XZ130

AGILENT ACQIRIS

PROGRAMMER'S

GUIDE

Agilent Technologies

June 2007

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 Technologies 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 Technologies Inc. or its affiliated companies.

Any versions of this manual which are supplied with a purchased product will be replaced at your request with the latest revision in electronic format. At Agilent Technologies 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 Acqiris representative or Acqiris Technical Support (support@acqiris.com) or come visit our web site at http://www.acqiris.com.

Trademarks: product and company names listed are trademarks or trade names of their respective companies

Acqiris Headquarters:	**Acqiris USA:**	**Acqiris Asia-Pacific:**
Agilent Technologies SA **12, chemin des Aulx** **CH-1228 Plan-les-Ouates** **Geneva** **Switzerland**	**Agilent Technologies Inc.** **P.O. Box 2203** **Monroe, NY 10949** **USA**	**Agilent Technologies** **Australia Pty Ltd** **347 Burwood Highway,** **Forest Hill VIC 3131** **Australia**
Tel: +41 22 884 32 90	**Tel: 845 782 6544**	**Tel: +61 3 9210 2890**
Fax: +41 22 884 32 99	**Fax: 845 782 4745**	**Fax: +61 3 9210 5929**

CONTENTS

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 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.

Chapter 4   ***RUNNING THE ACQIRIS DEMONSTRATION APPLICATION***, describes either

        the operation of AcqirisLive, an application that enables basic operation of Acqiris digitizers or averagers in a Windows 2000/XP environment;

        the operation of AP_SSRDemo and in the following chapter APx01Demo, applications that enable basic operation of Acqiris analyzers in a Windows 2000/XP environment;

        the operation of DemoTC, the demonstration program that enables basic operation of Acqiris Time Counters in a Windows 2000/XP environment;

        the operation of AcqirisAnalyzers, the demonstration program for the SC240/AC240/SC210/AC210 from a PC running a Windows 2000/XP operating system.

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:

        LabWindowsCVI, Visual C++, LabVIEW, MATLAB, Visual Basic, Visual Basic .NET.

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   DC135/DC140/DC211/DC211A/DC241/DC241A/
                         DC271/DC271A/DC271AR/DP214/DP235/DP240

AP-FAMILY   AP240/AP235/AP100/AP101/AP200/AP201

12-bit-FAMILY   DC440/DC438/DC436/DP310/DP308/DP306

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

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.

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++, Visual Basic, LabWindows/CVI, LabVIEW, and MATLAB. For VxWorks real-time systems there are sample programs for Tornado. An Application Program Interface (API) hlp file, with a shortcut named Acqiris Instrument Driver Help, is available and contains condensed descriptions of all of the interface functions.

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

- AcqrsT3 Time Counter functions - AqTx - to be used for the family of Time Counters

All of these functions are still contained in one library called **AqDrv4**. The LabView interface is split into the three 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++, VxWorks, and LabWindows/CVI all rely 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:

- Open either the project file **GetStartedVC.dsp** or **GetStartedAvgVC.dsp** in VisualC++ and build the project.

- Note that you should insert the lines

  ```
  #include "AcqirisImport.h"
  #include "AcqirisD1Import.h"
  ```

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

- The project should link to the **AqDrv4.lib** file.

For time counter users:

- Open either the project file **GetStartedTC840.dsp** or **GetStartedTC890.dsp** in VisualC++ and build the project.

- Note that you should insert the lines

  ```
  #include "AcqirisImport.h"
  #include "AcqirisT3Import.h"
  ```

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

- The project should link to the **AqDrv4.lib** file.

2.2. LabWindows/CVI

- Open the project file **GetStarted.prj** in LabWindows/CVI and build the project.

- Note that you should insert the lines

  ```
  #include "AcqirisImport.h"
  ```
  and either
  ```
  #include "AcqirisD1Import.h"
  ```
  or
  ```
  #include "AcqirisT3Import.h"
  ```

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

- The project should include the **AqDrv4.lib** file.

2.3. LabVIEW

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 three families discussed above have been placed into 3 projects. Each project contains Examples, Private, and Public Folders. The project folder also contains an Acqiris Xx Readme.html giving version information. As an example, the Tx project looks as shown below:
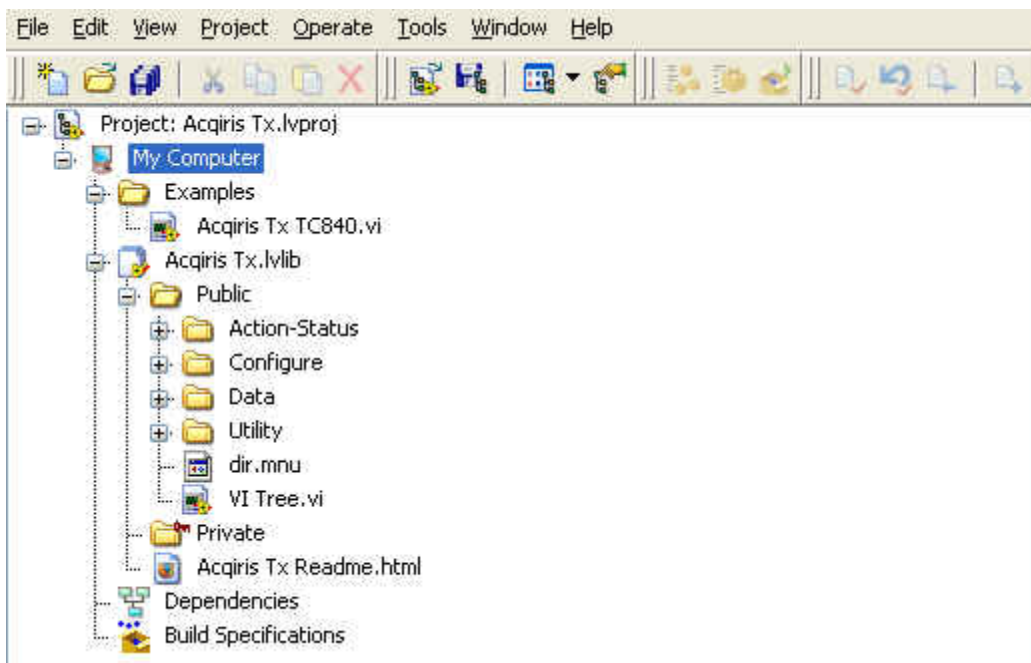
The Example folder contains ready to use potential starting points for your own programs. Individual library functions can be found at the Public level or in one of the 4 categories shown. Query functions are in the Utility folder. The block diagram of the VI Tree.vi can be used to access individual functions. It allows quick and easy access to all needed vi's.

The AqBx Generic functions family contains the following vi's.
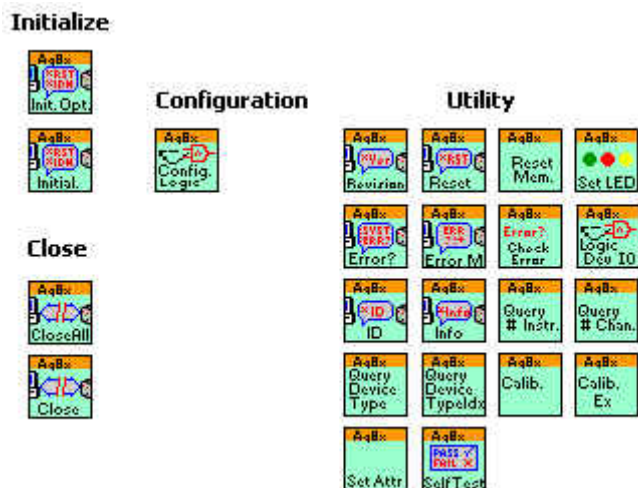
The AqBx Examples section contains three very simple cases to show some basic functionality.

The AqDx Digitizer functions family contains the following vi's. The diagram also shows the AqBx functions that can be used. These AqBx functions are to be preferred over their AqDx equivalents.
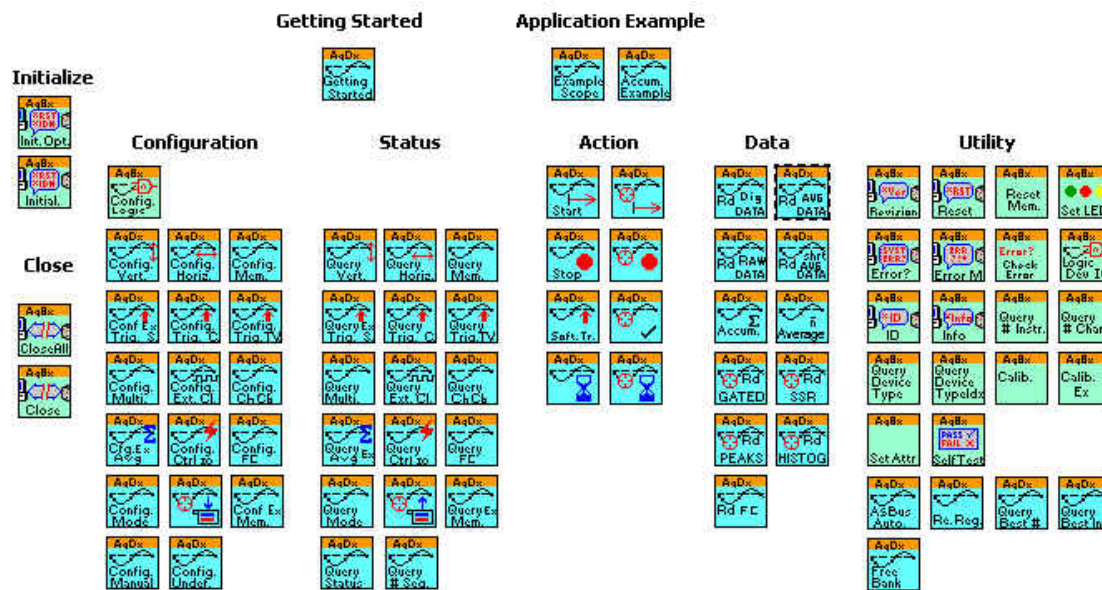
The AqTx Time Counter functions family contains the following vi's. The diagram also shows the AqBx functions that need to be used.
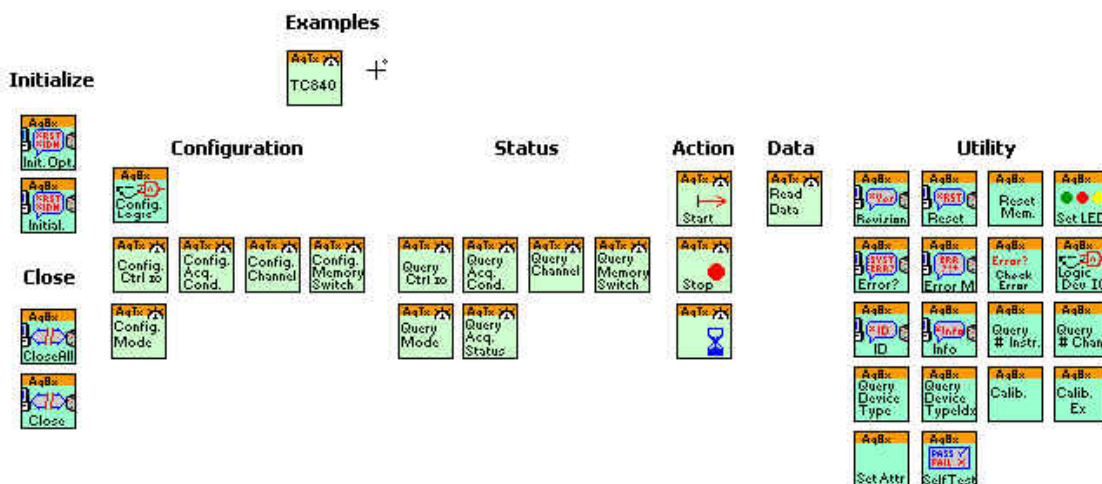
2.3.1. LabView 7.x/8 with old LabView programs

If you have an existing LabView program you can still use it or develop futher 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.

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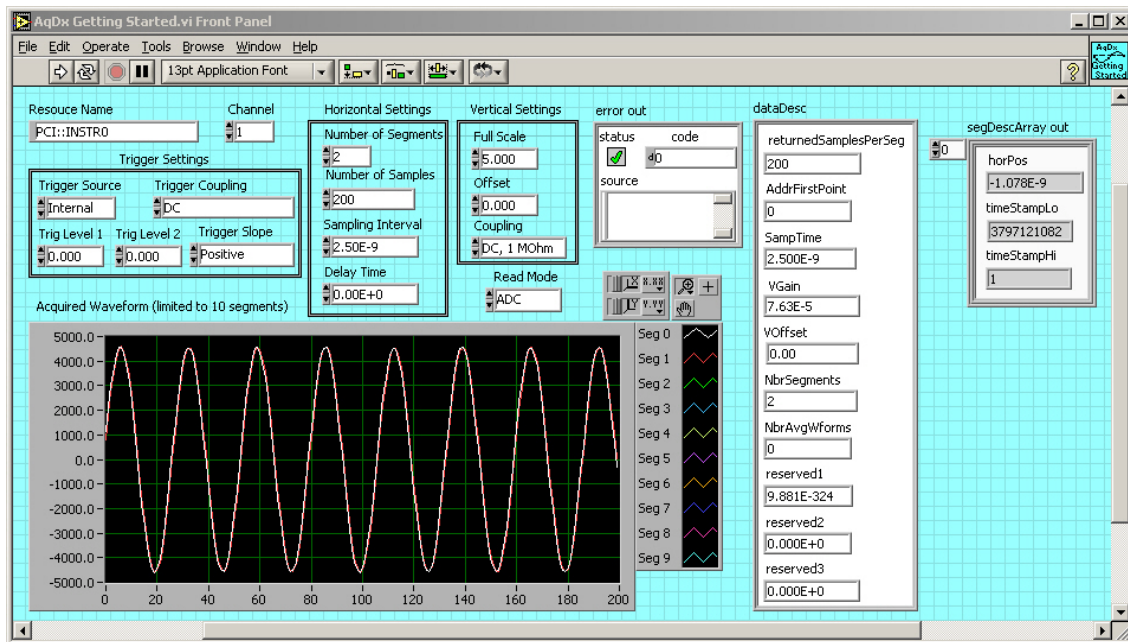
This VI demonstrates how to use some of the basic components of the AqDx Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires one waveform. Note that the front panel controls should be set to their desired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

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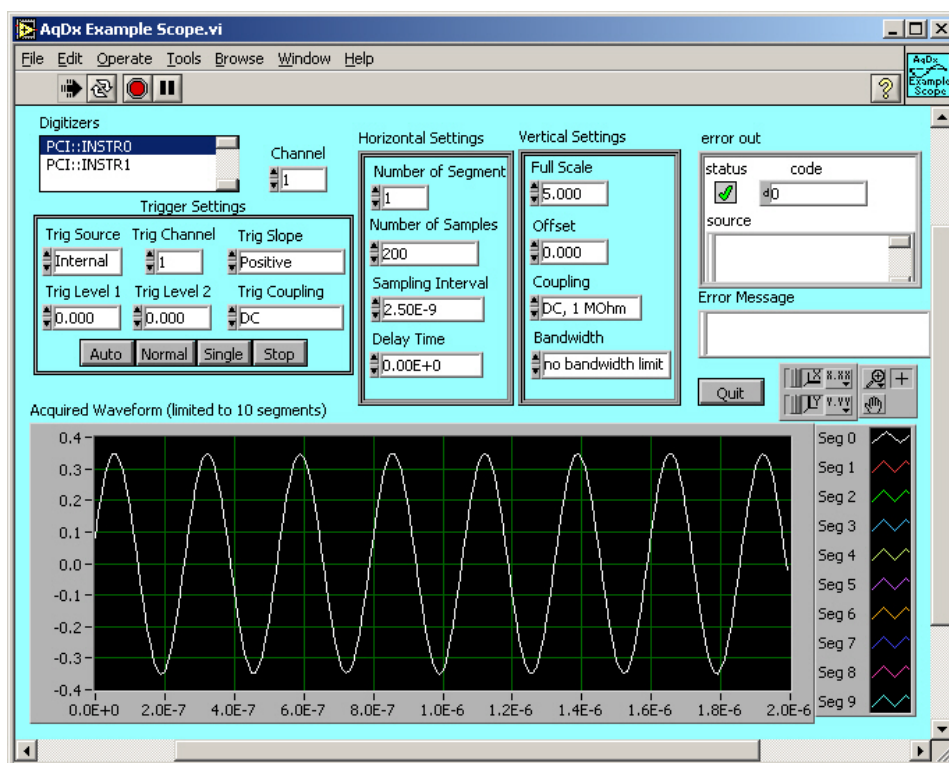
This VI presents a basic, interactive virtual oscilloscope using the AqDx Digitizer Driver. Not all the functionality of the Acqiris digitizers is supported in this program, but most of the most-commonly-used functions are demonstrated in it.

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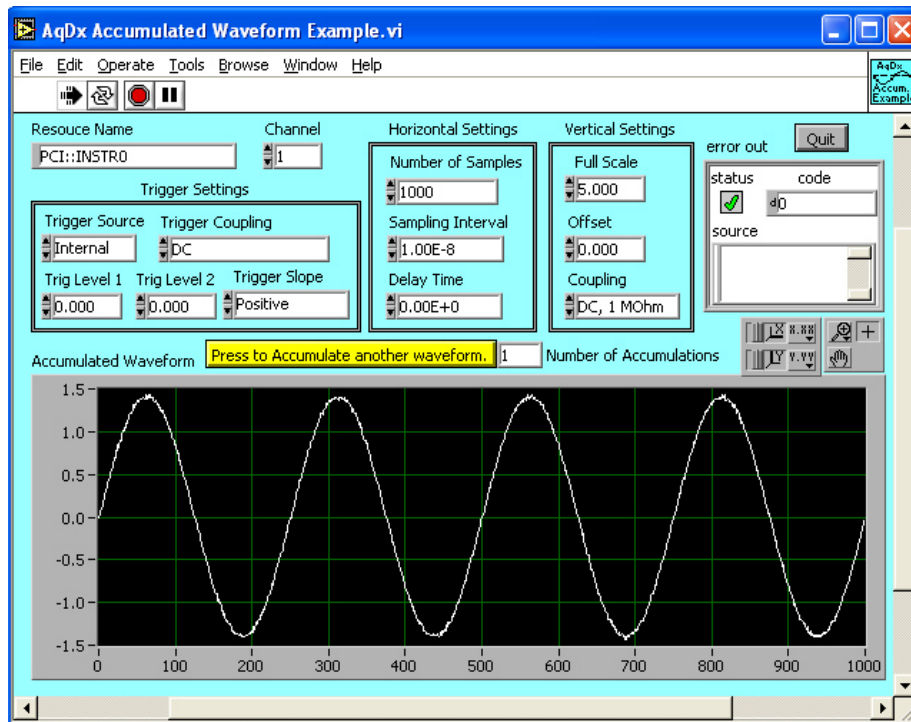
This VI demonstrates how to use the Accumulate Waveform function of the Acqiris Digitizer Driver. It finds and initializes a digitizer, sets the basic parameters according to the controls on the front panel, and then acquires and accumulates waveforms as the user requests. Note that the front panel controls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

2.3.5. AqTx TC890 VI
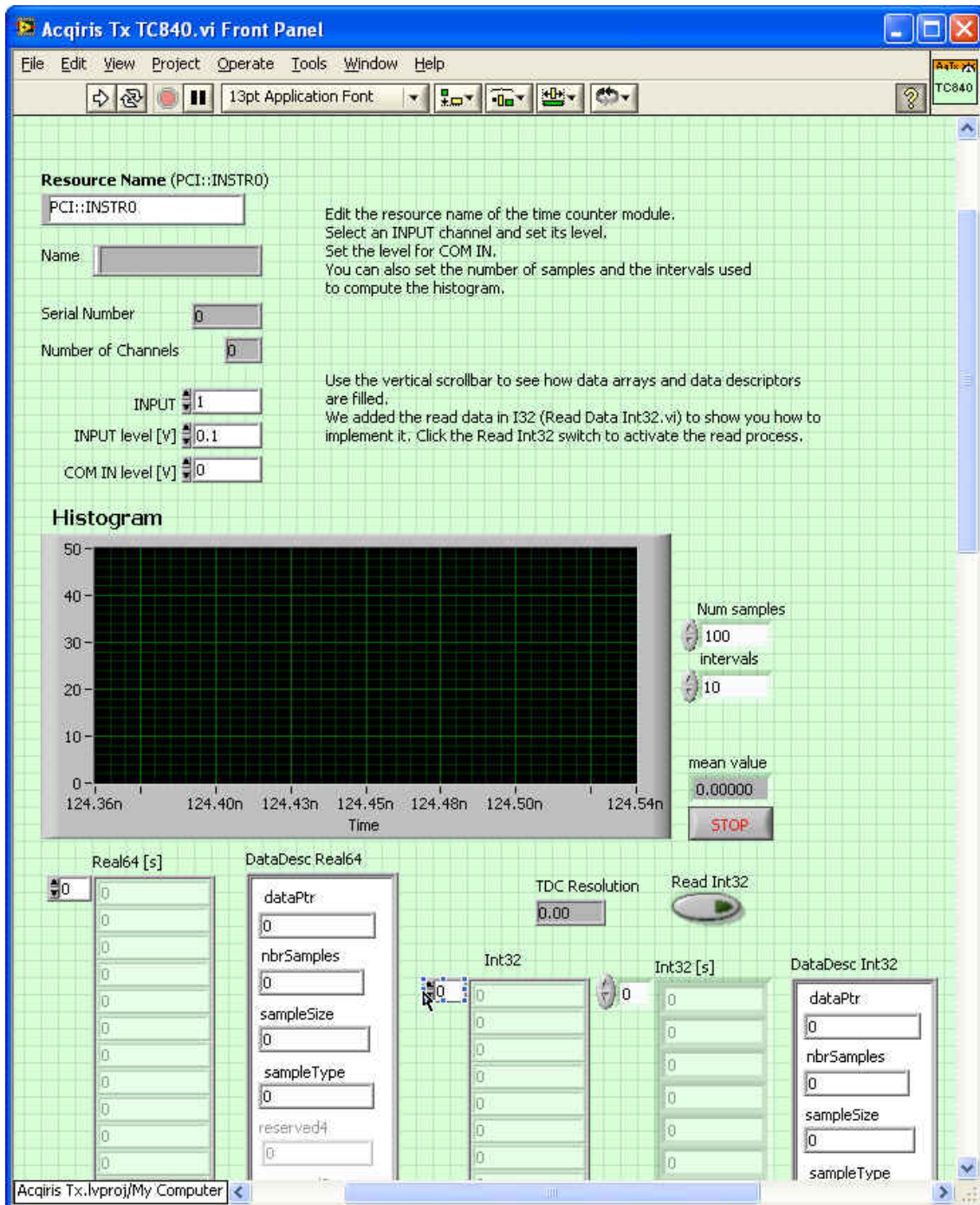
This VI demonstrates how to use a TC840 Time Counter. It finds and initializes an instrument, sets the basic parameters according to the controls on the front panel, and then acquires and accumulates data as the user requests. Note that the front panel controls should be set to their desired values before the VI is run.

2.3.6. AqTx TC890 VI
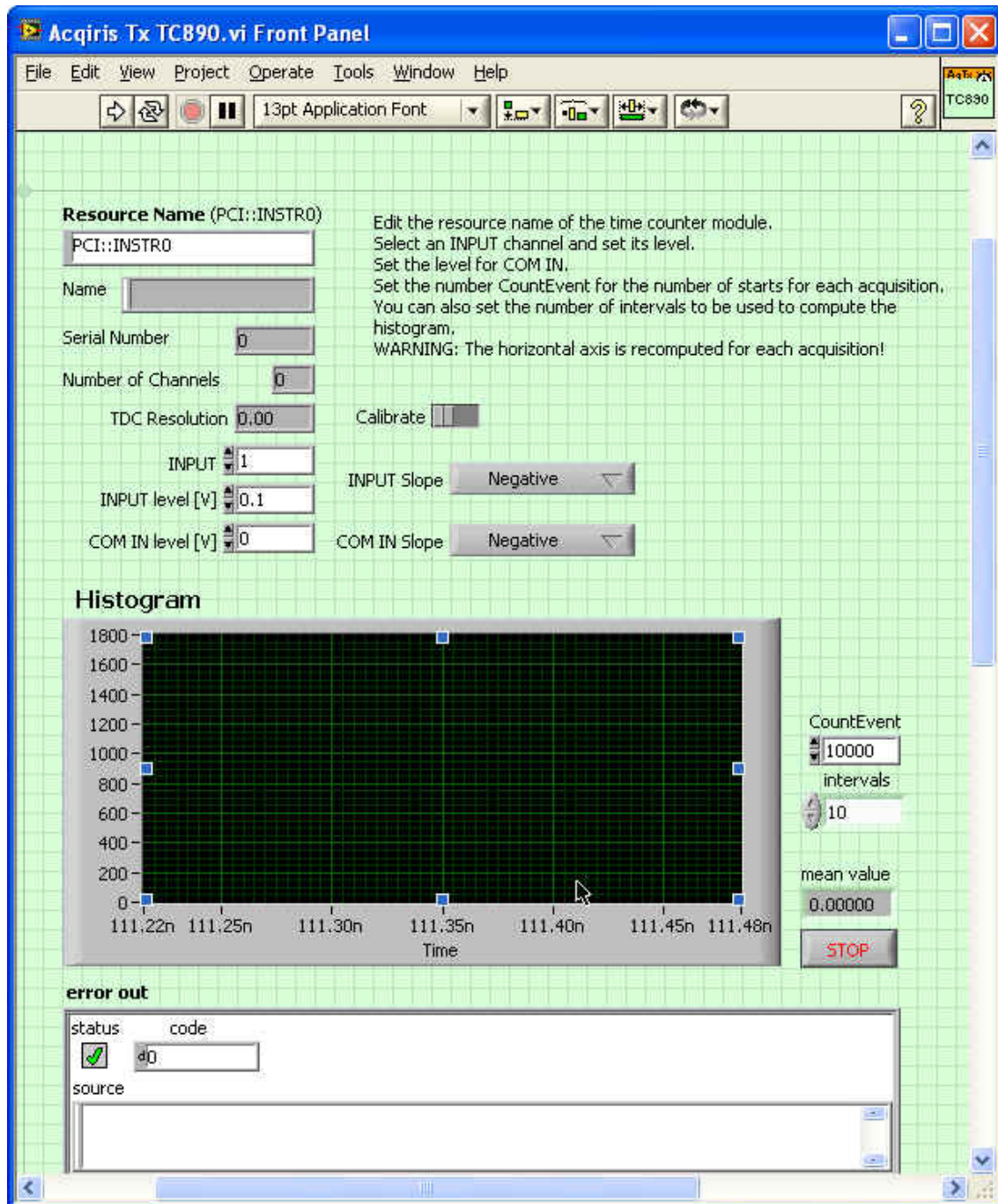
This VI demonstrates how to use a TC890 Time Counter. It finds and initializes an instrument, sets the basic parameters according to the controls on the front panel, and then acquires and accumulates data on the common and the chosen channel as the user requests. Note that the front panel controls should be set to their desired values before the VI is run.

2.4. Visual Basic

Visual Basic support is available for Versions 5.0 or 6.0 with sample programs in the VB directory. Basic .NET support (AcqrsD1Interface.vb) is also available but there are no sample programs yet.

The Visual Basic sample program comes in 8-bit and 12-bit versions. The 12-bit programs can also be used for the 10-bit digitizers. They each consist of 3 major parts:

1. The **AcqrsD1Interface.bas** file that contains the Visual Basic version of the interface to the Acqiris driver dll's (installed in the System directory). This file is needed for all Visual Basic projects involving Acqiris digitizers.

2. The **AcqirisShow8.frm** (or **AcqirisShow12.frm**) file that contains all the basic functionality of the sample program, such as initialization, acquisition control and display.

3. The **DevCtrlForm8.frm** (or **DevCtrlForm12.frm**) file with the code for a dialog box for the control of a complete digitizer.

The Visual Basic sample program is capable of managing several Acqiris digitizers attached to the computer and of displaying one channel of one digitizer at a time. If no digitizer exists on the computer, it initializes 3 simulated digitizers (in **Sub Form_Initialize()**). While it has enough functionality to permit a fairly complete operation of a digitizer, many possible features were left out in order to keep the program simple to understand.

In order to run the Visual Basic sample program, you must have installed the Visual Basic Version 5.0 (or 6.0 but not .NET):

- Start Visual Basic

- Use the menu **File → Open Project…** to point to the **AcqirisShow1.vbp** project file in the AcqirisLive\VB directory and choose **Open**

- After the project is loaded, use the menu **Run → Start** or **F5** to start executing the project. A waveform display window should pop up, displaying a real (or simulated) waveform.

- Use the button **Digitizer Control** to make a dialog box appear. The top-most control **Digitizer#** permits the choice of one of several digitizers (and to check how many were found). If no physical digitizer was found, the program initializes 3 simulated digitizers, a DP110, DC240 and a DC110 (with the 2MB option). You can change the simulated digitizers in **Sub Form_Initialize()**.

- The program is stopped by clicking on the **Close** button (**x**) on the upper right hand side of the waveform display window.

The waveform display window can be resized, but the dialog box cannot.

The acquisition mode **Norm** really is another **Auto** mode, with a slightly longer timeout period. A real **Norm** mode without timeout would require a way for the user to regain control in situations where there is no trigger.

Simulated digitizers have fixed simulated input signals (sine waves, triangle waves or square waves) that cannot be modified through the supplied API.

Note: The programming advice in the rest of this manual is given for the C language interface. However, it is equally valid for Visual Basic. Refer to the file **AcqrsD1Interface.bas** for the correspondence between the Visual Basic and C-language names of the Acqiris driver functions.

Note: There is no VB support for Time Counter modules.

2.5. MATLAB

MATLAB is a very powerful environment to analyse and display data. 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.

2.6. Wind River VxWorks (Tornado)

The VxWorks sample program is written for the Tornado environment. The **GetStartedVxW.cpp** file contains a simple user program which:

- spawns a process with a large stack as needed by the Acqiris driver

- finds the Acqiris digitizers on the target machine

- initializes the first (or only) one

- configures some acquisition parameters (and rereads them for checking)
- loops (100 times) over a cycle that
    - starts the acquisition
    - waits for it to terminate
    - reads the waveform

2.6.1. Compiling

This file can be compiled 'as is' within the **GetStartedVxW.wsp** Tornado workspace. However, please make sure to run an update to the project dependencies. You may need to change the directory paths for the Tornado include files and the Acqiris include files if you move the Tornado directory or use a different development environment. In the latter case you must verify that the compilation flags

–D_VXWORKS –D_ACQIRIS are present.

2.6.2. Loading

First download the Low-Level (kernel-mode) VxWorks Device Driver file **VxWorksDriverPCI.out** to the target machine, followed by the VxWorks Driver file **AcqirisVxWUMode.out**.

Finally, you should download the VxWorks application **GetStartedVxW.out**.

2.6.3. Running the Program

It is recommended that you follow the code with the debugger, since it is the only way to see the response of the function calls immediately.

2.7. Linux

The AcqirisDemo program can simply be run from the AcqirisLinux directory by entering:

Demo/AcqirisDemo

The library /usr/lib/libAqDrv4.so is compiled with either gcc 3.3, 3.4, or 4.1; you need to have installed the desired version.

The Linux **GetStarted** sample program is provided ready to run. It can also be created without the need of an additional development application.

- Enter the command "cd **usr/src**"

- Run **make** to compile and link the program

- Note that the include and library paths are defined in the **Makefile**.

The **GetStarted.cpp** file contains a simple user program which:

- has globally allocated buffers to achieve optimal readout performance

- finds the Acqiris digitizers on the target machine

- initializes the first (or only) one

- configures some acquisition parameters (and rereads them for checking)

- loops (100 times) over a cycle that
    - starts the acquisition
    - waits for it to terminate
    - reads the waveform

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 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. Only one process on a machine can have loaded the Acqiris driver at any given moment.

Most of the Acqrs and AcqrsXX driver functions are reentrant. After initialization, they are protected against multiple calls from different threads. The unprotected routines are :

Acqrs_init	**Acqrs_initWithOptions**	**Acqrs_getNbrInstruments**
Acqrs_close	**Acqrs_closeAll**	
AcqrsD1_multiInstrAutoDefine	**AcqrsD1_multiInstrDefine**	**AcqrsD1_multiInstrUndefineAll**

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: **AcqirisInterface.h, AcqirisD1Interface.h,** or **AcqirisT3Interface.h** for C-like environements, or the equivalent **.bas** or **.vb** files for Visual Basic.

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 ASBus, 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 ASBus and return the total number of instruments found, including individual modules without ASBus connections. It automatically searches for all sets of modules that are connected with ASBus, 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**.

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]));
}
```

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 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:

```
ViSession instrumentID;
Acqrs_InitWithOptions("PCI::SER10047", VI_FALSE, VI_FALSE, "",
                    &instrumentID);
```

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:

```
ViSession instrumentID;
Acqrs_InitWithOptions("PCI::BUS02::SLOT06", VI_FALSE, VI_FALSE, "",
                    &instrumentID);
```

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);
```

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<bus>::<device>::INSTR"
"PXI0::<bus>-<device>::INSTR"
"PXI0::CHASSIS<chassis>::SLOT<slot>::INSTR"

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

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 inititalization of a module needing an FPGA file, the desired file will be searched for in the working directory of the application.

- Then the working directory will be searched for a file "AqDrv4.ini"

- Finally the directory pointed to by the environment variable "AcqirisDxDir" will be searched for a file "AqDrv4.ini"

- The "AqDrv4.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 **AcqrsD1_configMode** is used to change functionality.

Additional VxWorks Instructions

For VxWorks users, the normal mechanism for finding the FPGA .bit files will not work; the driver has to be told explicitly where to find them. This procedure is also shown in the **GetStarted.cpp** VxWorks sample program. Thus, **Acqrs_InitWithOptions** has to be called with

```
ViString options = "cal=0";
status = Acqrs_InitWithOptions(resourceName, VI_FALSE, VI_FALSE, options,
                     &(instrumentID[i]));
```

Then, before using the desired module in any mode, you should execute code like that shown below:

```
ViString FPGADirectoryName = "C:\firmware"; // or "C:\" for ETS
Acqrs_configLogicDevice(instrumentID, NULL,
                               FPGADirectoryName, 2);
```

As a final step you should now calibrate the instrument (which will cause the FPGA files to be loaded):

```
Status = Acqrs_calibrate(instrumentID);
```

3.2.7. Automatic Definition of MultiInstruments

The function **AcqrsD1_multiInstrumentAutoDefine** automatically searches for all sets of modules that are connected with ASBus, and configures each such block as a single *MultiInstrument*. It then reports the total number

of instruments found, including individual modules without ASBus 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(&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]));
}
```

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 or XP. 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 3.17, **ASBus Operation**.

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

ASBus

- 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.

ASBus^2 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*.

ASBus^2 3U 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 ASBus. It cannot be used for ASBus^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 later
status = Acqrs_getNbrInstruments(&nbrInstruments);
```

```
    // Initialize the digitizers
    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]));
    }
    // 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);
```

The first part of the code above finds and initializes all individual digitizers, as shown in section 3.2.1, **PCI & VXI Identification** by Order Found. 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 ASBus 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 3.17, **ASBus Operation**.

3.2.9. AqGeo.map file positioning

For ASBus 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 ASBus 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 "AqDrv4.ini"

- Finally the directory pointed to by the environment variable "AcqirisDxDir" will be searched for a file "AqDrv4.ini"

- The "AqDrv4.ini" file should contain the name of a directory which will also be searched for the AqGeo.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. Any module 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:

```
ViSession instrumentID;
ViStatus status;
status = Acqrs_setSimulationOptions("M2M");
// Initialize the instrument
status = Acqrs_InitWithOptions("PCI::DC110", VI_FALSE, VI_FALSE,
                    "simulate=TRUE", &(instrumentID));
```

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 ("")**.

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]);
    // or status = AcqrsT3_stopAcquisition(instrumentID[i]);
}
Acqrs_closeAll();
```

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.

3.3. 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 !
AcqrsD1_configHorizontal(instrID, sampInterval, delayTime);
AcqrsD1_configMemory(instrID, nbrSamples, nbrSegments);
AcqrsD1_configVertical(instrID, channel, fullScale, offset, coupling,
                       bandwidth);
AcqrsD1_configTrigClass(instrID, 0, 0x00000001, 0, 0, 0.0, 0.0);
AcqrsD1_configTrigSource(instrID, channel, trigCoupling, trigSlope,
                         trigLevel, 0.0);
```

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 & DP1400** instruments offer additional functionality through **AcqrsD1_configMemoryEx**. This includes the Simultaneous Multibuffer Acquisition and Readout (SMAR) mode to allow a higher loss-less trigger rate. For 10-bit 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 3.12, *TRIGGER DELAY AND HORIZONTAL WAVEFORM 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 module, or an AP240/AP235 signal analyzer platform, add a call to **AcqrsD1_configVertical** with channel = -1 before the call to **AcqrsD1_configTrigSource**.

- The DP1400 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.

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 ± 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;

AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "NbrSamples", &nbrSamples);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "NbrWaveforms", &nbrWaveForms);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "DitherRange", &ditherRange);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "TrigResync", &trigResync);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "StartDelay", &startDelay);
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "StopDelay", &stopDelay);
```

Comments:

- The value *channelNbr* is usually 0. However, for AP240/AP235 Signal Analyzer platforms 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" (in future software versions, the parameter *sampInterval* of the function **AcqrsD1_configHorizontal** is likely to be replaced by "SampInterval")

  - *nbrSamples* and *nbrSegments* of the function **AcqrsD1_configMemory** are replaced by "NbrSamples" and "NbrSegments"

- 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**.

- The value *startDelay* controls the time between the trigger and the first data sample that is to be added to the averager sum. It is expressed in samples and must be an integer multiple 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.

- The *stopDelay* permits the addition of an extra delay to the dead time between the averaging of subsequent waveforms. Its minimum value may be zero. It also is expressed in samples and must be an integer multiple of the 'averaging block size'.

- The *ditherRange* value may be between 0 (no dithering) and 15 (max dithering). Please refer to the next section for further explanations.

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

```
AcqrsD1_configAvgConfig(instrID, channelNbr,
                        "DitherRange", N);
```

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.

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**

- 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:

```
const long channelNbr = 0;
double fsr, offset;
long coupl, bwidth, reSync, freeRun = 2;

// Make an acquisition, at the current conditions, but with
// "Grnd" coupling and free running trigger.
AcqrsD1_getVertical(instrID, 1, &fsr, &offset, &coupl, &bwidth);
AcqrsD1_configVertical(instrID, 1, fsr, offset, 0, bwidth);

AcqrsD1_getAvgConfig(instrID, channelNbr, "TrigResync", &reSync);
AcqrsD1_configAvgConfig(instrID, channelNbr, "TrigResync", &freeRun);

AcqrsD1_acquire(instrID);
long timeOut = 1000;         // depends on conditions!
AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);
```

```
      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];
      AcqrsD1_readRealWform(instrID, 1, 0, 0, nbrPoints, bckGndWform,
      &nbrReturnedSamples, &horPos, &sampTime, &timeStampLo, &timeStampHi);


      // Restore the settings of the averager
      AcqrsD1_configVertical(instrID, 1, fsr, offset, coupl, bwidth);
      AcqrsD1_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:

```
      const long channelNbr = 0;
      double fsr, offset;
      long coupl, bwidth, reSync, freeRun = 2;

      AcqrsD1_acquire(instrID);
      long timeOut = 1000;          // depends on conditions!
      AcqrsD1_waitForEndOfAcquisition(instrID, timeOut);


      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];
      AcqrsD1_readRealWform(instrID, 1, 0, 0, nbrPoints, bckGndWform,
      &nbrReturnedSamples, &horPos, &sampTime, &timeStampLo, &timeStampHi);
```

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

```
      long bckGndWform[nbrPoints];             // Background as 32-bit sum
      AcqrsD1_readData(instrID, channel, &readParams, waveformArray, &wfDesc,
                       &segDesc);
```

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. Here is an example:

```
const long channelNbr = 0;
double fsr,offset, threshold, base;

threshold = - offset; // place the threshold at the middle of the screen
base = threshold - fsr/10.; // place the base one division below the
                          threshold
// set the base and threshold voltage values
AcqrsD1_configAvgConfig(instrID, channelNbr, "Threshold", &threshold);
AcqrsD1_configAvgConfig(instrID, channelNbr, "NoiseBase", &base);

// enable the NSA functionality
AcqrsD1_configAvgConfig(instrID, channelNbr, "ThresholdEnable",1);
AcqrsD1_configAvgConfig(instrID, channelNbr, "NoiseBaseEnable",1);
```

3.5. Configuring SSR Analyzers

3.5.1. Acquisition Parameters

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

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

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

The *AutoSwitch* 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 argumenta to **AcqrsD1_configMode** must always be 0.

- When in *AutoSwitch* 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 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. 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;
g_lastGate[channel] = 1; // a very simple example
for(int g=0;g<g_lastGate[channel];g++)
{
 // the first gate starts with the first point
 configSetupData[g].GatePos           = g * gateSize;
 configSetupData[g].GateLength  = gateSize;
 g_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.

```
long channel = 1, gate =2;
AcqrsD1_configAvgConfig(instrID, channel, "GateType", &gate);
long preSamples = 0, postSamples = 0, maxGates = 1;
double threshold = 0.0; // in Volts
long thresEnable = 1;
status = AcqrsD1_configAvgConfig(instrID, channel,
                        "PreSamples", &preSamples);
status = AcqrsD1_configAvgConfig(instrID, channel,
                        "PostSamples",&postSamples);
status = AcqrsD1_configAvgConfig(instrID, channel,
                        "Threshold",  &threshold);
status = AcqrsD1_configAvgConfig(instrID, channel,
                        "NbrMaxGates",&maxGates);
```

3.5.3. 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;
Acqrs_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 using 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. Configuring 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_configAvgConfig(instrID, 0, " TdcHistogramDepth", &tdcDepth);
AcqrsD1_configAvgConfig(instrID, 0, " TdcHistogramIncrement", &tdcIncr);
AcqrsD1_configAvgConfig(instrID, 0, " TdcProcessType", &tdcType);
// if the acquisition has segments to be histogrammed independently
long tdcOverlay = 0;
AcqrsD1_configAvgConfig(instrID, 0, " TdcOverlaySegments", &tdcOverlay);
```

3.7. Configuring AP101/AP201 Analyzers

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

```
AcqrsD1_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.

3.8. Configuring TC8xx Time Counters

Use the following short code fragment for a device configuration:

```
// Configure
long i, n_chan, slope = 0;
double trigLevel = 0.2;
double timout = 0.1;
Acqrs_getNbrChannels(instrumentID, &n_chan);
for( i=0; i< n_chan; i++)
  AcqrsT3_configChannel(instrumentID, i+1, slope, trigLevel, 0);
AcqrsT3_configChannel(instrumentID, -1, trigSlope, trigLevel, 0);
AcqrsT3_configAcqConditions (instrumentID, timeout, 0, 0);
```

Comments:

- Channel numbers run from 1 to nbrChannels, not from 0! The common channel uses the number –1.

- The configuration functions are protected against illegal or incoherent values. Thus, the system might adapt the values you ask for. The 'query' counterpart of this function, **AcqrsT3_getChannel,** can be interrogated to learn what the driver did to the values.

- The granularity of a threshold value setting is limited by the hardware that uses a 12-bit DAC covering the desired range.

- The timeout for the acquisition must be chosen to allow the maximum reasonable value.

3.9. 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.

3.9.1. Starting an Acquisition

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

```
AcqrsD1_acquire(instrID); // start the acquisition
```

or the following line of code for starting an acquisition on a Time Counter:

```
AcqrsT3_acquire(instrumentID); // start the acquisition
```

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

3.9.2. 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):

```
Acqrs_getInstrumentInfo(instrID,"IsPreTriggerRunning",&PreTrigger);
```

3.9.3. 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
 //AcqrsT3_acqDone(instrID, &done); // the Time Counter equivalent

if (timeoutCounter <= 0) // timeout, stop acquisition
    STOP ACQUISITION
```

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.

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

```
ViBoolean done = 0;
long timeoutCounter = 100;
while ((!done) && (--timeoutCounter > 0))
{
    AcqrsD1_acqDone(instrID, &done); // poll for status or
//AcqrsT3_acqDone(instrID, &done); // the Time Counter equivalent
    Sleep(1);
}
if (timeoutCounter <= 0) // timeout, stop acquisition
     STOP ACQUISITION
```

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.

The equivalent of the above for a Time Counter would be:

```
ViStatus status;
long timeOut = 100; // in ms
status = AcqrsT3_waitForEndOfAcquisition(instrumentID, timeOut);
if (status == ACQIRIS_ERROR_ACQ_TIMEOUT) // timeout, stop
    status = AcqrsT3_stopAcquisition(instrumentID);
```

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.

3.9.4. 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:

```
{
    AcqrsD1_forceTrig(instrID);
    if (AcqrsD1_waitForEndOfAcquisition(instrID, timeOut) ==
        ACQIRIS_ERROR_ACQ_TIMEOUT)
    {
        AcqrsD1_stopAcquisition(instrID);
        SCREAM, because a major error occurred
    }
}
```

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.

3.9.5. Analyzer and Peak$^{\text{TDC}}$ Autoswitch mode

If with the **Peak$^{\text{TDC}}$** 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.

For all other cases the AP Analyzers implement an *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.

3.9.5.1. Sequence of actions for Autoswitch with event readout

The *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*.
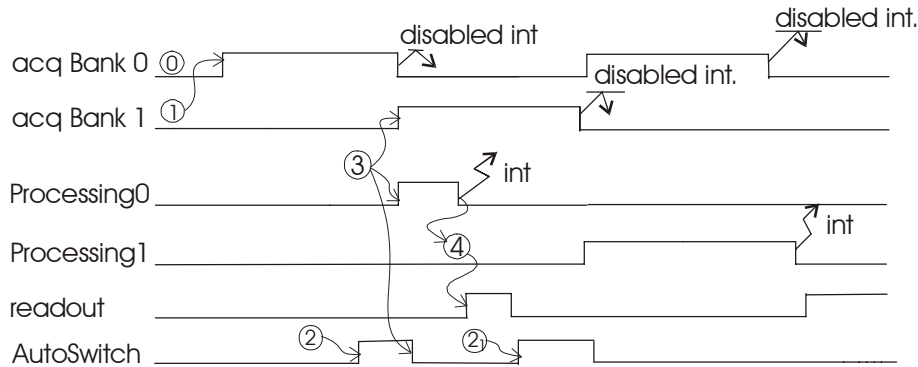
S	Description	Software implementation
0	The software configures the mode, the acquisition parameters, and the readout.	**AcqrsD1_configMode**(…); **AcqrsD1_configAvgConfig**(…); …
1	The software starts the first acquisition.	**AcqrsD1_acquire**(instrID);
2	The software sets the *AutoSwitch* semaphore. 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 finish, the FPGA automatically switches the banks, starts a new acquisition, a new processing and clears the *AutoSwitch* 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:

```
    status=AcqrsD1_acquire(instrID); // Start the acquisition
    processType = 0;


    for (;;)  //loop forever
    {
        status=AcqrsD1_processData(instrID, processType, 1);
        status=AcqrsD1_waitForEndOfProcessing(instrID,timeout);
        status=AcqrsD1_readData(instrID, channel, &readPar, &adcArray,
                        &dataDesc, &segDesc);
    }
```

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.

3.9.5.2.   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.

3.9.5.3.   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

```
    AcqrsD1_processData(instrID, processType, 2); // do a bank switch but do not
                        start
    status = AcqrsD1_waitForEndOfProcessing(instrID, timeout); // usual wait
    // finish reading the data associated with the old settings
    AcqrsD1_readData(instrID, channel, &readParams, waveformArray, &wfDesc,
                        &segDesc);
```

- 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); …
```

3.10.   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:

AcqrsD1_readCharWform

AcqrsD1_readCharSequence   **AcqrsD1_readRealWform**

AcqrsD1_readRealSequence   **AcqrsD1_accumulateWform**

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 3.10.6 **Reading SSR Analyzer Waveforms** or 3.10.7 **Reading AP101/AP201 Analyzer Waveforms**.

3.10.1. 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];

AqReadParameters      *readPar   = new AqReadParameters;
AqDataDescriptor      *dataDesc  = new AqDataDescriptor;
AqSegmentDescriptor   *segDesc   = new AqSegmentDescriptor[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->segDescArraySize = sizeof(AqSegmentDescriptor)*nbrSegments;
readPar->flags = 0;
readPar->reserved = 0;
readPar->reserved2 = 0.0;
readPar->reserved3 = 0.0;

status = AcqrsD1_readData(instrID, channel, readPar, dataArray ,
                   dataDesc, segDesc);
```

Comments:

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

- 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 3.13, *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 3.14, *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.

3.10.2. 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 $\geq$ *segmentOffset * (nbrSegments+1) * (dataType + 1)*

with

arraySize $\geq$ *(nbrSamplesNom + currentSegmentPad) * (nbrSegments+1) * (dataType + 1)*

and

segmentOffset$\geq$, nbrSamplesInSeg

where

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

```
Acqrs_getInstrumentInfo(instrID,"TbSegmentPad",
                                    &currentSegmentPad);
```

- 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+1)*

with

segmentOffset$\geq$, nbrSamplesInSeg

and

arraySize $\geq$ 8 * *segmentOffset + (nbrSamplesNom + currentSegmentPad) * nbrSegments * (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.

```
long nbrSegments   = 10;
long nbrPoints = 1000;
char *dataArrayP;
long currentSegmentPad;
long nbrSamplesNom,nbrSegmentsNom;

AqReadParameters      *readPar    = new AqReadParameters;
AqDataDescriptor      *dataDesc   = new AqDataDescriptor;
AqSegmentDescriptor   *segDesc    = new AqSegmentDescriptor[nbrSegments];
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 = Acqrs_getInstrumentInfo
                    (instrID,"TbSegmentPad",&currentSegmentPad);
// in this case the next call doesn't have any surprises
status = AcqrsD1_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.

3.10.3. 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 3.13, **Horizontal Parameters in Acquired Waveforms**) is not taken into account.

3.10.4. 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**.

3.10.4.1. 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:

```
AqReadParameters       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:

```
sum[i] = (volts[i] + offset + FS/2.0) * 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,    nbrSamples,
                      waveformArray,&returnedSamples,      &horPos,
                      &sampTime, &tStampLo, &tStampHi);
```

3.10.4.2. Averaged Waveforms as 32-bit Sums

You must use the general-purpose function **AcqrsD1_readData**.

Use this code fragment:

```
AqReadParameters   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;

AcqrsD1_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).

3.10.5. 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.

3.10.6. Reading SSR Analyzer Waveforms

3.10.6.1. 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.

3.10.6.2. 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.

3.10.6.3. 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 timestamps 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

31..24 (8 bits)	23..0 (24 bits)
Flag = 00000100 (0x04)	Time Stamp MSB
TimeStamp LSB	

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

Gate block

31..24 (8 bits)	23..0 (24 bits)		
Flag = 00000000	Gate position from the origin of the acquisition (not the segment!)		
31..0			
Gate length (number of Data bytes, always a multiple of 4)			
31..24	23..16	15..8	7..0
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.

3.10.6.4. 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.

3.10.7. Reading Peak^{TDC} Analyzer Data and Histograms

3.10.7.1. Reading the gated data

The gated data of the current event can be read out as described in 3.10.6.3 **Gated data**. This is the only output format that gives access to the segment time stamps.

3.10.7.2. 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:

Peak region block   8 points

31..24 (8 bits)	23..16 (8 bits)	15..8 (8 bits)	7..0 (8 bits)
Flag 00010010	0x00	Valid Left	Valid Right
31..0 (32 bits)			
Peak Position (= tmax)			
31..24 (8 bits)	23..16 (8 bits)	15..8 (8 bits)	7..0 (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

31..24 (8 bits)	23..16 (8 bits)	15..8 (8 bits)	7..0 (8 bits)
Flag 00010001	0x00	Valid Left	Valid Right
31..0 (32 bits)			
Peak Position (= tmax)			
31..24 (8 bits)	23..16 (8 bits)	15..8 (8 bits)	7..0 (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.

3.10.7.3. 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

31..24 (8 bits)	23	19..4 (16 bits)	3..0 (4 bits)

Flag 00010000 (0x10)	Unused	Peak amplitude with ADC resolution	Interpolated fractional part of amplitude (1/16 LSB)
29..4 (26 bits)			3..0 (4 bits)
Peak Position with sample rate resolution			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 substraction.

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.

3.10.7.4. 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 contains accumulated sums of either 2**32, as ViUInt32, or 2**16, as ViUInt16.

Histogram bin is 32 bits wide

31..0
Bin0
…

Histogram bin is 16 bits wide

31..16	15..0
Bin 1	Bin 0
…	…

3.10.8. Reading AP101/AP201 Analyzer Waveforms

3.10.8.1. 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 5.14, **Data Readout,** in the Standard Manual.

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 *autoswitch* 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.

With the function **AcqrsD1_readData**, use this code fragment:

```
AqReadParameters     readParams; // Read Definitions
AqDataDescriptor     dataDesc;   // Returned waveform values
AqSegmentDescriptor  segDesc;         // Returned segment values

long channel = 1, nbrSamples = 4000000;
char waveformArray[4000000];
readParams.dataType = ReadInt8;
readParams.readMode = ReadModeStdW;
readParams.nbrSegments = 1;
readParams.firstSampleInSeg = 0;
readParams.segmentOffset = nbrSamples;
readParams.firstSegment = 0;              // Read first segment
readParams.nbrSamplesInSeg = nbrSamples;
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);
AcqrsD1_acquire(instrID);         //essential!!


AcqrsD1_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.

3.10.8.2. 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:

```
const int NbrGates = 64;
long channelNbr = 0;
long configObj = AvgGate;
long lastGate = NbrGates;
AqGateParameters   gatePara[NbrGates];
for(int i=0;i<NbrGates;i++)
{
    gatePara[i].GateLength = 256;
    gatePara[i].GatePos = i*gatePara[i].GateLength;
}
AcqrsD1_configSetupArray(instrID, channelNbr, configObj, NbrGates,
                    gatePara);
```

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:

```
const int NbrGates = 64;
long channelNbr = 0;
long configObj = AvgGate;
long lastGate;
AqGateParameters    gatePara[NbrGates];

AcqrsD1_getSetupArray (instrID, channelNbr, configObj, NbrGates,
    gatePara, &lastGate);
```

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**.

```
AqReadParameters     readParams; // Read Definitions
AqDataDescriptor     dataDesc;   // Returned waveform values
AqSegmentDescriptor  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.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.

3.10.8.3. 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:

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:

```
AqReadParameters        readParams;    // Read Definitions
AqDataDescriptor     dataDesc;         // Returned waveform values
long channel = 1, segmentNumber = 0;
long nbrPeaks = 2 * nbrGates;     // nbrGates is defined by user
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);
AcqrsD1_acquire(instrID);          // Acquire into bank 0
AcqrsD1_waitForEndOfAcquisition(instrID, timeout);
// At this point, you should check the return value!
```

```
bool finished = false;
while(!finished)
{
    memoryBank = (memoryBank + 1)&0x1;// switch to other bank
    AcqrsD1_configMode(instrID, 3, 0, memoryBank);
    AcqrsD1_acquire(instrID);    // start new acquisition
    AcqrsD1_processData(instrID, 0, 0);// start processing
    AcqrsD1_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;

    AcqrsD1_readData(instrID, channel, &readParams, waveformArray,
                    &dataDesc, NULL);
    //...analyse and store data

    AcqrsD1_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];
```

3.11. T3-style Data Readout

For the reading of timer data the **AcqrsT3_readData** routine should be used.

Control of the read parameters is passed through the input structure **AqT3ReadParameters**. The output format is determined by these values as well as the type of module in use.

TC890 TOF modules use a direct read out of the modules data for maximum throughput. The PC does not have to analyze the data coming from the module. The data are packed into 4-byte timer hits for each signal.

TC840 modules have a two phase readout. The Acqiris driver software in the PC analyzes the raw data as read from the modules to derive the final timer results. These are presented in double floating point format or an integer count of the time in the specified time granularity of the module.

The TC890 data are coded on 32 bits as described below. This includes:

- Event on the channel 1 to 6, time relative to the common channel event
- Event on the common input, each common event are numbered, even the common event that are lost if the internal Buffer is Full.
- Aux IO Event Marker.
- Switch marker.

[31]	Overflow	Indicates the time value is not good because the real-time counter reached its maximum value. When the data is a marker this bit is also set '1'. if [30..28] = 0 to 6 => channel overflow if [30..28] = 7 => marker
[30..28]	DataType	Type of the current data, Common channel, Channel number or Marker 0 for common (or channel 0) 1-6 for channel 1-6 7 denote the data is a Marker
[27..0]	Time	Channel 0: The value plus one is the number of hit of the common channel Channel 1 to 6: The value is the time between the channel and the common. Marker: Value=0 Switch marker: Switch from Auxiliary input A Value=1 Switch marker: Common channel Event count. Value=2 Switch marker: Memory Full. Value=16 Marker: Auxiliary input B marker.

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
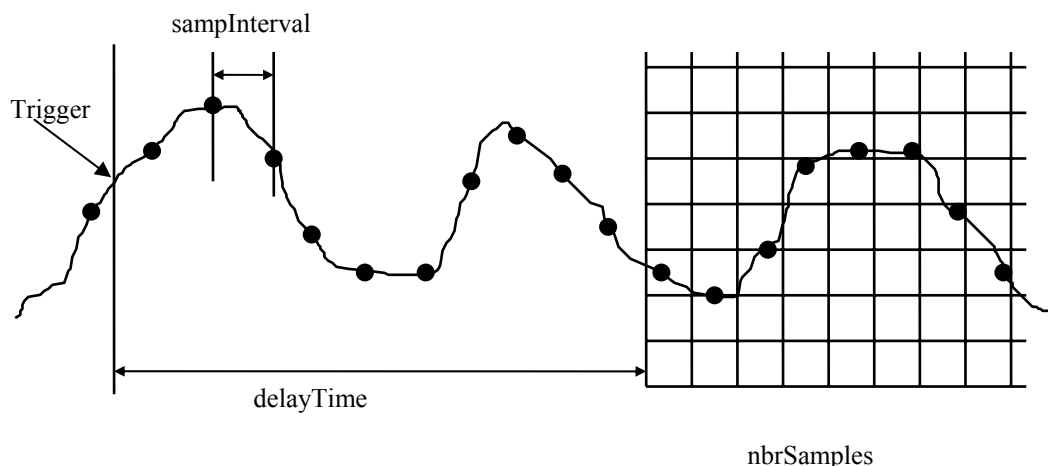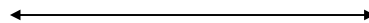
By convention, the nominal trigger delay is taken relative to the beginning of the trace, i.e. relative to the left edge of a real or virtual display grid. It can be interpreted as the time from the trigger to the start of waveform recording. If this number is positive, recording starts *after* the trigger (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.

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:
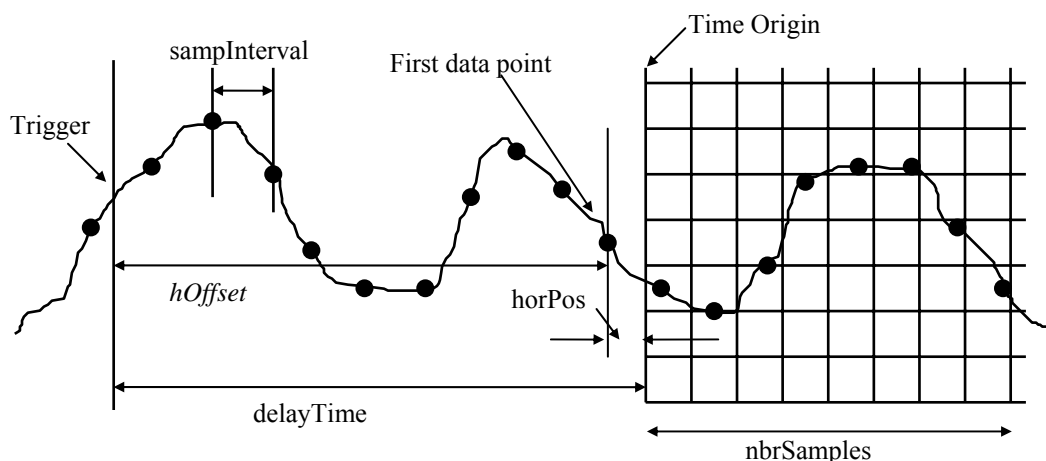
- The value of **delayTime** positions exactly the left edge of the display (or the exact nominal beginning of the waveform) with respect to the "stable" trigger time, which is the real reference point. We define the time 'trigger time + delayTime' as the time origin for the waveform, which is equivalent 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:

$$x[i] \quad = \quad horPos + i * sampInterval$$

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. Timestamps

The 10-bit-Family of digitizers implements a timestamp to measure the time of the trigger for each acquisition segment. These timestamps 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 'timestamp' in order to measure the time between the triggers of consecutive segments in the same acquisition. In fact, the timestamp counter is started when the Sequence acquisition is started, and keeps counting during the entire sequence. The difference between the timestamps of any pair of (not necessarily adjacent) segments is the time between their respective triggers.

The timestamp value is returned as a 64-bit integer, in units of picoseconds, with a resolution identical to that of the trigger time interpolator (see the appropriate product User manual). The waveform readout function **AcqrsD1_readData** returns the timestamp 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;
```

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

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

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

- In Visual Basic, use a decimal **Variant** variable as follows:

```
Const Two16 As Variant = 65536
Const Two32 As Variant = Two16 * Two16
Dim timeStamp As Variant, previousStamp as Variant
Dim timeDiff as Variant, xStampLo as Variant
...
If (tStampLo < 0) Then
    xStampLo = Two32 – Abs(tStampLo)
Else
```

```
        xStampLo = tStampLo
    End If
    timeStamp = CDec(tStampHi * Two32) + xStampLo
    ...
    timeDiff = 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 timestamp to an extended floating point number, and do arithmetic operations as with
  other variables.

- In LabWindows/CVI, the easiest way to manipulate timestamps is to convert them first to doubles:
  ```
  ViReal64 dlow, dhigh, tstamp;
  dlow = (ViReal64)low;
  dhigh = (ViReal64)high;
  tsamp = dlow + 4294967296.0 * dhigh;
  ```

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 12-bit-Family, the AC/SC Analyzers, the DP1400, 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/DC271/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 **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 reference 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 DP1400 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.*

*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 (models DC122, DC152, DC222, DC252, DC282, DP1400) 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, while it must be between 20 MHz and 2000 MHz for the second generation DC271-FAMILY or AP240/AP235 signal analysis platforms or, for the 12-bit-FAMILY, between 100 MHz and the value for the maximum allowed sampling frequency. Note that for normal operation of the 12-bit-FAMILY digitizers the sFmax for each converter should be kept above 50 MHz.

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	Input Frequency range (MHz)	sFmax vs. nbrConvertersPerChannel		
		1	2	4
DC122	100 – 2000	2 × inputFrequency		
DC135	20 – 2000	¼ × inputFrequency	½ × inputFrequency	
DC140	20 – 2000	½ × inputFrequency	1 × inputFrequency	
DC152	100 – 2000	1 × inputFrequency	2 × inputFrequency	
DC211/DC211A	20 – 2000	2 × inputFrequency		
DC222	100 – 2000	4 × inputFrequency		
DC241/DC241A	20 – 2000	1 × inputFrequency	2 × inputFrequency	
DC252	100 – 2000	2 × inputFrequency	4 × inputFrequency	
DC271/DC271A / DC271AR	20 – 2000	½ × inputFrequency	1 × inputFrequency	2 × inputFrequency
DC282	100 – 2000	1 × inputFrequency	2 × inputFrequency	4 × inputFrequency
DC436	100 – 200	½ × inputFrequency		
DC440/DC438	100 – 220	1 × inputFrequency		
DP1400	10 – 2000	1 × inputFrequency	2 × inputFrequency	
DP214	20 – 2000	1 × inputFrequency		
DP235/AP235	20 – 1000	½ × inputFrequency	1 × inputFrequency	
DP240/AP240	20 – 2000	½ × inputFrequency	1 × inputFrequency	
DP306	100 – 200	½ × inputFrequency		
DP308	100 – 200	1 × inputFrequency		
DP310	100 – 220	1 × inputFrequency		

Model	Input Frequency range (MHz)	sFmax vs. nbrConvertersPerChannel		
		1	2	4
AC210/SC210	20 – 2000	½ × inputFrequency		
AC240/SC240	20 – 2000	½ × inputFrequency	1 × inputFrequency	
1st generation	100 – 500	1 × 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:

$$\text{delay} = (\textbf{delayNbrSamples} – \textbf{nbrSamples}) / \textbf{sampFrequency}$$

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 µs. The delay would therefore have to be -10 µs 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:

```
// We assume that a normal calibration has been done, either
// during initialization, or explicitly
AcqrsD1_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 ± 0.5 samples. For implementation reasons, the acquired waveform in fact has a timing uncertainty that is twice as large, i.e. ± 1 samples. In this case, the trigger timestamps 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 amout 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 DP1400, 10-bit, or 12-bit 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 digitzer 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, 33.3e6, 33.3e6);
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:

```
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 amout 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. ASBus Operation

The ASBus and ASBus[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 ASBus 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 ASBus, 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.

3.17.1. Channel Numbering with ASBus

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

```
Acqrs_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 (), 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 **readChar/RealWform**, 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 ASBus

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 - 29	16 – 19	5 - 15	4	3	2	1	0
Ext 1	Ext 2	Other Ext Trigs	Module	Other Int Trigs	In 5	In 4	In 3	In 2	In 1

[0]	**IN 1**	Internal trigger channel 1
[1]	**IN 2**	Internal trigger channel 2
[2..4]	**IN 3, IN 4, IN 5**	Internal trigger channels 3, 4, 5
[5..15]	**OTHER INT TRIGS**	Other internal trigger channels within a module, up to 16.
[16..19]	**MODULE**	Module Number, running from 0 to (nbrModules – 1). In single digitizers, this field must be zero. In *MultiInstruments*, the trigger source number must be broken into a module number and a trigger channel number *within* the module.
[20..29]	**OTHER EXT TRIGS**	Other external trigger channels within a module, up to 12.
[30]	**EXT 2**	External trigger channel 2
[31]	**EXT 1**	External trigger channel 1

In future 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:

```
if (trigChan > 0)             // Internal Trigger
{
    long moduleNbr = (trigChan - 1) / nbrIntTrigsPerModule;
    long inputNbr  = (trigChan - 1) % nbrIntTrigsPerModule;
    srcPattern = (moduleNbr<<16) + (0x1<<inputNbr);
}
else if (trigChan < 0)        // External Trigger
{
    trigChan = -trigChan;
    long moduleNbr = (trigChan - 1) / nbrExtTrigsPerModule;
    long inputNbr  = (trigChan - 1) % nbrExtTrigsPerModule;
    srcPattern = (moduleNbr<<16) + (0x80000000>>inputNbr);
}
```

```
    else
        PROBLEM!
```

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 **AcqrsD1_configMode**. The default state of any digitizer is **mode** = 0 and **flags** = 0, corresponding to the *normal* digitizer operation, as described in the other sections of this manual.

3.18.1. Frequency Counter

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.

```
AcqrsD1_configFCounter(instrID, channel, type, targetValue, apertureTime,
                       0.0, 0);
```

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.

3.18.2. 'Start on Trigger'

The 'Start on Trigger' mode 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, and the 12-bit digitzers ;

      others (DP105, DP106, DP110, DP111, DP210, DP211, 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 ± ½ sampling interval.

In 'Start on Trigger' mode, the trigger occurs *before* recording starts. It still occurs asynchronously with respect to the internal VCO (which is *always* running). However, if the requested sampling rate is less than the VCO 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 ± ½ VCO time interval, **not** ± ½ sampling interval. Therefore when averaging, the bandwidth reduction will be less than in the 'normal' mode.

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:

```
AcqrsD1_configXXX(.. );           // configure other parameters
AcqrsD1_configMode(instrID, 0, 0, 1);
AcqrsD1_acquire(instrID);
AcqrsD1_waitForEndOfAcquisition(instrID, timeout);
// Read out data etc. before calling again "AcqrsD1_acquire"
```

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.

3.18.3. 'Sequence Wrap'

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 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.

While this mode even 'works' for **nbrSegments** = 1, in practice the value of **nbrSegments** should be at least 2. 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 ~ 1 µs) starts recording into it. Thus, the very last memory segment 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 5th segment, followed by the 6th, 7th, 8th, 1st etc. The 'youngest' useful segment is the 3rdone, while the 4th segment corresponds to the segment that was being recorded when the stop-command was received. The 4th segment (in this example) typically does not contain any useful data.

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:
    ```
    Acqrs_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 ASBus 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 $\geq 60\,^oC$ 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 $10\,^oC$ reduction in circuit temperature is expected to improve the mean-time-between-failures (MTBF) by a factor of 2.

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.

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
N	Number of samples per segment
Xfr	Transfer time per sample in μs, typically 0.01 ??
$Ovhd_{DMA}$	DMA overhead time (per segment) in μs
$Ovhd_{buffer}$	Circular buffer analysis overhead time (per segment) in μs
$Extra$	Number of 'overhead' data points per segment
Cpy	Time to copy a sample in μs

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 μs) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium $Ovhd_{buffer}$ is ~ 10 μs, and for a 500 MHz Pentium ~ 5 μs.

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:

Extra	Type
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 μs) / (Pentium CPU speed in MHz).
   Thus, for a 250 MHz Pentium *Cpy* is ~ 0.02 μs, and for a 500 MHz Pentium it is ~ 0.01 μs.

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 μs) / (Pentium CPU speed in MHz). Thus, for a 250 MHz Pentium *Conv* is ~ 0.14 μs, and for a 500 MHz Pentium ~ 0.07 μs.

4.2. Examples

(A) DP210 in a 800 MHz Pentium:

Transferring single records of 100'000 samples, recorded at 2 GS/s:

$$M = 1 \quad N = 100\,000 \quad Xfr = 0.01 \quad Cpy = 0.00625$$
$$Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672$$
$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 62.5 + 1000 = \underline{1063\,\mu s}$$
$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$
$$= 62.5 + 3.125 + 100\,672 \cdot 0.01 + 100\,000 \cdot 0.00625 = \underline{1698\,\mu s}$$

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:

$$M = 100 \quad N = 1000 \quad Xfr = 0.01 \quad Cpy = 0.00625$$
$$Ovhd_{DMA} = 62.5 \quad Ovhd_{buffer} = 3.125 \quad Extra = 480 + 192 = 672$$
$$T_1 = M \cdot Ovhd_{DMA} + M \cdot N \cdot Xfr = 100 \cdot 62.5 + 100 \cdot 1000 \cdot 0.01 = \underline{7250\,\mu s}$$
$$T_2 = Ovhd_{DMA} + M \cdot Ovhd_{buffer} + M \cdot (N + Extra) \cdot Xfr + M \cdot N \cdot Cpy$$
$$= 62.5 + 100 \cdot 3.125 + 100 \cdot 1672 \cdot 0.01 + 100 \cdot 1000 \cdot 0.00625 = \underline{2672\,\mu s}$$

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

(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:

$$M = 1 \quad N = 100\,000 \quad Xfr = 0.012 \quad Cpy = 0.01$$

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

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

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

$$= 100 + 5 + 100\,120 \cdot 0.012 + 100\,000 \cdot 0.01 = \underline{2306\,\mu s}$$

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:

$$M = 1000 \quad N = 500 \quad Xfr = 0.012 \quad Cpy = 0.01$$

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

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

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

$$= 100 + 1000 \cdot 5 + 1000 \cdot 716 \cdot 0.012 + 1000 \cdot 500 \cdot 0.01 = \underline{18\,692\,\mu s}$$

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**

Time in ms		250 MHz Pentium			500 MHz Pentium			800 MHz Pentium		
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**

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).

Time in ms		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.