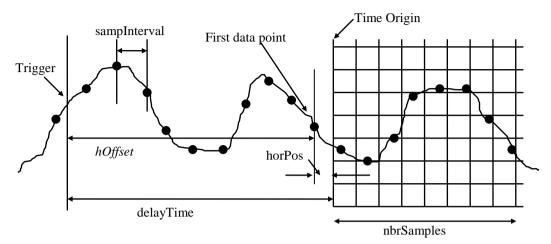
Triggers usually occur asynchronously with respect to the sampling clock. Thus, between similar events, the time from the trigger to the next sampling clock varies randomly in the range [0 ... sampInterval].

The true time reference for any waveform is the trigger point, *not* the sampling times, because the trigger is attached to a given feature of the waveform (e.g. a transition at a predetermined level). For highly stable displays, it is important to know the time between the trigger and the next sampling clock to within a fraction of the sampling interval, and to place the displayed data points in such a way that the trigger point stays at a constant position. This is particularly important for persistence displays or highly zoomed random-interleaved displays, as generated from overlaid segments, where a single waveform (or waveform segment) contributes only a few data points to the display.

Programmer's Guide Page 47 of 64

Acqiris digitizers feature a Trigger Time Interpolator (TTI), which measures the time between the trigger event and the next sampling clock to a fraction of the sampling interval. It permits very precise positioning of the acquired trace in highly zoomed displays, particularly when multiple acquisitions of the same signal are used. In many other applications, this value can be ignored.

The following drawing completes the description of a 'real-life' waveform:



- The value of **delayTime** positions exactly the left edge of the display (or the exact nominal beginning of the waveform) with respect to the "stable" trigger time, which is the real reference point. We define the time 'trigger time + delayTime' as the time origin for the waveform, which is equivalent to saying that the trigger always occurs exactly at the time **-delayTime**.
- The first data point of the waveform is defined as the *last* acquired data point *before* the time origin. It is indexed with i = 0 in the formula below.



NOTE: It is important to realize that if a single segment is read (e.g. with AcqrsD1_readData) the first data point will be dataArray[readPar.indexFirstPoint] and that this is not necessarily the first point given.

- The exact position of the first data point with respect to the time origin is a negative number **horPos**. It is by definition in the range [-sampInterval, 0].
- The time between the trigger and the first data point **hOffset** need not be recorded since it can always be computed as **delayTime** + **horPos**. Note: **delayTime** + **horPos** < **delayTime** by definition.
- In order to obtain a very stable image, even in a highly zoomed display, the user only needs to position the acquired data points with the aid of **horPos**, by using the following formula for the x-position of point i with respect to the left edge of the display:

x[i] = horPos + i * sampInterval

3.14. Sequence Acquisitions

For digitizers in Sequence acquisition mode, multiple waveforms are acquired autonomously, with a single start command **AcqrsD1_acquire**. Whenever a trigger is received, the current acquisition segment is normally terminated. The digitizer then automatically initializes another acquisition into the next memory segment, until all requested segments are filled.

3.15. Time stamps

The U1071A, 10-bit-FAMILY and U1084A digitizers implement a time stamp to measure the time of the trigger for each acquisition segment. These time stamps can be used to calculate the time between any two triggers for any pair of triggers over multiple acquisitions.

The other, "older" Acqiris digitizers feature a 'time stamp' in order to measure the time between the triggers of consecutive segments in the same acquisition. In fact, the time stamp counter is started when the Sequence acquisition is started, and keeps counting during the entire sequence. The difference between the time stamps of any pair of (not necessarily adjacent) segments is the time between their respective triggers.

The resolution of the timestamp is identical to that of the trigger time interpolator for that model. For most models the time stamp value is returned as a 64-bit integer, in units of picoseconds, but there are certain exceptions, for example the timestamp resolution U1084A-DGS in Zero-suppression mode will depend on the model options and the

Programmer's Guide Page 48 of 64

sampling rate being used (see the product brochure/datasheet to confirm the resolution). The waveform readout function **AcqrsD1_readData** returns the time stamp value as 2 32-bit values. In order to do time differences, you should transform them into a 64-bit integer:

• In Visual C/C++, use the 64-integer **int64** as follows:

```
__int64 timeStamp = timeStampHi;
timeStamp = timeStamp<<32 + (unsigned long)timeStampLo;</pre>
```

Arithmetic operations between such integers can be done as with shorter integers.

You also can convert a time stamp difference to an extended floating point number, and do arithmetic operations as with other variables:

```
double deltaTime = (double)(timeStamp - previousStamp);
```

Arithmetic operations between such decimal variants can be done as with other integer variables. The manipulation of **tStampLo** is somewhat complicated because this variable is a signed 32-bit integer, but must be added as an unsigned integer to the (shifted) **tStampHi**.

 In LabVIEW, convert the time stamp to an extended floating point number, and do arithmetic operations as with other variables.

3.16. External Clock and Reference

The *external reference mode* replaces the internal 10 MHz reference clock with an external one at the same or a similar frequency, from which the actual sampling clock is derived.

In the *external clock mode*, a waveform is sampled according to a clock derived from transitions of the external clock signal through the user-defined threshold. We distinguish between *continuous* external clock operation and *start/stop* external clock operation.

All external clock/reference modes are configured with the function AcqrsD1_configExtClock.

The external clock/reference signal should have a peak-peak amplitude of at least

- 0.5 V for the DC135/DC140/DC211A/DC241A/DC271A/DC271AR and 10-bit-FAMILY,
- 1 V for the other DC271-FAMILY digitizers, the U1084A-FAMILY, U1071A-FAMILY, 12-bit-FAMILY, the AC/SC Analyzers, and the AP Averagers and Analyzers,
- 2 V peak to peak for all other models.

. The **inputThreshold** value should be set to the center of the signal.

3.16.1. External Reference

This *external reference* mode (**clockType** = 2) simply replaces the internal 10 MHz reference clock with an external one at the same or a similar frequency. Alternatively, for the DC135/DC140/DC211/DC211A/DC241/DC241A/DC271A/DC271AR, the AC/SC and the 10-bit-FAMILY, the PXI 10 MHz System Clock can be used as the reference.

If you need a more precise timebase, or want to ensure that the timebases of several modules are at exactly the same frequency, you should use $\mathbf{clockType} = 2$ in the function, and apply an external 10 MHz signal. All other settings of the digitizer are exactly the same as with an internal \mathbf{clock} .

If you need to sample at a rate that deviates from the nominal values, you may apply an external reference signal with a constant frequency in the range of

- [9.97, 10.03] MHz for the U1084A-FAMILY, U1071A-FAMILY and the 10-bit-FAMILY
- [9.0, 11.0] MHz for the 12-bit-FAMILY
- [9.0, 10.2] MHz for all other modules.

You need to correct for the reference frequency difference in your application since the digitizer and the driver do not take the deviations into account.



NOTE: A square wave with better than 5 ns risetime should be used. This is needed to avoid false or multiple transitions on a slower risetime signal. Alternatively, a > 2 V amplitude signal could be used.

Programmer's Guide Page 49 of 64



NOTE: When using this capability please make sure that the module is correctly synchronized on the signal. This might require adjusting the threshold. If this is not the case the data will be useless and calibration can fail in rather obscure ways.

3.16.2. External Clock (Continuous)

The *continuous external clock* mode (**clockType** = 1) permits the application to the digitizer of a continuous, constant frequency, external clock in order to sample at an arbitrary frequency. This mode uses normal triggering, from the input signal or through the external trigger input

We need to distinguish between *first generation* digitizers (models DP105, DP106, DP110, DP111, DP210, DP211, DP212, DC110, DC240, DC265, DC270), *second generation* digitizers of the DC271-FAMILY (models DC135, DC140, DC211, DC211A, DC241, DC241A, DC271, DC271A, DC271AR, DP214, DP235, DP240), the AP240/AP235 signal analyzer platforms, the AC210/AC240/SC210/SC240 analyzers, and the 12-bit-FAMILY (DC440, DC438, DC436, DP310, DP308, DP306), and *third generation* digitizers (10-bit-FAMILY, U1071A-FAMILY) since their behavior in this mode is quite different. *AP200/AP201/AP100/AP101 Averagers and Analyzers* are considered to be *first generation* modules.

The horizontal control parameters **sampInterval** and **delayTime** as defined by **AcqrsD1_configHorizontal** are ignored. You need to give the driver the current input frequency and the requested sampling frequency with the variables **inputFrequency** and **sampFrequency** of the function **AcqrsD1_configExtClock**.

The input frequency inputFrequency must be between

- 10 MHz and 500 MHz in the first generation models,
- 20 MHz and 2000 MHz for the second generation DC271-FAMILY, U1071A-FAMILY, AC/SC, or AP240/AP235,
- 100 MHz and the value for the maximum allowed frequency for the 12-bit-FAMILY.

Note that for normal operation of Revision B DC4xx and Revision A DP3xx 12-bit-FAMILY digitizers the sFmax for each converter should be kept between 50 and 110 MHz. Furthermore, the sFmax for these older DC438 and DP308 units is the same as the sampling frequency.

The acceptable values for **sampFrequency** are dividers (sFmax/n) of the maximum allowed sampling frequency, sFmax, where n=1,2,4,8,20,40,80,200,... The sFmax depends on the model and on the number of combined channels **nbrConvertersPerChannel** of **AcqrsD1_configChannelCombination**:

| Model | Channel | | | |
|---------------------------|--------------------------|--------------------------------|--------------------------------|---------------------------|
| | Frequency range (MHz) | 1 | 2 | 4 |
| DC122 | 1000 - 2000 | 2 × inputFrequency | | |
| DC135 | 20 - 2000 | ¹⁄₄ × inputFrequency | $\frac{1}{2}$ × inputFrequency | |
| DC140 | 20 - 2000 | ½ × inputFrequency | $1 \times input$ Frequency | |
| DC152 | 200 - 2000 | $1 \times inputFrequency$ | $2 \times input Frequency \\$ | |
| DC211/DC211A | 20 - 2000 | $2 \times inputFrequency \\$ | | |
| DC222 | 1000 - 2000 | $4 \times input$ Frequency | | |
| DC241/DC241A | 20 - 2000 | $1 \times inputFrequency$ | $2 \times input$ Frequency | |
| DC252 | 1000 - 2000 | $2 \times inputFrequency$ | $4 \times input$ Frequency | |
| DC271/DC271A / DC271AR | 20 – 2000 | ½ × inputFrequency | 1 × inputFrequency | $2 \times inputFrequency$ |
| DC282 | 200 - 2000 | $1 \times inputFrequency$ | $2 \times input Frequency \\$ | $4 \times inputFrequency$ |
| DC436 Rev B | 100 - 200 | ½ × inputFrequency | | |
| DC438 | 100 - 400 | $\frac{1}{2}$ × inputFrequency | | |
| DC440 | 100 - 420 | $1 \times inputFrequency$ | | |
| DC440/438 Rev B | 100 - 400 | $1 \times inputFrequency$ | | |
| DP1400 | 20 - 2000 | ½ × inputFrequency | $1 \times input$ Frequency | |
| DP214 | 20 - 2000 | $1 \times inputFrequency$ | | |

Programmer's Guide Page 50 of 64

| Model | Input Frequency | sFmax vs. nbrConvertersPerChannel | | | | |
|----------------------------|--------------------|-----------------------------------|---------------------------|---|--|--|
| | range (MHz) | 1 | 2 | 4 | | |
| DP235/AP235 | 20 - 1000 | ¹∕₂ × inputFrequency | $1 \times inputFrequency$ | | | |
| DP240/AP240 | 20 - 2000 | ½ × inputFrequency | $1 \times inputFrequency$ | | | |
| DP306 | 100 - 200 | ½ × inputFrequency | | | | |
| DP306 Rev A | 100 - 400 | ½ × inputFrequency | | | | |
| DP308 | 100 - 400 | ½ × inputFrequency | | | | |
| DP308 Rev A | 100 - 400 | 1 × inputFrequency | | | | |
| DP310 | 100 - 420 | $1 \times inputFrequency$ | | | | |
| DP310 Rev A | 100 - 400 | $1 \times inputFrequency$ | | | | |
| AC210/SC210 | 20 - 2000 | $\frac{1}{2}$ × inputFrequency | | | | |
| AC240/SC240 | 20 - 2000 | ½ × inputFrequency | $1 \times inputFrequency$ | | | |
| U1084A | 1000 or 2000 | 1 x inputFrequency | 2x inputFrequency | | | |
| 1 st generation | 100 - 500 | $1 \times inputFrequency$ | | | | |

Example: When using a DC241 with 2 combined channels, and an external clock frequency of 1800 MHz (= inputFrequency), the possible sampling frequencies are 3.6 GS/s, 1.8 GS/s, 900 MS/s and 450 MS/s.

The ratio of sFmax to inputFrequency can also be learned at run-time by using a call to **Acqrs_getInstrumentInfo**(instrID," ExtCkRatio", &ratio).

The system computes the required memory overhead (in data samples) on the basis of the current **sampFrequency** and **nbrConvertersPerChannel**. Use the function **AcqrsD1_bestNominalSamples** to obtain the maximum available memory, after setting **these parameters**.

The equivalent of **delayTime** is defined with the value **delayNbrSamples**, which only applies to external clock operation. The actual delay value is easily computed as follows:

```
delay = (delayNbrSamples - nbrSamples) / sampFrequency
```

Example: In a 1st generation module with an external clock running at 200 MHz, if you wanted to acquire 2000 data points at 50 MS/s with the trigger point at the end of the first quarter of the time window, you would use the code:

```
AcqrsD1_configMemory(instrID,2000,1);
AcqrsD1_configExtClock(instrID,1,threshold, 1500, 2.0e+8, 5.0e+7);
AcqrsD1_acquire(instrID); // start the acquisition
AcqrsD1 waitForEndOfAcquisition(instrID, timeOut);
```

The value of **delayNbrSamples** is 1500 because 500 points need to be acquired *before* and 1500 points *after* the trigger, in order to position the trigger point at the 1st quarter of the time window.

Equivalently, you could have computed the time window to be 2000 x 20 ns = 40 μ s. The delay would therefore have to be -10 μ s to get the trigger point to the 1st quarter of the time window. Since **nbrSamples** = 2000 and **sampFrequency** = 5.0e+7, you would obtain **delayNbrSamples** = 1500.

Since the sampling frequency is known in this clock mode, through the variable **sampFrequency**, any read functions correctly return the value of the sampling interval.

In addition, if the user-supplied clock frequency **inputFrequency** is > 800 MHz on DC271-FAMILY digitizers or in all cases for 10-bit and 12-bit digitizers, the system correctly measures the value **horPos** and returns it with any waveform read function, such as **AcqrsD1_readData**. Thus, the time position of the sampled data points can be known to within a small fraction of the sampling interval, permitting very precise timing measurements as with the internal clock. However, the digitizer must be calibrated at the external clock frequency in use, whenever **inputFrequency** or **sampFrequency** are changed. Use this code:

Programmer's Guide Page 51 of 64

```
// We assume that a normal calibration has been done, either
// during initialization, or explicitly
AcqrsDl_configExtClk(..) // Set to (cont) Ext Clk
// Make sure to apply the same external frequency as the value
// 'inputFrequency', set in the function call above
Acqrs calibrateEx(instrID, 2, 0, 0);
```

The function **Acqrs_calibrateEx** with **calType** = 2 readjusts some timing calibration constants, but does not modify any vertical adjustment values, such as gain or offset.

In first generation digitizers, or when **inputFrequency** is ≤ 800 MHz in the second generation digitizers (or AP/AC/SC analyzers in the digitizer mode), the data read functions will return **horPos** = 0.0, equivalent to a timing uncertainty of \pm 0.5 samples. For implementation reasons, the acquired waveform in fact has a timing uncertainty that is twice as large, i.e. \pm 1 samples. In this case, the trigger time stamps of the sequence acquisition mode are not available.

Depending on the ratio of sampFrequency/inputFrequency, a waveform is sampled either on negative-going transitions of the external clock signal through the user-defined threshold or, when the ratio is > 1, on both of the transitions.



NOTE: First generation digitizers that have more than one converter/channel (DC240, DP210, and DP211) will generate two data samples for each sampling interval. You must dimension your acquisition and readout for twice the normal amount of data and can then either, drop every other data sample from the record, or average the two data values which could enhance the signal to noise ratio.

3.16.3. External Clock (Start/Stop)

The *start/stop external clock* mode (**clockType** = 4) permits the application of a (variable) external clock. It should not be used for the 10-bit-FAMILY, 12-bit-FAMILY, U1071A-FAMILY, or U1084A-FAMILY digitizers. The clock can be setup to give bursts during which the frequency is between 10 MHz and 500 MHz. The first sample of each burst may have to be ignored. The waveform is sampled on positive-going transitions of the external clock signal through the user-defined threshold. Thus, the sampling rate is equal to the input frequency.

For digitizers and Averagers/Analyzers in the digitizer mode, there is no concept of trigger when a Start/Stop clock is used. Therefore, all trigger parameters will be ignored. This also means that there is no concept of sequence acquisition. Operation in a channel combined mode is not possible.

The AC/SC Analyzers can be used in this mode. A continuous clock frequency of up to 800 MHz, to give 800 MS/s sampling, will work.

In this mode, the horizontal control parameters **sampInterval** and **delayTime** are completely ignored, as well as the value of **delayNbrSamples**. The waveform length is, as usual, controlled by the number of samples in the function **AcqrsD1_configMemory**. Careful synchronization between the function calls to the driver and the generation of the clock burst is required.

There are 2 ways of terminating an acquisition in the start/stop mode:

- 1. Generate a number of clock transitions that corresponds exactly to the requested number of samples, and stop the acquisition with the function **AcqrsD1_stopAcquisition**. This requires that the host computer obtain some external signal when the clock sequence is terminated.
- 2. Generate some extra clock transitions, which will fully terminate the acquisition. You can then use the functions **AcqrsD1_acqDone** or **AcqrsD1_waitForEndOfAcquisition** to detect the end of acquisition.

Example for Termination (1): if you wanted to acquire 20 waveforms of 2000 data points each, at a sampling rate of 33.3 MHz, and a time distance of 5 μs between the waveforms, you would use:

```
AcqrsD1_configMemory(instrID, 40000, 1);
AcqrsD1_configExtClock(instrID, 4, threshold, 0, 0., 0.);
AcqrsD1_acquire(instrID); // start the acquisition
```

→ Generate 20 bursts of 2000 clock pulses at 33.3 MHz. At the end, you need to inform the host computer to terminate the acquisition and:

Programmer's Guide Page 52 of 64

```
AcqrsD1_stopAcquisition(instrID);
AcqrsD1 readXXXWform(instrID, . . .);
```

Note that the sampling rate and the time between bursts have no incidence on the configuration parameters of the digitizer, i.e. they appear nowhere.

Example for Termination (2): if you wanted to acquire 5000 waveforms of 200 data points each, you would write:

```
AcqrsD1_configMemory(instrID, 100000, 1);

AcqrsD1_configExtClock(instrID, 4, threshold, 0, 0.0, 0.0);

AcqrsD1_acquire(instrID); // start the acquisition

→ Generate 5000 bursts of 200 clock pulses at the required frequency. At the end, you need to generate ≥ 160/320/640 additional clock pulses.

AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);

AcqrsD1_readXXXWform(instrID, . . .);
```

The 320 (640 on 2 GS/s, or 1280 on 4 GS/s digitizers) extra clocks could be generated by 1 or more extra bursts of 200 clock cycles or a special burst. There is no risk of overwriting the earliest data, since the memory is **not** circular in this mode.

Comments valid for both termination mechanisms:

Although the function **AcqrsD1_acquire** sets it to the ready state, the digitizer cannot actually record data while the external clock is idle. The clock burst must start *after* the digitizer has been started, and it must start in a very clean way, i.e. the first pulse must be already well over the threshold and its width must be ≥ 1 ns.

The digitizer sees the multiple clock bursts as a single acquisition. It knows neither the sampling frequency, nor the time difference between the waveforms. It simply acquires a number of data points. In termination mechanism (2) it also records the extraneous points and then stops.

When reading the data, the multiple waveforms appear as a contiguous waveform. The only way of distinguishing one waveform from the other is by counting samples, i.e. the first 2000 samples belong to the 1st waveform, the next 2000 to the 2nd etc. It is therefore imperative to exactly control the number of clocks in a burst.



NOTE: If the time difference between 2 bursts is > 100 ns, the digitizer tends to drift into saturation, from which it has to recover when the next burst resumes. The first data sample of such a burst is thus invalid. In many cases, this first data sample is sufficiently different from the rest of the waveform that it can serve as a 'segment marker'.



NOTE: Digitizers that have more than one converter/channel (DC211, DC240, DC241, DP210, DP211, and DP214) will generate two (four-for theDC211) data samples for each sampling interval. You must dimension your acquisition and readout for twice (4x) the normal amount of data and can then either, drop the extra data sample from the record, or average the data values which could enhance the signal to noise ratio.

3.17. AS bus Operation

The AS bus and AS bus 2 are intended to synchronize a number of similar CompactPCI modules, in order to make them appear as a single instrument with more channels. After a number of digitizers have been combined with the functions **AcqrsD1_multiInstrAutoDefine** (or **AcqrsD1_multiInstrDefine** for AS bus only), each combined instrument can be controlled, when using its **instrumentID**, with the same functions as single digitizers. We recommend the use of the automatic function, unless you need special control over the order in which the digitizers are numbered within the *MultiInstrument*. Please refer to the function **AcqrsD1 multiInstrAutoDefine**, for details.

If you mix modules (of the same model number) with different memory lengths, you must make sure that you never use more than the shortest memory length available. Otherwise, you will get invalid data on the short memory modules. Under AS bus, the automatic function always assigns the clock master role to a module with the shortest memory, with the result that the function **AcqrsD1_configMemory** refuses to accept memory lengths beyond the shortest. When configuring manually, you might want to do the same. Otherwise, you need to explicitly check your requested memory lengths.

Programmer's Guide Page 53 of 64

3.17.1. Channel Numbering with AS bus

In a *MultiInstrument*, input channels are numbered from 1 to **nbrChannels**. The number of channels can be retrieved with the function call:

```
Acgrs getNbrChannels(instrID, &nbrChannels);
```

Channel 1 corresponds to channel 1 of module 0. Channel numbers increase first through module 0, then through modules 1, 2 etc.

For Acqiris CC10x crates when a *MultiInstrument* is defined with **AcqrsD1_multiInstrAutoDefine**, module 0 is always closest to the controller slot (in Acqiris CC10x crates), whereas with **AcqrsD1_multiInstrDefine**, it corresponds to the first module in the initializing list. The Acqiris CC121 crates have a different ordering; please refer to the Acqiris CC121 CompactPCI Crate User Manual.

E.g. when combining 3 DC270 4-channel digitizers, you would use channel number 10 in the function calls **config/get_MultiInput**, **config/get_Vertical** and **readData**, if you wanted to refer to Input 2 of the third DC270.

Channel numbering does not depend on which module is the actual clock or trigger master.

3.17.2. Trigger Source Numbering with AS bus

Acqiris digitizers do not necessarily have as many internal triggers as channel inputs, nor exactly one external trigger. You should retrieve, for every *MultiInstrument*, additional information with the following calls:

```
Acqrs_getInstrumentInfo(instrID, "NbrInternalTriggers", &nbrIntTrigs);
Acqrs_getInstrumentInfo(instrID, "NbrExternalTriggers", &nbrExtTrigs);
Acqrs_getInstrumentInfo(instrID, "NbrModulesInInstrument", &nbrModules);
nbrIntTrigsPerModule = nbrIntTrigs /nbrModules;
nbrExtTrigsPerModule = nbrExtTrigs /nbrModules;
```

In a *MultiInstrument* composed of 4 DC240 (2 channel, 2 GS/s digitizers), you would get 8 internal trigger sources, 4 external trigger sources and 4 for the value of **nbrModules**. Thus, **nbrIntTrigsPerModule** would be 2 and **nbrExtTrigsPerModule** would be 1, as expected for a DC240.

Internal triggers are associated to the input channels, and follow the same numbering rules.

External triggers follow similar rules, i.e. extTrig = 1 corresponds to external trigger 1 of the first module, extTrig = 2 corresponds to external trigger 2 of the first module (if **nbrExtTrigsPerModule** > 1) or to external trigger 1 of the second module etc. The externalTrigger 2 is the name used for the PXI Bus Star Trigger.

The functions $AcqrsD1_configTrigSource$ and $AcqrsD1_getTrigSource$ use the explicit trigger channel number, with the internal trigger channel running from 1 to nbrIntTrigs, and the external trigger running from -1 to -nbrExtTrigs. Note that 0 is an invalid trigger source, resulting in an error code.

The functions **AcqrsD1_configTrigClass** and **AcqrsD1_getTrigClass** encode the trigger source in a 32-bit source pattern:

| 31 | 30 | 20 - 2 | 9 | 16 – 19 | 5 - 15 | 4 | 3 | 2 | 1 | 0 |
|----------|--|----------------|-------|--|--|----------|--------|---------|----|------|
| Ext 1 | Ext 2 | Other Ext | Trigs | Module | Module Other Int Trigs In 5 In 4 In 3 In 2 | | | | | In 1 |
| [0] IN 1 | | | | Internal trigg | ger channel 1 | | | | | |
| [] | [1] IN 2 Internal trigger channel 2 | | | | | | | | | |
| [24 | 4] | IN 3, IN 4, | IN 5 | Internal trigg | ger channels 3, 4, 5 | | | | | |
| [515 | 5] | OTHER TRIGS | INT | Other internal trigger channels within a module, up to 16. | | | | | | |
| [1619 | Module Number, running from 0 to (nbrModules – 1). In single digitizers, this field must be zero. In <i>MultiInstruments</i> , the trigger source number must be broken module number and a trigger channel number within the modu | | | | | | | | | |
| [2029 | 9] | OTHER TRIGS | EXT | Other extern | al trigger channels | within a | module | up to 1 | 2. | |
| [30 | 0] | EXT 2 | | External trig | gger channel 2 | | | | | |
| [3] | 1] | EXT 1 | | External trig | ger channel 1 | | | | | |

Programmer's Guide Page 54 of 64

In digitizers with trigger pattern capabilities, several trigger bits could be set simultaneously. However, no trigger pattern capabilities *between different modules* can be coded, i.e. only a single module in a MultiInstrument can be the trigger source, although the source in the single module might be a pattern. For these reasons, the module number must be coded explicitly.

To translate a trigger channel number **trigChan** into a trigger source pattern, use the following code:

Note that **moduleNbr** and **inputNbr** start from 0. An 'industrial strength' implementation should contain some checks on the range of **trigChan** and/or **inputNbr**.

3.18. Special Operating Modes

Some Acqiris digitizers offer alternative operating modes, which are controlled with the function $\mathbf{AcqrsD1_configMode}$. The default state of any digitizer is $\mathbf{mode} = 0$ and $\mathbf{flags} = 0$, corresponding to the *normal* digitizer operation, as described in the other sections of this manual. Please refer to the Programmers Reference for details on this function.

The available modes are model and option dependent; please refer to the specific model data sheet or user manual for more information. The table below gives a summary of all modes, and the following sub-sections provide details on these modes.

| Mode Index | Name | Description | Available on models | | | |
|---------------|-------------------|---|--|--|--|--|
| 0 | Normal | Normal (digitizer) acquisition mode. | All | | | |
| 1 | Stream data | Stream data to the DPU. | SC2x0, AC2x0 Data Streamers only. | | | |
| 2 | Averager | Averaging mode. | AP235, AP240, AP100, AP200, U1084A only. | | | |
| 3 | Dual memory | Dual memory mode. | AP101, AP201 only. | | | |
| 5 | Peak TDC | Peak detection and Time-to-Digital Conversion and Analysis mode. | Peak TDC Analysers (AP240, U1084A) only. | | | |
| 6 | Frequency Counter | Frequency counter mode. | DC135, DC140 only. | | | |
| 7 | SSR | Sustained Sequential Recording mode | AP235/240, or U1084A-DGS only. | | | |
| 12 | DDC | Digital Down-Converter mode. | M9202A with –DDC option only. | | | |
| 14 | Custom firmware | (Please contact Agilent Sales for further information) | | | | |

Note: After switching between modes it is recommended to perform an internal calibration to ensure that the instrument is performing within specification.

Programmer's Guide Page 55 of 64

3.18.1. Normal Acquisition (mode 0)

The normal digitizer operating mode which is present on most instrument models. It also has sub-modes which may be enabled by using the 'flags' parameter of the AcqrsD1_configMode function.

3.18.1.1. 'Start on Trigger' in Normal Acquisition mode

The 'Start on Trigger' mode is a sub-mode of Normal Acquisition mode, it begins data recording only upon receipt of a trigger signal, and stops after **nbrSamples** data points are acquired. Not all digitizers are capable of this mode;

```
those that never have it are the DC110, DC240, DC265, DC270, 12-bit digitizers, and U1084A; others (DP105, DP106, DP110, DP111, DP210, DP211, and DP212) can have it as an option only.
```

It is useful in the special case where the sampling rate is **less** than the maximum possible and where an optimum time correlation between the trigger and the sampling clock is required (typically when averaging waveforms). This mode also requires that the trigger is available *before* the waveform of interest.

In the 'Normal' mode, data recording begins at the time of arming, with the function **AcqrsD1_acquire**. The trigger occurs asynchronously to the sampling clock, and thus will fall randomly anywhere within a sampling interval. When averaging waveforms, this will result in an effective bandwidth reduction since the waveforms are randomly shifted with respect to each other by up to $\pm \frac{1}{2}$ sampling interval.

In 'Start on Trigger' mode, the trigger occurs *before* recording starts. It still occurs asynchronously with respect to the internal reference clock (which is *always* running). However, if the requested sampling rate is less than the internal reference clock frequency (e.g. 100 MS/s, while the clock runs at 500 MHz), then the time correlation between the trigger and the effective sampling clock is within $\pm \frac{1}{2}$ internal reference clock time interval, **not** $\pm \frac{1}{2}$ sampling interval. Therefore when averaging, the bandwidth reduction will be less than in the 'normal' mode. In general, the internal reference clock runs at the upper frequency shown for the model-dependent "Input Frequency range" shown in the table of section 3.15.2 **External Clock (Continuous)**.

The value **delayTime** in the function **AcqrsD1_configHorizontal** is ignored. As usual, the digitizer requires some memory overhead for additional samples. The function **AcqrsD1_bestNominalSamples** returns the maximum number of available samples.

Use this code to use the 'Start on Trigger' mode:

Note that the function **AcqrsD1_acquire** is still needed. However, it behaves somewhat differently in that is does *not* start data recording but waits until a trigger signal is received.

Due to some circuit delays, the waveform recording starts approximately 20ns after the receipt of the trigger signal. Furthermore the first data points may be invalid. For the DC271 family, this means that the first 8 ns worth of data should be ignored for sampling rates 4 GS/s > SR > 500 MS/s and the first 4 ns, 16 points, for SR = 4 GS/s.

Note: This function is not compatible with ASBus.

3.18.1.2. 'Sequence Wrap' in Normal Acquisition mode

The normal operation of the digitizer requires that it stop recording waveforms when the pre-defined number of segments has been acquired. Thus, **nbrSegments** triggers are needed to acquire the requested number of segments into the same number of different memory sections. After the acquisition has terminated, all of the waveform segments are finally available for readout.

The 'Sequence Wrap' mode also pre-defines the desired number of different memory sections, but it permits a larger number of triggers. After the first **nbrSegments** waveform segments are acquired, the digitizer 'wraps' around to the first memory segment and keeps on recording waveforms. This sequence can go on indefinitely, since no hardware condition will stop it. The only way to terminate this infinite loop is to stop it with the function **AcqrsD1_stopAcquisition**. This mode is available in all digitizers except the U1071A-FAMILY, 10-bit-FAMILY and U1084A-FAMILY.

This mode is useful when only the last N out of many occurrences of a signal are of interest. E.g. if you search for a rare event out of many occurrences, and you only can determine its interest *after* the event has occurred, then the 'Sequence Wrap' mode is applicable.

Programmer's Guide Page 56 of 64

While this mode even 'works' for **nbrSegments** = 1, in practice the value of **nbrSegments** should be at least 3. It is important to note that after the acquisition of a segment, the digitizer automatically advances to the next memory section and immediately (with a dead time of $\sim 1~\mu s$) starts recording into it. Thus, usually the very last memory segment used will necessarily contain uninteresting data, since it will not be stopped with a trigger, but be terminated with the software command **AcqrsD1_stopAcquisition**.

Use this code to use the 'Sequence Wrap' mode:

```
AcqrsD1_configXXX(..);  // configure other parameters
AcqrsD1_configMode(instrID, 0, 0, 2);
AcqrsD1_acquire(instrID);
..
AcqrsD1_stopAcquisition(instrID);
```

The time at which the sequence is terminated with the function **AcqrsD1_stopAcquisition** depends on an external event, e.g. operator intervention.

When reading the segments, the segment number should take on the values 0,...(**nbrSegments**-1). They correspond to the memory section numbers in the digitizer, **not** the time order of the acquired segments.

Example: if **nbrSegments** = 8, the time order of the acquired segments might be (depending on when the sequence was stopped) 5, 6, 7, 0, 1, 2, 3, 4. Here, the 'oldest good' segment is the 5^{th} segment, followed by the 6^{th} , 7^{th} , 8^{th} , 1^{st} etc. The 'youngest' useful segment is the 2^{nd} one, while the 3^{rd} or 4^{th} segment corresponds to the segment that was being recorded when the stop-command was received. The 4^{th} segment (in this example) usually does not contain any useful data. However, depending on the timing of the triggers with respect to the acquisition stop it could be that the 3^{rd} segment has the corrupted data and the 4^{th} is valid.

3.18.2. Stream Data (mode 1)

The SC2x0 Streamer Analyzers (U1091ASC1, U1080A-002), and the AC2x0 Signal Analyzers (U1091AAC1, U1080A-001) support this data streaming mode. Please refer to the specific product manuals for further information:

- Signal Analyzers, User Manual (U1092-90006)
- Streamer Analyzers User Manual (U1092-90007)

3.18.3. Averager (mode 2)

In Averaging mode, newly acquired waveforms are summed in real-time with the corresponding samples of previous waveforms. The corresponding samples are those with identical delay calculated from the beginning (trigger time) of the acquisition. This mode is supported by AP235/AP240 (U1082-AVG), AP100/AP200 (U1081A) and U1084A models only. For further information please refer to the specific user manual:

- Averagers User Manual (U1092-90011)
- MD1 Digitizer User Guide & SFP Help U1084A Averager section.

3.18.4. Dual Memory (mode 3)

The AP101, AP201 (U1081A) Analyzers support a Dual Memory or Buffered data acquisition mode. The Buffered Sequence Acquisition mode allows the capture and storage of consecutive "single" waveforms. Sequence Acquisition mode is useful as it can optimize the digitizer's sampling rate and memory requirements for applications where only portions of the signal being analyzed are important. For further information please refer to the specific product manual:

• Analyzers, User Manual (U1092-90020)

3.18.5. Peak^{TDC} (mode 5)

This mode is supported by PeakTDC Analyzer models AP240 (U1082-TDC) and U1084A-TDC model. The peak detection and analysis mode allows real-time acquisition and peak detection. The mode also enables the creation of a histogram of peak position vs. time for successive acquisitions, the histogram bins containing peak counts or summed peak amplitudes. Please refer to the specific product manual for further information:

- Analyzers, User Manual (U1092-90020)
- MD1 Digitizer User Guide & SFP Help U1084A Peak TDC section.

Programmer's Guide Page 57 of 64

3.18.6. Frequency Counter (mode 6)

This is an option available for the DC140 and DC135 digitizers. It is implemented with a signal counter that counts trigger signals from the user-requested channel. A time counter generates the user-programmed aperture time, during which the measurement is performed.

The user-requested signal channel has to be programmed for the expected signal characteristics; the standard config functions should be used to set the full-scale, coupling, offset and trigger threshold. The HF trigger mode may be set by the driver software on the basis of the user-supplied target frequency value. In order to obtain the best results, it is recommended to adjust the full scale and offset so that they span the expected input signal. In addition, the trigger threshold should be set to approximately the center of the signal voltage range. Calls to **AcqrsD1_configVertical** and **AcqrsD1_configTrigClass** are needed. If an external time base reference is desired, **AcqrsD1_configExtClock** should be called. If the totalize in gate functionality is desired the source of the gate must be set with the function **AcqrsD1_configControlIO**, with the parameters *connector* = 1 (I/O A) or 2 (I/O B) and *signal* = 9. Note that when this mode is in use the Enable trigger input (signal = 6) functionality of **AcqrsD1_configControlIO** cannot be used.

The function AcqrsD1_configFCounter sets the parameters specific to the frequency measurements.

Comments:

- Channel numbers run from 1 to the available number of signal channels in the digitizer.
- The value type is 0 for Frequency, 1 for Period, 2 for Totalize by Time and 3 for Totalize by Gate.
- The *targetValue* is an estimator of the expected result. If no estimate is possible, use the value 0.0. This value is only used to activate the HF trigger mode which extends the useable frequency range. By default, the frequency range is extended except:

| type | Measurement | Divide by 1 |
|------|------------------|---|
| 0 | Frequency | if targetValue is smaller than 1 kHz (1000.0) and larger than 0.0 |
| 1 | Period | if targetValue is larger than 1ms (0.001) |
| 2 | Totalize by Time | always (the HF mode is never used) |
| 3 | Totalize by Gate | always (the HF mode is never used) |

• The *apertureTime* defines the minimum time for a frequency measurement (it may be longer if the frequency is very low!). In the Totalize by Time mode, the value of *apertureTime* determines the time window during which the input pulses are counted.

The frequency counter mode is set with the function **AcqrsD1 configMode**, with mode = 6.

After configuring the instrument parameters, the measurement sequence is started with the function AcqrsD1_acquire. This function returns before the measurement is terminated. The user must wait until it is terminated with the functions AcqrsD1_acqDone or AcqrsD1_waitForEndOfAcquisition. For the case of Totalize by Gate the program must stop the acquisition by making a call to the function AcqrsD1_stopAcquisition.

FC results can be readout with the function **AcqrsD1_readFCounter**. The result is always a single double precision number whose units are those appropriate for the *type* of measurement chosen.

3.18.7. SSR – Sustained Sequential Recording (mode 7)

SSR mode is supported by certain AP235/240 and U1084A models. SSR functionality includes a ping-pong memory buffer architecture. The processing memory is split in two so that module can continuously acquire data into one memory bank while the other bank is read into the host processor. This technique is particularly useful for continuous acquisition of data bursts at a sustained trigger rate.

For further information please refer to section 3.5 of this document.

3.18.8. DDC – Digital Down-Conversion (mode 12)

Digital Down-Conversion is currently supported only on the M9202A model. It allows for operation as the Intermediate Frequency (IF) digitization element in a digital receiver. For futher information on the configuration and operation, please refer to the 'Digital Down-Conversion' topic in the MD1 User Guide & SFP Help system.

Programmer's Guide Page 58 of 64

3.19. Readout of Battery Backed-up Memories

Acqiris digitizers with the battery back-up option permit retaining acquired waveforms during periods of time when the power might be interrupted.

3.19.1. Preparations before Power-Off

A digitizer only remembers the digitized data array, but not all of the parameters that are needed to interpret the waveform. These parameters are normally retained in the driver, but are typically lost when power is lost or when the controlling application is terminated.

It is therefore necessary for the application to transfer the relevant parameters **before** power-off, typically to a disk, in such a way that they are again available when restarting the application after power is restored. The following parameters must be transferred to persistent storage, for each instrument:

- Parameters of AcqrsD1_configHorizontal, i.e. sampling interval and delay
- Parameters of AcqrsD1 configMemory, i.e. number of samples and number of segments
- Parameters of AcqrsD1_configVertical, i.e. Full Scale and offset (the coupling is not relevant!), for each channel of interest
- Two calibrated delay parameters, as obtained with the function calls

```
Double delayOffset, delayScale;
Acqrs_getInstrumentInfo(ID, "DelayOffset", &delayOffset);
Acqrs getInstrumentInfo(ID, "DelayScale", &delayScale);
```

The functions **Acqrs_getInstrumentInfo** should be called just before the start of the acquisition, i.e. before calling **AcqrsD1_acquire**, but after all **AcqrsD1_config...** functions.

3.19.2. Recovery after Power-Off

In order to read a battery backed-up waveform, you need to execute a special sequence of initialization functions

1. Initialize the digitizer with the following function call:

```
Acgrs_InitWithOptions(resourceName, false, false, "CAL=FALSE", &instrID);
```

It is important to specify "CAL=FALSE" to prevent any calibration in the digitizer, which would destroy any retained data.

- 2. Call the functions **AcqrsD1_configHorizontal**, **AcqrsD1_configMemory**, **AcqrsD1_configVertical** (for each channel!) with the same parameters that were used in the original acquisition.
- 3. Call the function **AcqrsD1_restoreInternalRegisters**, with the parameters *delayOffset* and *delayScale*. In case these parameters are not available (e.g. due to earlier software versions), you should use the values –20.0e-9 for *delayOffset* and 5.0e-12 for *delayScale*.
- 4. Call the function **AcqrsD1 readData** to read the battery backed-up data.

Failing to restore the originally used digitizer parameters may result in erroneous data. After data recovery, and before using the digitizer for any new acquisitions, don't forget to calibrate the instrument with the function **Acqrs_calibrate**.

3.20. Reading the Instrument Temperature

The temperature of an instrument can be obtained with the following code:

```
long temperature; // will be in degrees C
Acqrs_getInstrumentInfo(instrID, "Temperature", &temperature);
```

When multiple digitizers are combined via AS bus to a MultiInstrument, use the strings "Temperature 0", "Temperature 1"... to refer to the individual modules.



NOTE: The returned temperature value corresponds to the ambient temperature on the main printed-circuit board, typically near the timebase circuit. It cannot represent all possible temperature values that are present on the circuit. Values ≥ 60 °C indicate that the circuit is near its operational limit, and a cooling failure occurred or better cooling should be installed.

We recommend keeping the temperature as low as possible, since a 10°C reduction in circuit temperature is expected to improve the mean-time-between-failures (MTBF) by a factor of 2.

Programmer's Guide Page 59 of 64

We also recommend reading the temperature when the instrument is stopped. The read operation may generate small signal perturbations through cross talk, if it is executed while an acquisition is in progress.

3.21. Building applications that can Hibernate

The Acqrs_powerSystem function was implemented to be called from a routine that treats Windows events. The code below shows a standard treatment of such events:

```
switch (winMsgP->message)
{
case WM_POWERBROADCAST:
    switch (winMsgP->wParam)
    {
    case PBT_APMRESUMEAUTOMATIC:
        status = Acqrs_powerSystem(AqPowerOn, 0);
        break;
    case PBT_APMSUSPEND:
        status = Acqrs_powerSystem(AqPowerOff, 0);
        break;
    default: // There are other power events, but we only show these two.
        break;
    }
    break;
default:
    break;
}
```

If such code is not available the Acqiris devices will not be useable when the PC becomes active after hibernation.

Programmer's Guide Page 60 of 64

4. Appendix A: Estimating Data Transfer Times

The time to transfer a waveform may be a significant part of the execution time of a program, and thus becomes an important design consideration for new applications. We present here a simple timing model with the aim of predicting the transfer time for digitizer readout as a function of the number of segments, number of samples, CPU speed and the operating system.

4.1. Principles & Formulas

The function **AcqrsD1_readData** for readMode = 0 executes a direct waveform transfer from the digitizer memory to the user-allocated buffer, for a single segment at a time. Since direct-to-memory access (DMA) is used, this is the most time-efficient method for segments of 10'000 or more samples. However, each segment requires its own DMA setup. When segments are very short, the transfer overhead starts to dominate the overall transfer time. After the transfer the data in the buffer is ready to be used.

The function **AcqrsD1_readData** for readMode = 1 (ReadModeSeqW) reduces the overhead time by transferring with a single DMA a complete digitizer memory image to the host computer memory. The memory image is composed of a number of segments, each of which is a *circular* buffer. The first data point of interest in the circular buffer may be anywhere, and its position usually changes randomly between acquisitions. Thus, after the DMA is terminated, the driver must copy data from the image to the final linear buffer for each segment. This method is therefore only interesting for relatively short segments. However, if timing is not an important consideration, it offers the convenience of a single function call for the complete transfer of a waveform sequence.

Formulas for estimating the transfer times in µs are:

• **AcqrsD1_readData** for readMode = 0:

$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr$$

• **AcqrsD1_readData** for readMode = 1 (ReadModeSeqW):

$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$

with the following definitions:

M Number of segments

Number of samples per segment

Xfr Transfer time per sample in μ s, typically 0.01 ??

 $Ovhd_{DMA}$ DMA overhead time (per segment) in μ s

 $Ovhd_{buffer}$ Circular buffer analysis overhead time (per segment) in μs

Extra Number of 'overhead' data points per segment

Cpy Time to copy a sample in μ s

The formulas above assume that M and N correspond to the number segments and samples set with the function **AcqrsD1_configMemory**. If fewer segments or samples are transferred, the timing might be somewhat less favorable than estimated by the formula.

- 1. The transfer time *X_{fr}* is typically 0.009 to 0.01 μs (9 10 ns) per 8-bit sample for digitizers directly inserted on the PCI bus of the host computer or connected through the SBS-Bit3 interface (Acqiris model number IC200). If a digitizer is connected through the National Instruments MXI-3 interface, *X_{fr}* is typically 0.012 μs (12 ns). Of course, these times would get larger, if there was considerable additional I/O traffic on the PCI bus.
- 2. The DMA overhead time $Ovhd_{DMA}$ is approximately (50'000 µsec) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium $Ovhd_{DMA}$ is ~ 200 µs, and for a 500 MHz Pentium it is ~ 100 µs. With applications running under Windows 95, $Ovhd_{DMA}$ has been observed to be ~ 20% lower.
- 3. The circular buffer analysis overhead time $Ovhd_{buffer}$ is ~ (2'500 μ s) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium $Ovhd_{buffer}$ is ~ 10 μ s, and for a 500 MHz Pentium ~ 5 μ s.
- 4. An *Extra* number of data samples must be transferred to the host computer with the memory image. It depends on the sampling interval (in ns) and on the digitizer type. Here are some rough values:

Programmer's Guide Page 61 of 64

| Extra | Туре |
|--------------------------|--|
| 300 / sampInterval + 98 | digitizers with ≤ 1 GS/s maximum sampling rates |
| 300 / sampInterval + 194 | digitizers with 2 GS/s maximum sampling rates or DC271-FAMILY instruments set for a combine channel mode allowing 2 GS/s |
| 300 / sampInterval + 386 | DC271-FAMILY instruments set for a 4 GS/s combine channel mode |

5. The copying time Cpy per 8-bit sample is ~ (5 μ s) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium Cpy is ~ 0.02 μ s, and for a 500 MHz Pentium it is ~ 0.01 μ s.

Benchmarks were run on 266 and 550 MHz Pentiums, running under Windows 98 and NT. The observed transfer times agreed with the formula within better than 20%.

• dataType=3:

Add the conversion time T_3 for conversion from ADC codes to Volts, to T_2 or T_1 respectively:

$$T_3 = M \cdot N \cdot Conv$$

where Conv is the conversion time per sample in microseconds. It is ~ (35 μ s) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium Conv is ~ 0.14 μ s, and for a 500 MHz Pentium ~ 0.07 μ s.

4.2. Examples

(A) DP210 in a 800 MHz Pentium:

Transferring single records of 100'000 samples, recorded at 2 GS/s: $M = 1 \quad N = 100\,000 \quad Xfr = 0.01 \quad Cpy = 0.00625$ $Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672$ $T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 62.5 + 1000 = \underline{1063}\mu S$ $T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$ $= 62.5 + 3.125 + 100\,672 \cdot 0.01 + 100\,000 \cdot 0.00625 = 1698\mu S$

It is therefore more favorable to use the function $AcqrsD1_readData$ for readMode = 0.

```
Transferring 100 segments of 1000 samples, recorded at 2 GS/s: M = 100 \quad N = 1000 \quad Xfr = 0.01 \quad Cpy = 0.00625
Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672
T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100 \cdot 62.5 + 100 \cdot 1000 \cdot 0.01 = 7250 \mu s
T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy
= 62.5 + 100 \cdot 3.125 + 100 \cdot 1672 \cdot 0.01 + 100 \cdot 1000 \cdot 0.00625 = 2672 \mu s
```

The function $AcqrsD1_readData$ for readMode = 1 (ReadModeSeqW) is about 2.7 times faster.

Programmer's Guide Page 62 of 64

(B) DC270 connected to a 500 MHz Pentium with a MXI-3 interface:

Transferring single records of 100'000 samples, recorded at 100 MS/s:
$$M = 1 \quad N = 100 \, 000 \quad Xfr = 0.012 \quad Cpy = 0.01$$

$$Ovhd_{DMA} = 100 \quad Ovhd_{buffer} = 5 \quad Extra = 240/10 + 96 = 120$$

$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100 + 1200 = \underline{1300 \, \mu s}$$

$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$

$$= 100 + 5 + 100120 \cdot 0.012 + 100000 \cdot 0.01 = 2306 \, \mu s$$

It is therefore more favorable to use the function AcqrsD1 readData for readMode = 0.

Transferring 1000 segments of 500 samples each, recorded at 500 MS/s:
$$M = 1000 \quad N = 500 \quad Xfr = 0.012 \quad Cpy = 0.01$$

$$Ovhd_{DMA} = 100 \quad Ovhd_{buffer} = 5 \quad Extra = 240/2 + 96 = 216$$

$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100\,000 + 6\,000 = \underline{106\,000\,\mu s}$$

$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$

$$= 100 + 1\,000 \cdot 5 + 1\,000 \cdot 716 \cdot 0.012 + 1\,000 \cdot 500 \cdot 0.01 = 18\,692\,\mu s$$

The function **AcqrsD1_readData** for readMode = 1 (ReadModeSeqW) is about 5 times faster.

4.3. Comparison Chart for Typical Transfers

| Time in ms | | 250 MHz Pentium | | 500 MHz Pentium | | | 800 MHz Pentium | | | |
|------------------|---------------------|-----------------|-------|-----------------|-------|-------|-----------------|-------|-------|------|
| # of Segments | Samples/ Segment | T_1 | T_2 | R | T_1 | T_2 | R | T_1 | T_2 | R |
| 1 | 200 | 0.20 | 0.22 | 1.10 | 0.10 | 0.12 | 1.13 | 0.06 | 0.08 | 1.17 |
| 1 | 1 K | 0.21 | 0.25 | 1.17 | 0.11 | 0.13 | 1.20 | 0.07 | 0.09 | 1.22 |
| 1 | 10 K | 0.30 | 0.52 | 1.72 | 0.20 | 0.31 | 1.56 | 0.16 | 0.23 | 1.45 |
| 1 | 100 K | 1.20 | 3.22 | 2.68 | 1.10 | 2.11 | 1.92 | 1.06 | 1.70 | 1.60 |
| 1 | 1 M | 10.2 | 30.2 | 2.96 | 10.1 | 20.1 | 1.99 | 10.0 | 16.3 | 1.62 |
| | | | | | | | | | | |
| 10 | 200 | 2.02 | 0.43 | 0.21 | 1.02 | 0.26 | 0.25 | 0.65 | 0.19 | 0.30 |
| 10 | 1 K | 2.10 | 0.67 | 0.32 | 1.10 | 0.42 | 0.38 | 0.73 | 0.32 | 0.45 |
| 10 | 10 K | 3.00 | 3.37 | 1.12 | 2.00 | 2.22 | 1.11 | 1.63 | 1.79 | 1.10 |
| 10 | 100 K | 12.0 | 30.4 | 2.53 | 11.0 | 20.2 | 1.84 | 10.6 | 16.4 | 1.54 |
| 10 | 1 M | 102 | 300 | 2.94 | 101 | 200 | 1.98 | 101 | 163 | 1.62 |
| | | | | | | | | | | |
| 100 | 200 | 20.2 | 2.47 | 0.12 | 10.2 | 1.67 | 0.16 | 6.45 | 1.37 | 0.21 |
| 100 | 1 K | 21.0 | 4.87 | 0.23 | 11.0 | 3.27 | 0.30 | 7.25 | 2.67 | 0.37 |
| 100 | 10 K | 30.0 | 31.9 | 1.06 | 20.0 | 21.3 | 1.06 | 16.2 | 17.3 | 1.06 |
| 100 | 100 K | 120 | 302 | 2.52 | 110 | 202 | 1.83 | 106 | 164 | 1.54 |
| | | | | | | | | | | |
| 1000 | 200 | 202 | 22.9 | 0.11 | 102 | 15.8 | 0.16 | 64.5 | 13.2 | 0.20 |
| 1000 | 1 K | 210 | 46.9 | 0.22 | 110 | 31.8 | 0.29 | 72.5 | 26.2 | 0.36 |
| 1000 | 10 K | 300 | 317 | 1.06 | 200 | 212 | 1.06 | 163 | 170 | 1.06 |
| 8000 | 200 | 1616 | 182 | 0.11 | 816 | 126 | 0.15 | 516 | 105 | 0.20 |
| 8000 | 1 K | 1680 | 374 | 0.22 | 880 | 254 | 0.29 | 580 | 209 | 0.36 |
| 8000 | 2 K | 1760 | 614 | 0.35 | 960 | 414 | 0.43 | 660 | 339 | 0.51 |

Programmer's Guide Page 63 of 64

Comments:

- We assume a 2 GS/s digitizer running at the highest sampling rate, direct connection of the digitizer to the host PCI bus or through the SBS-Bit3 interface (Acqiris model number IC200), Windows NT.
- $R = T_2/T_1$ If R > 1.0, a loop over **AcqrsD1_readData** for readMode = 0 is faster than **AcqrsD1_readData** for readMode = 1 (ReadModeSeqW).

| <u>Time</u> | 866 MHz Pentium | | | | |
|------------------|---------------------|-------|-------|------|--|
| # of Segments | Samples/ Segment | T_1 | T_2 | R | |
| 1 | 200 | 0.03 | 0.04 | 1.22 | |
| 1 | 1 K | 0.04 | 0.05 | 1.21 | |
| 1 | 10 K | 0.12 | 0.17 | 1.45 | |
| 1 | 100 K | 0.98 | 2.12 | 2.15 | |
| 1 | 1 M | 9.85 | 25.5 | 2.59 | |
| 10 | 200 | 0.3 | 0.14 | 0.46 | |
| 10 | 1 K | 0.35 | 0.27 | 0.76 | |
| 10 | 10 K | 1.21 | 1.61 | 1.33 | |
| 10 | 100 K | 11.6 | 16.2 | 1.40 | |
| 10 | 500 k | 45.4 | 94.4 | 2.08 | |
| 100 | 200 | 3.00 | 0.92 | 0.31 | |
| 100 | 1 K | 4.27 | 2.37 | 0.56 | |
| 100 | 10 K | 16.1 | 15.0 | 0.93 | |
| 100 | 50 K | 44.7 | 74.2 | 1.66 | |
| 1000 | 200 | 47.2 | 13.5 | 0.29 | |
| 1000 | 1 K | 44.3 | 23.4 | 0.53 | |
| 1000 | 5 K | 85.9 | 87.8 | 1.02 | |
| 8000 | 200 | 457 | 64.8 | 0.14 | |
| 8000 | 500 | 496 | 202 | 0.41 | |

Comments: Measured under the Linux OS.

Programmer's Guide Page 64 of 64