Programmers Reference
Perception RPC interface
Perception RPC interface

Document version 1.0 – April 2009

For Perception 6.0 or higher

For HBM's Terms and Conditions visit www.hbm.com/terms

HBM GmbH
Im Tiefen See 45
64293 Darmstadt
Germany
Tel: +49 6151 80 30
Fax: +49 6151 8039100
Email: info@hbm.com
www.hbm.com/highspeed

Copyright © 2009

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.
Table of Contents

TABLE OF CONTENTS .......................................................................................................................... 3

1 GETTING STARTED ............................................................................................................................... 5
  1.1 INTRODUCTION .......................................................................................................................... 5
  1.2 INTENDED AUDIENCE .............................................................................................................. 6
  1.3 REQUIREMENTS AND INSTALLATION ...................................................................................... 6
    1.3.1 System requirements ........................................................................................................ 6
    1.3.2 Supported hardware ...................................................................................................... 6
    1.3.3 Installation .................................................................................................................. 6
  1.4 STARTING THE RPC INTERFACE .......................................................................................... 7
    1.4.1 To quit the Perception RPC interface: ........................................................................ 7

2 USING PERCEPTION RPC .................................................................................................................. 8
  2.1 COMMAND OVERVIEW ........................................................................................................... 8
    2.1.1 Acquisition control ........................................................................................................ 8
    2.1.2 State retrieval ............................................................................................................... 8
    2.1.3 File operations ............................................................................................................. 8
    2.1.4 Settings ...................................................................................................................... 9
    2.1.5 Data retrieval ............................................................................................................. 9
  2.2 THE IDL AND ACF FILES ......................................................................................................... 9
    2.2.1 The PerceptionRPC.idl file ......................................................................................... 9
  2.3 USAGE OF SUPPORT FILES .................................................................................................... 11
  2.4 COMPILER CONSIDERATIONS ............................................................................................... 13
  2.5 YOUR FIRST RPC PROGRAM .................................................................................................. 13

3 PERCEPTION RPC REFERENCE ......................................................................................................... 15
  3.1 ACQUISITION CONTROL .......................................................................................................... 15
    3.1.1 Start .............................................................................................................................. 15
    3.1.2 Stop ............................................................................................................................. 15
    3.1.3 Pause .......................................................................................................................... 15
    3.1.4 Trigger ......................................................................................................................... 16
    3.1.5 OneShot ..................................................................................................................... 16
    3.1.6 Example to the acquisition control functions .............................................................. 16
  3.2 ACQUISITION STATUS ............................................................................................................... 18
    3.2.1 GetAcquisitionState GetTriggerState ........................................................................ 18
    3.2.2 Example to the acquisition/trigger status functions ..................................................... 21
    3.2.3 Timing considerations ................................................................................................. 22
  3.3 FILE OPERATIONS ..................................................................................................................... 23
    3.3.1 GetFilenames ............................................................................................................... 24
  3.4 LOADVWB ................................................................................................................................. 25
    3.4.1 SaveVWB ..................................................................................................................... 25
    3.4.2 LoadSettings ............................................................................................................... 26
    3.4.3 SaveSettings ................................................................................................................. 26
    3.4.4 Example to the file operation functions ....................................................................... 27
  3.5 HARDWARE SETTINGS ............................................................................................................... 29
    3.5.1 [Set|Get]ChannelEnabled ........................................................................................... 30
    3.5.2 [Set|Get]TriggerSettings ............................................................................................ 31
    3.5.3 [Set|Get]TimebaseSettings ......................................................................................... 32
    3.5.4 [Set|Get]StorageSettings ........................................................................................... 33
    3.5.5 [Set|Get]SweepSettings .............................................................................................. 34
    3.5.6 [Set|Get]ContinuousSettings .................................................................................... 36
    3.5.7 Retrieve hardware names ............................................................................................. 37
    3.5.8 Example to the hardware settings functions ............................................................... 37

3.6 DATA RETRIEVAL ......................................................................................................................... 40
Perception RPC interface

3.6.1 GetAvailableWaveforms ................................................................................................... 40
3.6.2 SubscribeLiveWaveform ................................................................................................. 42
3.6.3 UnsubscribeLiveWaveform ............................................................................................ 44
3.6.4 GetData ........................................................................................................................... 44
3.6.5 Example to the waveform data functions ...................................................................... 45
3.6.6 Understanding min-max pairs ...................................................................................... 49
3.6.7 [Set][Get]SCInputRangeSettings .................................................................................... 51
3.6.8 [Set][Get]RecStorageSettings ....................................................................................... 52

4 APPENDIX: USING RPC-COM WRAPPER AND C# ............................................................... 54
4.1 INTRODUCTION ................................................................................................................. 54
4.2 GETTING STARTED ............................................................................................................. 55
4.3 CREATING YOUR FIRST RPC-COM CLIENT APPLICATION .............................................. 56
4.4 EXTENDING THE APPLICATION WITH ACQUISITION CONTROL ................................. 60
1 Getting Started

Welcome to the HBM Perception Remote Procedure Call (RPC) interface. RPC is a powerful, robust, efficient, and secure inter-process communication (IPC) mechanism that enables data exchange and invocation of functionality residing in a different process. That different process can be on the same machine, on the local area network, or across the Internet. Remote Procedure Call defines a powerful technology for creating distributed client/server programs. RPC can be used in all client/server applications based on Microsoft® Windows® operating systems. It can also be used to create client and server programs for heterogeneous network environments that include such operating systems as Unix, Linux and Apple.

1.1 Introduction

A data acquisition system consisting of a Gen series data acquisition system in combination with Perception controlling software can be part of a larger data acquisition system.

Programmers of the overall system might want to control the Gen series Perception acquisition subsystem from other software, which might run on the same PC as the Perception software, but (more often) which could also run on another computer on the local area network. This other computer can be a Windows-based platform but also Linux and Unix based platforms are used as overall controlling computers.

It is not possible to control the Gen series data acquisition system directly, but the user can control Perception (which controls the hardware), and from both an application and a programming point of view this is then the same. It does not matter where to send a command to – to a receiving hardware or receiving software - as long as the desired result is achieved.

In order to control the Perception software from another application, the Perception RPC interface can be used. This option enables Perception to be remotely controlled.

This document describes the RPC interface to control Perception. It contains a command overview, command reference and examples. Examples are written in C/C++ as console applications using the Microsoft Visual Studio development environment.

Examples are also available in C#, these examples are using a COM wrapper around the RPC client and are handled in the appendix of this manual.
This manual is only describing a part of the RPC command set; all commands can be found in the HBM RPC/COM help file.

1.2 Intended audience

RPC is designed to be used by C/C++/C#/VB programmers. You must be proficient in your programming language and RPC interface technology in order to write custom programs. This documentation assumes you understand your HBM equipment, software, and basic acquisition terminology.

Understanding acquisition terminology is vital to understanding digital recordings: trigger, sample rate, pre-/post trigger, etc.

1.3 Requirements and installation

The HBM RPC Interface is an option that is enabled through the use of the HASP®4 USB Token. When this option is installed, a colored icon is shown on the splash screen at start-up. When this icon is grayed you should contact your local dealer for more information on how to obtain this option.

This option is also listed as Remote API in the Perception menu:

Help > About HBM Perception > More... > Options page

In addition you must install the required software modules as described below.

1.3.1 System requirements

- HBM Perception software with Remote API option enabled

1.3.2 Supported hardware

- HBM GEN series Modular Data Acquisition System
- HBM Liberty Ruggedized In-vehicle Data Acquisition System

1.3.3 Installation

You will need to copy the supplied Perception.idl and Perception.acf files into your project directory. Additionally you can copy also the supplied support files.
1.4 Starting the RPC interface

The RPC interface, when enabled, is part of the Perception software engine. When you start Perception you have direct access to the RPC interface.

Be sure the Perception RPC Server is enabled. You should go to the Preferences dialog and enable the RPC server, see picture below.

1.4.1 To quit the Perception RPC interface:

The Perception RPC interface is automatically closed when Perception is closed.
2 Using Perception RPC

Procedure-oriented programming languages provide simple mechanisms for specifying and writing procedures. For example, the ANSI-standard C-function prototype is a construct used to specify the name of a procedure, the type of the result it returns (if any) and the number, sequence, and type of its parameters. Using the function prototype is a formal way to specify an interface between procedures.

RPC builds on that programming model by allowing procedures, grouped together in interfaces, to reside in different processes than the caller. RPC also adds a more formal approach to procedure definition that allows the caller and the called routine to adopt a contract for remotely exchanging data and invoking functionality. In the RPC programming model, traditional function calls are supplemented with an additional element:

- An .idl/.acf file that precisely describes the data exchange and parameter-passing mechanism between the caller and called procedure.

Later in this document we will describe the use of these files.

2.1 Command overview

The Perception RPC interface provides a variety of functions, divided into 5 basic categories as described below. For a complete list of commands we refer to the HBM RPC-COM help file. This help file describes all functions while this manual only describes a sub set.

2.1.1 Acquisition control

The acquisition controls are used to start, stop and pause recordings. You can also initiate a one-shot sweep or generate a trigger.

- **Start**  
  Start recording using the active settings.
- **Stop**  
  Stop the active recording.
- **OneShot**  
  This method initiates a single-shot sweep.
- **Pause**  
  Pauses the active recording.
- **Trigger**  
  Generate a (software) trigger.

2.1.2 State retrieval

It is possible to retrieve the acquisition state and trigger state.

- **GetAcquisitionState**  
  Retrieve the current acquisition state of all recorders.
- **GetTriggerState**  
  Retrieve the current trigger state of all recorders.

2.1.3 File operations

Through file operations you can open and save Workbenches and settings or retrieve a list of available files in a specific directory.

- **GetFilenames**  
  Retrieve all available filenames from a specific directory.
- **LoadVwb**  
  Load the specified workbench.
SaveVwb
Save the workbench to a specified file.

LoadSettings
Load the settings from a selected file.

SaveSettings
This method saves the settings to the specified file.

### 2.1.4 Settings

These commands allow you to set and get various parameters related to acquisition, trigger and storage. Most of these commands operate on a selected mainframe/recorder/channel.

- **[Set][Get]ChannelEnabled** Enables a channel for storage.
- **[Set][Get]TriggerSettings** Trigger mode and levels.
- **[Set][Get]TimebaseSettings** Sample rate settings.
- **[Set][Get]StorageSettings** Type of storage.
- **[Set][Get]SweepSettings** Typical sweep related settings like pre- and post trigger.
- **[Set][Get]ContinuousSettings** Typical settings for continuous acquisition mode.

All these commands are described in detail in the command reference section.

### 2.1.5 Data retrieval

Fetch a data stream from available waveforms. For each waveform a subscription is needed in order to get the data. This is the same live data that is on a Perception display, i.e. reduced using min-max pairs.

- **GetAvailableWaveforms** Get a list of available waveforms.
- **SubscribeLiveWaveform** Connect to a selected waveform.
- **UnsubscribeLiveWaveform** Disconnect from a selected waveform.
- **GetData** Returns data of the specified subscription.

### 2.2 The IDL and ACF files

Two distinct files are used to provide the RPC interface: the Interface Definition Language (IDL) file and the application configuration file (ACF). These files contain attributes that direct the generation of the C-language stub files that manage the remote procedure call. The IDL file contains a description of the interface between the client and the server programs. RPC applications use the ACF file to describe the characteristics of the interface that are specific to the hardware and operating system that make up a particular operating environment. The purpose of dividing this information into two files is to keep the software interface separate from characteristics that affect only the operating environment.

The IDL file specifies a network contract between the client and server—that is, the IDL file specifies what is transmitted between the client and the server. Keeping this information distinct from the information about the operating environment makes the IDL file portable to other environments.

The ACF file contains configuration attributes that apply to types and functions defined in the interface body of the IDL file. The ACF specifies behaviour on the local computer and does not affect the data transmitted over the network.

#### 2.2.1 The PerceptionRPC.idl file

The supplied PerceptionRPC.idl file contains the typical IDL components, but also includes the following specific item:
This compiler directive is used to create a datatype called `signed32` to be used on Windows systems.

Included are also the various constants that are used throughout this document:

```c
typedef enum WaveMonitorTriggerMode
{
    WMTM_FreeRunning   = 0,
    WMTM_TrigOnly   = 1,
    WMTM_TrigOrStabilizer  = 3,
    WMTM_TrigOrTimeout  = 5,
    WMTM_TrigOrStabilizeOrTimeout = 7,
}WaveMonitorTriggerMode;

typedef enum StorageMode
{
    STM_None    = 0,
    STM_Sweeps    = 1,
    STM_Continuous   = 2,
    STM_Dual    = 3,
} StorageMode;

typedef enum RecordingMode
{
    RM_StopOnFull   = 0,
    RM_CircularRecording  = 1,
    RM_LimitedRecording  = 2,
    RM_StopOnTrigger   = 3,
} RecordingMode;

typedef enum ChannelTriggerDirection
{
    CTD_RisingEdge   = 0,
    CTD_FallingEdge   = 1,
} ChannelTriggerDirection;

typedef enum EnableMode
{
    CENA_No    = 0,
    CENA_Yes    = 1,
} EnableMode;

typedef enum SweepMode
{
    SWM_Normal    = 0,
    SWM_PreTrigger   = 1,
    SWM_Delayed    = 2,
} SweepMode;

typedef enum SweepCountEnable
{
    SCE_No    = 0,
} SweepCountEnable;
```
The various structures used and the function declarations conclude this file.

2.3 Usage of support files

When using the Microsoft Visual Studio programming environment you must include the supplied PerceptionRPC.idl and PerceptionRPC.acf files in your project.

Below you see an example of these files included in a tutorial solution/project.

Before you can use the Perception RPC interface it is advised to create an rpcsupport.h and rpcsupport.cpp file. The rpcsupport.h looks as follows:

```c
#pragma once
#define _WIN32_WINNT 0x500
#include "PerceptionRPC_h.h"

bool InitRPC(char * szAddress);
```

The initRPC function initializes the RPC interface and is defined in rpcsupport.cpp:
#include "StdAfx.h"
#include "rpcsupport.h"
#pragma comment(lib,"rpcrt4.lib")

extern "C"
{
    #include "PerceptionRPC_c.c"
}

void* __RPC_USER midl_user_allocate(size_t size)
{
    return malloc(size);
}

void __RPC_USER midl_user_free(void* p)
{
    free(p);
}

bool InitRPC(char * szAddress)
{
    RPC_STATUS status;
    unsigned char* szStringBinding = NULL;

    if(_tcslen(szAddress) <= 0)
    {
        return FALSE;
    }

    status = RpcStringBindingCompose(NULL,
        reinterpret_cast<unsigned char*>("ncacn_ip_tcp"),
        reinterpret_cast<unsigned char*>(szAddress),
        reinterpret_cast<unsigned char*>("4747"),
        NULL, &szStringBinding);

    if (status)
    {
        return FALSE;
    }

    status = RpcBindingFromStringBinding(
        szStringBinding,&hPerceptionRPCBinding);
    RpcStringFree(&szStringBinding);

    if (status)
    {
        return FALSE;
    }

    return TRUE;
}

The main program now could look like this:
#include "stdafx.h"  // standard stuff
#include "rpcsupport.h"  // basic rpc support like initRPC

char szSystemName[64];  // network name of computer that
// runs the Perception software

int _tmain(int argc, _TCHAR* argv[])
{
    strcpy(szSystemName, "control_pc");
    // Initialise RPC, continue when successful
    if (InitRPC(szSystemName))
    {
        // do your stuff here
    }
    return FALSE;
}

2.4 Compiler considerations

Currently there are no additional compiler considerations.

2.5 Your first RPC program

Using the aforementioned information you now should be able to create, compile and run your
first RPC program:

//
// MyFirstRPC.cpp : Defines the entry point
// for the console application.
//
#include "stdafx.h"  // standard stuff
#include "rpcsupport.h"  // basic rpc support like initRPC

#define MAX_NAMESTRING 64
 char szSystemName[MAX_NAMESTRING];

int _tmain(int argc, _TCHAR* argv[])
{
    // use the computer name or IP-address
    strcpy(szSystemName, "control_pc");
    // Initialise RPC, continue when successful
    if (InitRPC(szSystemName))
    {
        Start ();  // start acquisition
    }
    // done
In order to run this program, you must have Perception running on the computer that has the name as specified in `szSystemName`. This can be the same computer that you use to create your program.

Also make sure that acquisition hardware is connected and selected in Perception. Or use the “GEN7t Firmware Simulator” to simulate the presence of acquisition hardware.

When you run this program, the acquisition status will change to “run”.

```c
return FALSE;
}
```
3 Perception RPC Reference

This chapter of the document describes all available functions within the Perception RPC interface. For each function a description is given and the correct syntax, including an explanation of each parameter as well as the applicable return values. For each function also an example is given.

3.1 Acquisition control

Various functions are available to control acquisition. You can start, stop, and pause an acquisition. When an acquisition is active you can generate 'software' triggers. Also a function is available to initiate a single shot recording.

3.1.1 Start

Use this function to start an acquisition.

Syntax
Start()

Return values
If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example
The following example demonstrates the use of the Start function.

```c
if (Start() == 0)
{
    // do your stuff here when start returns ok
}
```

3.1.2 Stop

Use this function to stop an acquisition.

Syntax
Stop()

Return values
If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example
The following example demonstrates the use of the Stop function.

```c
if (Stop() == 0)
{
    // do your stuff here when stop returns ok
}
```

3.1.3 Pause

Use this function to hold an acquisition temporarily.
Perception RPC interface

Syntax

Pause()

Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example

The following example demonstrates the use of the Pause function.

```c
if (!Pause()) // same as Pause() == 0
{
    // do your stuff here when pause returns ok
}
```

3.1.4 Trigger

This command initiates a trigger. Depending on the acquisition settings this can finish the active acquisition.

Syntax

Trigger()

Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example

The following example demonstrates the use of the Trigger function.

```c
if (!Trigger()) // same as Trigger() == 0
{
    // do your stuff here when trigger returns ok
}
```

3.1.5 OneShot

This command initiates a single shot acquisition (or sweep).

Syntax

OneShot()

Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example

The following example demonstrates the use of the OneShot function.

```c
if (OneShot() == 0)
{
    // do your stuff here when oneshot returns ok
}
```

3.1.6 Example to the acquisition control functions

The following code is an example that combines the various acquisition control functions into a single console application.
// program 1: acquisition control – author: Cees J de Vries
// (c) 2005 LDS Test and Measurement

#include "stdafx.h" // standard stuff
#include "rpcsupport.h" // basic rpc support like initRPC
#include <iostream> // basic I/O
using namespace std;

#define MAX_NAMESTRING 128
char szSystemName[MAX_NAMESTRING];

int _tmain(int argc, char* argv[])
{
    // use the computer name or IP-address
    strcpy(szSystemName, "10.134.144.123");
    char cAnyKey = ' ';

    // Initialise RPC, continue when successful
    if (InitRPC(szSystemName))
    {
        // these are the possibilities:
        cout << "Enter (S)tart, s(T)op, (P)ause, (O)neshot, t(R)igger or (Q)uit." << endl;
        // for demonstration purposes no error checking
        // continue until quit
        while (!(cAnyKey == 'Q') || (cAnyKey == 'q'))
        {
            // fetch keyboard input.
            // allow upper and lower case commands
            cin >> cAnyKey;
            switch (cAnyKey)
            {
            case 's':
            case 'S':
                cout << "System start, you can use trigger." << endl;
                Start();
                break;
            case 't':
            case 'T':
                cout << "System stop." << endl;
                Stop();
                break;
            case 'p':
            case 'P':
                cout << "Pause." << endl;
                Pause();
                break;
            case 'o':
            case 'O':
                cout << "Single shot acquisition, you can use trigger." << endl;
                Start();
            default:
                continue;
            }
        }
    }
    return 0;
}
```cpp
OneShot ();
break;
case 'r':
case 'R':
    cout << "Trigger." << endl;
    Trigger ();
    break;
default:
    break;
}
}
// done
return FALSE;
```

Before you run this program, make sure Perception is active on the machine as identified by `szSystemName`. When you run this program you should see the acquisition control buttons and indicators in Perception change according to the commands issued. The following image shows an example of the console application itself in action.

### 3.2 Acquisition status

Using the functions described in this section you can query the acquisition status and the trigger status of the available recorders. Multiple values are returned in one go.

#### 3.2.1 GetAcquisitionState

Use this function to retrieve the acquisition status of all available recorders.

**Syntax**  
```
GetAcquisitionState(&iRun, &iOneShot, &iStop, &iPause, &iIdle)
```
Perception RPC interface

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRun</td>
<td>signed32</td>
<td>Number of recorders currently running</td>
</tr>
<tr>
<td>iOneShot</td>
<td>signed32</td>
<td>Number of recorders currently in sweep mode</td>
</tr>
<tr>
<td>iStop</td>
<td>signed32</td>
<td>Number of recorders currently stopped</td>
</tr>
<tr>
<td>iPause</td>
<td>signed32</td>
<td>Number of recorders currently in pause mode</td>
</tr>
<tr>
<td>iIdle</td>
<td>signed32</td>
<td>Number of recorders currently idle</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the GetAcquisitionState function.

```cpp
signed32 iRun, iOneShot, iStop;
signed32 iPause, iIdle;

GetAcquisitionState(&iRun, &iOneShot, &iStop, &iPause, &iIdle);
cout << iRun << " " << iOneShot << " " << iStop
    << " " << iPause << " " << iIdle << endl;
```

**GetTriggerState**

Use this function to retrieve the trigger status of all available recorders. When for one reason or another a recorder is not able to accept triggers, e.g. memory is full, this recorder is said to be “blocked”.

**Syntax**

GetTriggerState(&iIdle, &iBlocked, &iWasBlocked, &iArmed, &iTriggered, &iTriggerCount, &iAllTriggerCountsEqual)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iIdle</td>
<td>signed32</td>
<td>Number of recorders currently idle</td>
</tr>
<tr>
<td>iBlocked</td>
<td>signed32</td>
<td>Number of recorders currently blocked</td>
</tr>
<tr>
<td>iWasBlocked</td>
<td>signed32</td>
<td>Total number of blocked recorders</td>
</tr>
<tr>
<td>iArmed</td>
<td>signed32</td>
<td>Number of recorders currently armed</td>
</tr>
<tr>
<td>iTriggered</td>
<td>signed32</td>
<td>Number of recorders currently triggered</td>
</tr>
<tr>
<td>iTriggerCount</td>
<td>signed32</td>
<td>Maximum number of triggers so far</td>
</tr>
<tr>
<td>iAllTriggerCountsEqual</td>
<td>signed32</td>
<td>Nonzero: all recorders have equal trigger count</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.
Example
The following example demonstrates the use of the GetTriggerState function.

```c
signed32 iIdle, iBlocked, iWasBlocked, iArmed;
signed32 iTriggered, iTriggerCount, iAllTriggerCountsEqual;
GetAcquisitionState(&iIdle, &iBlocked, &iWasBlocked, &iArmed,
&iTriggered, &iTriggerCount, &iAllTriggerCountsEqual);

cout << iIdle << " " << iBlocked << " " << iWasBlocked << " " <<
iArmed << " " << iTriggered << " " << iTriggerCount << " "
<< iAllTriggerCountsEqual << endl;
```
3.2.2 Example to the acquisition/trigger status functions

The following code is an example that combines the various acquisition control functions and the acquisition/trigger status functions into a single console application.

```
//
// program 2: acquisition control and status
// author: Cees J de Vries
// (c) 2005 LDS Test and Measurement
//
#include "stdafx.h"    // standard stuff
#include "rpcsupport.h" // basic rpc support like initRPC
#include <iostream>     // basic I/O
using namespace std;

#define MAX_NAMESTRING 128
char szSystemName[MAX_NAMESTRING];

int _tmain(int argc, char* argv[])
{
    // use the computer name or IP-address
    strcpy(szSystemName, "10.134.144.123");
    char cAnyKey = ' ';
    signed32 iRun, iOneShot, iStop;
    signed32 iPause, iAcqIdle;
    signed32 iTrigIdle, iBlocked, iWasBlocked, iArmed;
    signed32 iTriggered, iTriggerCount, iAllTriggerCountsEqual;

    // Initialise RPC, continue when successful
    if (InitRPC(szSystemName))
    {
        // these are the possibilities:
        cout << "Enter (S)tart, s(T)op, (P)ause, (O)neshot, t(R)igger or (Q)uit." << endl;
        // for demonstration purposes no error checking
        // continue until quit
        while (!((cAnyKey == 'Q') || (cAnyKey == 'q')))
        {
            // fetch keyboard input.
            // allow upper and lower case commands
            cin >> cAnyKey;
            switch (cAnyKey)
            {
                case 's':
                case 'S':
                    cout << "System start. Status: ";
                    Start();
                    break;
                case 't':
                case 'T':
                    cout << "System stop. Status: ";
                    Stop();
                    break;
            }
        }
    }
```
case 'p':
case 'P':
    cout << "Pause. Status: ";
    Pause ();
    break;
case 'o':
case 'O':
    cout << "Single shot. Status: ";
    OneShot ();
    break;
case 'r':
case 'R':
    cout << "Trigger. Status: ";
    Trigger ();
    break;
default:
    break;
}
GetAcquisitionState(&iRun, &iOneShot, &iStop,
    &iPause, &iAcqIdle);
GetTriggerState(&iTrigIdle, &iBlocked,
    &iWasBlocked, &iArmed, &iTriggered,
    &iTriggerCount, &iAllTriggerCountsEqual);
printf("%d %d %d %d %d - ", iRun, iOneShot,
    iStop, iPause, iAcqIdle);
printf("%d %d %d %d %d %d %d\n", iTrigIdle,
    iBlocked, iWasBlocked, iArmed,
    iTriggered, iTriggerCount,
    iAllTriggerCountsEqual);

Before you run this program, make sure Perception is active on the machine as identified by szSystemName. When you run this program you should see the acquisition control buttons and indicators in Perception change according to the commands issued.

3.2.3 Timing considerations

In the above example (and other examples in this document) timing issues are not taken into account. Therefore the result of the output - especially status information - can be different between the examples and your own situation. When you send any command there is an undetermined time lag between the moment you issue the command and the moment it is actually executed. This is also true for retrieval of status information. Adding a delay may not always be desirable or sufficient. Typically status retrieval should be done by 'polling'.

The following image shows an example of the console application itself in action.
The application controls two recorders. When generating triggers you can see that, depending on the timing, recorders can go from armed to triggered.

### 3.3 File operations

File operations are used to open and save setups. It is also possible to retrieve a list of available files in a specified directory.

The `GetFileNames` function supports virtual directory names that get replaced automatically by the real locations. These virtual directories are:

- **SharedDocs** - Represents the Shared Documents folder on the computer (C:\Documents and Settings\All Users\Documents)
- **MyDocs** - Represents the My Documents folder of the currently logged in user (C:\Documents and Settings\Username\My Documents)

Virtual directories names are surrounded by the two characters `< and >`, called angle brackets.
3.3.1 GetFilenames

With this function you can retrieve all available file names that match a search pattern, from a specified directory.

**Syntax**

```
GetFileNames(szLocation, szSearchString, &pfileNames)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szLocation</td>
<td>string</td>
<td>Search directory / folder</td>
</tr>
<tr>
<td>szSearchString</td>
<td>string</td>
<td>Search string</td>
</tr>
<tr>
<td>pfileNames</td>
<td>pointer</td>
<td>pointer to a string array that holds the search results</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

In `PerceptionRPC.idl` the `stringArray` structure is defined as follows:

```cpp
typedef [ptr, string] char * string_t;

typedef struct {
    signed32 stringCount;
    [size_is(stringCount)] string_t strings[];
} stringArray
```

**Example**

The following example demonstrates the use of the GetFileNames function.

```cpp
cchar szLocation[256];
cchar szSearchString[256];

strcpy(szLocation,"E:\Temp");
// you can also use virtual directory names, e.g.
// strcpy(szLocation,"<MyDocs>");
strcpy(szSearchString, "*.vwb");

stringArray * pfileNames = new stringArray;
int i;

if (GetFilenames(szLocation, szSearchString, &pfileNames) == 0) {
    for (i = 0 ; i < pfileNames->stringCount ; i++)
    {
        cout << pfileNames->strings[i] << endl;
    }
}
### 3.4 LoadVWB

Use this function to load an existing Virtual WorkBench (*.vwb) file into Perception.

#### Syntax

```
LoadVWB(szFullPath)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szFullPath</td>
<td>string</td>
<td>Full path and file name</td>
</tr>
</tbody>
</table>

#### Return values

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

In `PerceptionRPC.idl` the `FileName` type is defined as follows:

```
typedef [string] char FileName[300];
```

#### Example

The following example demonstrates the use of the `LoadVWB` function.

```c
FileName szFullPath;
strcpy(szFullPath, "E:\Temp\MyWorkbench.vwb");
LoadVWB(szFullPath);
```

### 3.4.1 SaveVWB

Use this function to save the current Perception workbench into a file.

#### Syntax

```
SaveVWB(szFullPath)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szFullPath</td>
<td>string</td>
<td>Full path and file name</td>
</tr>
</tbody>
</table>

#### Return values

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

#### Example

The following example demonstrates the use of the `SaveVWB` function.

```c
FileName szFullPath;
strcpy(szFullPath, "E:\Temp\MySavedWorkbench.vwb");
SaveVWB(szFullPath);
```
### 3.4.2 LoadSettings

Use this function to load a saved settings file (*.pset) into Perception.

**Syntax**

```
LoadSettings(szFullPath)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szFullPath</td>
<td>string</td>
<td>Full path and file name</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the LoadSettings function.

```
FileName szFullPath;
strcpy(szFullPath, "E:\Temp\GenesisSet1.pset");
LoadSettings(szFullPath);
```

### 3.4.3 SaveSettings

Use this function to save the current hardware settings into a file.

**Syntax**

```
SaveSettings(szFullPath)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szFullPath</td>
<td>string</td>
<td>Full path and file name</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the SaveSettings function.

```
FileName szFullPath;
strcpy(szFullPath, "E:\Temp\SavedLibertySet.pset");
SaveSettings(szFullPath);
```
3.4.4 Example to the file operation functions

The following code is an example that combines various file operation functions into a single console application. First a search is performed in a specific directory for all *.vwb files. After this the user can select which file to load. Before actually loading a backup is created in the same directory.

```cpp
// program 3: file operations - author: Cees J de Vries
// (c) 2005 LDS Test and Measurement

#include "stdafx.h"    // standard stuff
#include "rpcsupport.h"    // basic rpc support like initRPC
#include <iostream>    // basic I/O
using namespace std;
#define MAX_NAMESTRING 128
char szSystemName[MAX_NAMESTRING];
char szLocation[MAX_NAMESTRING];
char szSearchString[MAX_NAMESTRING];
int _tmain(int argc, char* argv[])
{
    // use the computer name or IP-address
    strcpy(szSystemName, "10.134.144.123");
    // define directory and search condition
    strcpy(szLocation,"E:\Temp\");
    strcpy(szSearchString, "*.vwb");
    // create the string array pointer
    stringArray *pfileNames = new stringArray;
    FileName FullPath;
    int iAnyKey = 0;
    int i;

    // Initialise RPC, continue when successful
    if (InitRPC(szSystemName))
    {
        if (GetFilenames(szLocation, szSearchString,
                         pfileNames) == 0)
        {
            // loop for number of files found
            for (i = 0; i < pfileNames->stringCount; i++)
            {
                // display count and name
                cout << i + 1 << ":" << " 
                pfileNames->strings[i] << endl;
            }
        }
    }
    cout << endl << "Enter your selection
(0 to cancel): ";
    cin >> iAnyKey;
    if (iAnyKey > 0)
```
```cpp
if (pfileNames->stringCount > 0)
{
    if (iAnyKey <= pfileNames->stringCount)
    {
        // create a backup first
        // using SaveVWB
        strcpy(FullPath, szLocation);
        strcat(FullPath, "backup.vwb");
        cout << endl << "Creating backup "
             << "backup.vwb";
        SaveVWB(FullPath);

        // load the requested vwb
        // with LoadVWB
        strcpy(FullPath, szLocation);
        strcat(FullPath,
                 pfileNames->strings[iAnyKey-1]);
        cout << endl << endl <<
             "Loading " << FullPath
             << endl;
        cout << endl << "Please wait ..." << endl << "this may take a minute or so ;-)"
             << endl << endl;
        LoadVWB(FullPath);
        cout << "Press any key followed by Enter to quit";
        cin >> iAnyKey;
    }
}
// done
return FALSE;
```

Next is an example of the actual console application.
3.5 Hardware settings

How a recording is made is determined by a variety of 'settings' that are stored in the actually hardware. They determine at what rate data is converted, when to trigger, what to store, etc. There are numerous settings. The Perception RPC interface allows you to query and modify the settings most relevant to basic recording capabilities.

The settings described in this section require one or more of the following parameters:

- **Mainframe** - a mainframe is the physical housing that comprises one or more recorders.

- **Recorder** - a recorder is usually a single board with a number of channels. Each recorder can have its own timebase and trigger settings.

- **Channel** - one or more channels are within a single recorder. All channels within a single recorder use the same timebase and trigger settings.

These parameters can be either an index in the form of $<n>$, or the actual name. The actual name can be retrieved by one of the Get...Name functions as described at the end of this section.
3.5.1 **[Set][Get]ChannelEnabled**

For each channel you can define whether its data should be stored yes or no.

**Syntax**

SetChannelEnabled(szMainframe, szRecorder, szChannel, iEnabled)
GetChannelEnabled(szMainframe, szRecorder, szChannel, &iEnabled)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>szChannel</td>
<td>string</td>
<td>Name or index of channel</td>
</tr>
<tr>
<td>iEnabled</td>
<td>signed32</td>
<td>Channel enabled:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CENA_No = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CENA_Yes = 1</td>
</tr>
</tbody>
</table>

**Return values**

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the ChannelEnabled functions.

```cpp
char szMainframeName[64];
char szRecorderName[64];
char szChannelName[64];
signed32 iEnabled;

// load names -> use index only
strcpy(szMainframeName, "$1$" );
strcpy(szRecorderName, "$1$" );
strcpy(szChannelName, "$1$" );

// Get status from 1st channel, 1st recorder, 1st mainframe
GetChannelEnabled(szMainframeName, szRecorderName,
    szChannelName, &iEnabled);
cout << iEnabled;

// Set enable of 2nd channel to false -> no storage
strcpy(szChannelName, "$2$" );
SetChannelEnabled(szMainframeName, szRecorderName,
    szChannelName, CENA_No);
```
3.5.2 [Set][Get]TriggerSettings

You can define the basic trigger parameters on a per channel basis.

Syntax

SetTriggerSettings(szMainframe, szRecorder, szChannel, iTTriggerMode,
              dPrimaryLevel, dSecondaryLevel, iDirection)
GetTriggerSettings(szMainframe, szRecorder, szChannel, &iTTriggerMode,
        &dPrimaryLevel, &dSecondaryLevel, &iDirection)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>szChannel</td>
<td>string</td>
<td>Name or index of channel</td>
</tr>
<tr>
<td>iTTriggerMode</td>
<td>signed32</td>
<td>Channel trigger mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_Off = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_Basic = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_Dual = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_Window = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_DualWindow = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_Sequential = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_BasicQualifier = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTM_DualQualifier = 7</td>
</tr>
<tr>
<td>dPrimaryLevel</td>
<td>double</td>
<td>Primary trigger level in technical units</td>
</tr>
<tr>
<td>dSecondaryLevel</td>
<td>double</td>
<td>Secondary trigger level in technical units</td>
</tr>
<tr>
<td>iDirection</td>
<td>signed32</td>
<td>Direction of primary level. Direction of secondary level is opposite by default.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTD_RisingEdge = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTD_FallingEdge = 1</td>
</tr>
</tbody>
</table>

Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example

The following example demonstrates the use of the TriggerSettings functions.

```c
char szMainframeName[64];
char szRecorderName[64];
char szChannelName[64];
signed32 iTTriggerMode, iDirection;
double dPLevel, dSLevel;

// load names -> use index only
strcpy(szMainframeName, "$1$`);
strcpy(szRecorderName, "$1$`);
strcpy(szChannelName, "$1$`);

// Get settings from 1st channel, 1st recorder, 1st mainframe
GetTriggerSettings(szMainframeName, szRecorderName,
        szChannelName, &iTTriggerMode, &dPLevel, &dSLevel,
```
3.5.3 [Set][Get]TimebaseSettings

For each recorder you can set the sample rate. Depending on the capabilities of the recorder and the selected storage mode you can set a high sample rate and a low sample rate. E.g. for sweep acquisitions the high rate is used.

**Syntax**

SetTimebaseSettings(szMainframe, szRecorder, dHighRate, dLowRate)
GetTimebaseSettings(szMainframe, szRecorder, &dHighRate, &dLowRate)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>dHighRate</td>
<td>double</td>
<td>High (fast) sample rate in Hz</td>
</tr>
<tr>
<td>dLowRate</td>
<td>double</td>
<td>Low (slow) sample rate in Hz</td>
</tr>
</tbody>
</table>

**Return values**

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

**Example**
The following example demonstrates the use of the TimebaseSettings functions.

```c
char szMainframeName[64];
char szRecorderName[64];
double dHighRate, dLowRate;

// load names -> use index only
strcpy(szMainframeName, "$1$" );
strcpy(szRecorderName, "$1$" );

// Get settings from 1st recorder, 1st mainframe
GetTimebaseSettings(szMainframeName, szRecorderName, 
&dHighRate, &dLowRate);

printf("High rate = %f, Low rate = %f", dHighRate, dLowRate);
// Now do new settings
```
dHighRate = 25000; // 25 kHz
dLowRate = 5000; // 5 kHz

SetTimebaseSettings(szMainframeName, szRecorderName, dHighRate, dLowRate);

3.5.4 [Set][Get]StorageSettings

The storage settings define what part of the recording to store:

- **None** - although you are recording, nothing is stored
- **Sweeps** - triggered segments at high sample rate are stored. This can be in transient (sweep) recording mode, but also in continuous recording mode when only the triggered segments are stored.
- **Continuous** - the continuous data stream at low sample rate is stored
- **Dual** - When using dual timebase (slow-fast-slow), both the triggered segments and the continuous data are stored each with their own sample rate.

Syntax  
SetStorageSettings(szMainframe, szRecorder, iStorageMode)
GetStorageSettings(szMainframe, szRecorder, &iStorageMode)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>iStorageMode</td>
<td>signed32</td>
<td>Storage mode of recorder:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_None = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Sweeps = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Continuous = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Dual = 3</td>
</tr>
</tbody>
</table>

Return values
If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.
Example
The following example demonstrates the use of the StorageModeSettings functions.

```c
char szMainframeName[64];
char szRecorderName[64];
signed32 iStorageMode;

// load names --> use index only
strcpy(szMainframeName, "$1$);
strcpy(szRecorderName, "$1$);
// Get settings from 1st recorder, 1st mainframe
GetStorageSettings(szMainframeName, szRecorderName,
   &iStorageMode);
printf("Storage mode = %d", iStorageMode);
// Set mode to off, i.e. no storage
SetStorageSettings(szMainframeName, szRecorderName, STM_None);
```

3.5.5 [Set][Get]SweepSettings

The SweepSettings function is used to set sweep-related parameters:

- **Length** - the length of the sweep.
- **Mode** - a sweep can have no pre-trigger, can have pre-trigger, or has a delayed trigger point.
- **Trigger position** - when in pre-trigger mode, the trigger position must be set.
- **Delay** - when in delayed trigger mode, the delay must be set.
- **Count enabled** - by default only one sweep is recorded. To record more sweeps after each other you must enable the sweep count.
- **Count** - when the sweep count is enabled the count defines the number of sweeps to record.

Syntax

- `SetSweepSettings(szMainframe, szRecorder, iSweepLength, iSweepMode, dTriggerPosition, dDelay, iSweepCountEnabled, iSweepCount )`
- `GetSweepSettings(szMainframe, szRecorder, &iSweepLength, &iSweepMode, &dTriggerPosition, &dDelay, &iSweepCountEnabled, &iSweepCount )`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
</tbody>
</table>
Perception RPC interface

<table>
<thead>
<tr>
<th>iSweepLength</th>
<th>signed32</th>
<th>Length of the sweep in samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSweepMode</td>
<td>signed32</td>
<td>Defines the sweep mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SWM_Normal = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SWM_PreTrigger = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SWM_Delayed = 2</td>
</tr>
<tr>
<td>dTriggerPosition</td>
<td>double</td>
<td>Position of the trigger within</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the sweep when in pre-trigger</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mode expressed as a percentage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the sweep length.</td>
</tr>
<tr>
<td>dDelay</td>
<td>double</td>
<td>Trigger delay when in delayed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sweep mode in seconds.</td>
</tr>
<tr>
<td>iSweepCountEnabled</td>
<td>signed32</td>
<td>Sweep count enable:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCE_No = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCE_Yes = 1</td>
</tr>
<tr>
<td>iSweepCount</td>
<td>signed32</td>
<td>Number of sweeps when sweep</td>
</tr>
<tr>
<td></td>
<td></td>
<td>count enabled.</td>
</tr>
</tbody>
</table>

Return values
If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

Example
The following example demonstrates the use of the SweepSettings functions.

```c
char szMainframeName[64];
char szRecorderName[64];
signed32 iSweepLength, iSweepMode;
signed32 iSweepCountEnabled, iSweepCount;
double dTriggerPosition, dDelay;

// load names -> use index only
strcpy(szMainframeName, "$1$);
strcpy(szRecorderName, "$1$);

// Get settings
GetSweepSettings(szMainframeName, szRecorderName, &iSweepLength,
                 &iSweepMode, &dTriggerPosition, &dDelay,
                 &iSweepCountEnabled, &iSweepCount);

// Print settings
printf("Mode: %d, Length: %d, Position: %f, Delay: %f, Enable: %d, Count: %d",
       iSweepMode, iSweepLength, dTriggerPosition, dDelay, iSweepCountEnabled,
       iSweepCount);

// Modify settings
SetSweepSettings(szMainframeName, szRecorderName, 10000,
                 SWM_PreTrigger, 25, 0, SCE_Yes, 10);
```
3.5.6 [Set][Get]ContinuousSettings

The ContinuousSettings function is used to set parameters that are related to continuous recording:

- **Mode** - a continuous recording can stop when a trigger occurs, when the storage location is full, when a manual command is issued or when a specified amount of data is recorded.

- **History** - defines the amount of 'pre-trigger' in a continuous recording.

- **Post Trigger** - defines the amount of 'post-trigger' in a continuous recording.

**Syntax**

```c
SetContinuousSettings(szMainframe, szRecorder, iRecordingMode, dRecordingTimeHistory, dPostTriggerRecordingTime)
```

```c
GetContinuousSettings(szMainframe, szRecorder, &iRecordingMode, &dRecordingTimeHistory, &dPostTriggerRecordingTime)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>iRecordingMode</td>
<td>signed32</td>
<td>Defines the continuous recording mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_StopOnFull = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_CircularRecording = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_LimitedRecording = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_StopOnTrigger = 3</td>
</tr>
<tr>
<td>dRecordingTimeHistory</td>
<td>double</td>
<td>Pre-trigger part of the continuous recording in seconds.</td>
</tr>
<tr>
<td>dPostTriggerRecordingTime</td>
<td>double</td>
<td>Posttrigger part of the continuous recording in seconds.</td>
</tr>
</tbody>
</table>

**Return values**

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the ContinuousSettings functions.

```c
char szMainframeName[64];
char szRecorderName[64];
signed32 iRecordingMode;
double dRecordingTimeHistory, dPostTriggerRecordingTime;

// load names -> use index only
strcpy(szMainframeName, "$1$"血液);
strcpy(szRecorderName, "$1$"血液);

// Get settings
GetContinuousSettings(szMainframeName, szRecorderName,
```
3.5.7 Retrieve hardware names

Three functions are provided that retrieve the actual name of a mainframe, recorder or channel, based on an index. You can use these functions to find the actual names, but also to find out how many mainframes, recorders and channels are available.

Three functions are provided:

- GetMainframeName to retrieve the name of a mainframe
- GetRecorderName to retrieve the name of a recorder
- GetChannelName to retrieve the name of a channel

Syntax

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetMainframeName(szIndex, szMainframe)</td>
<td>Retrieve the name of a mainframe</td>
</tr>
<tr>
<td>GetRecorderName(szMainframe, szIndex, szRecorder)</td>
<td>Retrieve the name of a recorder</td>
</tr>
<tr>
<td>GetChannelName(szMainframe, szRecorder, szIndex, szChannel)</td>
<td>Retrieve the name of a channel</td>
</tr>
</tbody>
</table>

Parameter Type Description
---
| szMainframe | string | Name or index of mainframe |
| szRecorder | string | Name or index of recorder |
| szChannel | string | Name of channel |
| szIndex | string | Index |

Return values none

Example See the example in the following section that also demonstrates the use of the Get...Name functions.

3.5.8 Example to the hardware settings functions

The following code is an example that combines the use of the Get...Name functions and the retrieval of various settings into a single console application.

```
#include "stdafx.h"

// program 4: name retrieval - author: Cees J de Vries
// (c) 2005 LDS Test and Measurement

#include "stdafx.h" // standard stuff
```
#include "rpcsupport.h"    // basic rpc support like initRPC
#include <iostream>        // basic I/O
using namespace std;

#define MAX_NAMESTRING 128
#define MAX_HARDWARENAME 256

char szSystem[MAX_NAMESTRING];
char szMainframe[MAX_HARDWARENAME];
char szRecorder[MAX_HARDWARENAME];
char szChannel[MAX_HARDWARENAME];
char szIndex[MAX_HARDWARENAME];
int iMainframeCount, iRecorderCount, iChannelCount;
char cAnyKey;
signed32 iStorageMode, iEnabled;

int _tmain(int argc, char* argv[]) {  
    // use the computer name or IP-address
    strcpy(szSystem, "10.134.144.123");

    // Initialise RPC, continue when successful
    if (InitRPC(szSystem)) {
        // loop through mainframes
        iMainframeCount = 0;
        do {
            iMainframeCount++;
            strcpy(szMainframe, "");
            // create index string from counter
            sprintf(szIndex, "$%d$", iMainframeCount);
            GetMainframeName(szIndex, szMainframe);
            if (strlen(szMainframe) != 0) {
                cout << endl << "Mainframe  " << iMainframeCount << ": " << szMainframe << endl;
                // loop through recorders
                iRecorderCount = 0;
                do {
                    iRecorderCount++;
                    strcpy(szRecorder, "");
                    // create index string from counter
                    sprintf(szIndex, "$%d$", iRecorderCount);
                    GetRecorderName(szMainframe, szIndex, szRecorder);
                    if (strlen(szRecorder) != 0) {
                        // also get storage mode
                        GetStorageSettings(szMainframe, szRecorder, &iStorageMode);
                        cout << endl << "  Recorder  " << iRecorderCount << ": " << szRecorder << " - Storage: ";
                    }
                }
            }
        }
    }
    return 0;
}
if (iStorageMode == 0) cout
  << "No storage";
if (iStorageMode == 1) cout
  << "Sweeps only";
if (iStorageMode == 2) cout
  << "Continuous data";
if (iStorageMode == 3)
  cout << "Both slow and fast rate";
cout << endl << endl;
// loop through channels
iChannelCount = 0;
do
{
  iChannelCount++;
  strcpy(szChannel, "") ;
  // create index string
  // from counter
  sprintf(szIndex,
    "$%d$",iChannelCount);
  GetChannelName( szMainframe, szRecorder,
                  szIndex, szChannel);
  if (strlen(szChannel) != 0)
    {
    // also get enabled
    // status
    GetChannelEnabled( szMainframe,szRecorder,
                       szChannel, &iEnabled);
    cout << " Channel "
      << iChannelCount
      << ": "      << szChannel;
    cout << " - Output Enabled:
          "      << (iEnabled ? 
                            "Yes" : "No")      << endl;
    }
  }
  while (strlen(szChannel) !=0);
} }
while (strlen(szRecorder) != 0); }
} while (strlen(szMainframe) != 0);

// done
cin >> cAnyKey;
return FALSE;

The console output could look like this:
3.6 Data retrieval

Use these functions to fetch a data stream from available waveforms. For each waveform a subscription is needed in order to get the data. This is the same live data that is on a Perception display, i.e. reduced using min-max pairs.

3.6.1 GetAvailableWaveforms

With this function you can retrieve all available waveform names.

Syntax

GetAvailableWaveforms(&pwaveForms)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pwaveForms</td>
<td>pointer</td>
<td>pointer to a string array that holds the search</td>
</tr>
</tbody>
</table>

Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.

In *PerceptionRPC.idl* the *stringArray* structure is defined as follows:

```c
typedef [ptr, string] char * string_t;

typedef struct {
    signed32 stringCount;
    [size_is(stringCount)] string_t strings[];
} stringArray
```
Example

The following example demonstrates the use of the GetFileNames function.

```c++
int i;
stringArray * pwaveForms = new stringArray;
if (GetAvailableWaveforms(&pwaveForms) == 0)
{
    cout << endl << "Available waveforms:" << endl << endl;
    for (i = 0 ; i < pwaveForms->stringCount ; i++)
    {
        cout << "    " << pwaveForms->strings[i] << endl;
    }
}
```

When this example is combined with the example for the hardware settings, the output could look like this:

```
Mainframe 1: sim_pc_at_work
    Recorder 1: HiSpeed Recorder - Storage: Continuous data
    Channel 1: TP #01 R - Output Enabled: No
    Channel 2: TP #02 L - Output Enabled: No
    Channel 3: Throttle - Output Enabled: Yes
    Channel 4: Vacuum - Output Enabled: Yes

    Recorder 2: LoSpeed Recorder - Storage: Sweets only
    Channel 1: Ch B1 - Output Enabled: Yes
    Channel 2: Ch B2 - Output Enabled: Yes
    Channel 3: Ch B3 - Output Enabled: Yes
    Channel 4: Ch B4 - Output Enabled: Yes

Available waveforms for display:
    Active.sim_pc_at_work.HiSpeedRecorder.Throttle
    Active.sim_pc_at_work.HiSpeedRecorder.ValueCmd
    Active.sim_pc_at_work.LoSpeedRecorder.Ch_B1
    Active.sim_pc_at_work.LoSpeedRecorder.Ch_B2
    Active.sim_pc_at_work.LoSpeedRecorder.Ch_B3
    Active.sim_pc_at_work.LoSpeedRecorder.Ch_B4
```
### 3.6.2 SubscribeLiveWaveform

Before you can actually fetch data from a waveform you need to subscribe to that waveform. When you subscribe to a waveform you get a Subscriber ID that can is used to retrieve the data.

**Syntax**

```
SubscribeLiveWaveform(szWaveform, iPairsPerBuffer, dTimePerBuffer, iTriggerMode, dTimeShift, &iSubscriberID)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szWaveform</td>
<td>string</td>
<td>Name of waveform</td>
</tr>
<tr>
<td>iPairsPerBuffer</td>
<td>signed32</td>
<td>Number of min-max pairs per buffer (segment)</td>
</tr>
<tr>
<td>dTimePerBuffer</td>
<td>double</td>
<td>Time per buffer in seconds</td>
</tr>
<tr>
<td>iTriggerMode</td>
<td>signed32</td>
<td>Display stabilizer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMTM_FreeRunning = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMTM_TrigOnly = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMTM_TrigOrStabilizer = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMTM_TrigOrTimeout = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMTM_TrigOrStabilizeOrTimeout = 7</td>
</tr>
<tr>
<td>dTimeShift</td>
<td>double</td>
<td>Time shift of trigger position in percentage of the Subscriber ID</td>
</tr>
<tr>
<td>iSubscriberID</td>
<td>signed32</td>
<td>Returned ID</td>
</tr>
</tbody>
</table>

**Return values**

- If the function succeeds, the return value is zero.
- If the function fails, the return value is nonzero.

**Example**

The following example demonstrates the use of the SubscribeLiveWaveform function.

```c
char szWaveform[128];

strcpy (szWaveform, "Active.sim_pc.LoSpeed_Recorder.Ch_B1");

if (SubscribeLiveWaveform(szWaveform, 100, 0.5, WMTM_FreeRunning, 0, &iSubscriberID) == 0)
{
    // do your stuff
}
```
3.6.2.1 More on data buffers

Live data is supplied in pieces called buffers or segments. When you subscribe to the live data you need to specify the length of such a piece. This length is specified in seconds. E.g. you can subscribe to get pieces of data that equal 0.5 seconds or 1 second. You also need to specify how many data points you want to be within that segment. A data point here is defined as one min-max pair. When acquisition is active, live data is processed to deliver the required data in buffers. It is the responsibility of the client program to retrieve the data fast enough. Up to 10 buffers can be stored in the system before overrun occurs.

The Waveform Monitor Trigger Mode can be used to deliver pieces of data that are synchronised to a trigger level. For more details refer to the Perception manual. When a mode is selected that uses a trigger you can specify a shift for that trigger point within the data buffer. The following diagram is a graphical example of some of the trigger modes.
3.6.3 UnsubscribeLiveWaveform

When you are done using the data from a subscribed waveform you must unsubscribe.

Syntax

UnsubscribeLiveWaveform(iSubscriberID)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSubscriberID</td>
<td>signed32</td>
<td>ID as returned by the corresponding SubscribeLiveWaveform function.</td>
</tr>
</tbody>
</table>

Return values

None

Example

The following example demonstrates the use of the UnsubscribeLiveWaveform function.

```c
char szWaveform[128];
strcpy(szWaveform, "Active.sim_pc.LoSpeed_Recorder.Ch_B1");
if (SubscribeLiveWaveform(szWaveform, 100, 0.5, WMTM_FreeRunning, 0, &iSubscriberID) == 0) {
    // do your stuff
    // release waveform when done
    UnsubscribeLiveWaveform(iSubscriberID);
}
```

3.6.4 GetData

The GetData function is used to actually fetch data from a data stream of a subscribed waveform. You can only fetch data when the acquisition hardware is recording or is in pause mode. In each function call you receive data that is available at that time. For more data you need subsequent function calls, typically within a loop.

Syntax

GetData(iSubscriberID, iMaxSegmentCount, iMaxBinCount, pSegmentInfo, &iResultSegmentCount, pBins, &iResultBinCount)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSubscriberID</td>
<td>signed32</td>
<td>ID as returned by the corresponding SubscribeLiveWaveform function.</td>
</tr>
<tr>
<td>iMaxSegmentCount</td>
<td>signed32</td>
<td>Maximum number of segments to get. Limited to 10 by Perception.</td>
</tr>
<tr>
<td>iMaxBinCount</td>
<td>signed32</td>
<td>The maximum number of min-max pairs to receive.</td>
</tr>
<tr>
<td>pSegmentInfo</td>
<td>pointer</td>
<td>Pointer to a SegmentInfo array.</td>
</tr>
</tbody>
</table>
**Perception RPC interface**

<table>
<thead>
<tr>
<th><strong>iResultSegmentCount</strong></th>
<th><strong>signed32</strong></th>
<th>Number of segments returned by the function call.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pBins</strong></td>
<td><strong>pointer</strong></td>
<td>Pointer to the ReducedDataBin array that contains the returned data.</td>
</tr>
<tr>
<td><strong>iResultBinCount</strong></td>
<td><strong>signed32</strong></td>
<td>The number of bins returned by the function call.</td>
</tr>
</tbody>
</table>

In Perception.idl the SegmentInfo structure and the ReducedDataBin structure are defined as follows:

```cpp
typedef struct {
    signed32 nSegmentLength;
    double dStartTime;
    double dEndTime;
} SegmentInfo;

typedef struct {
    float fValue1;
    float fValue2;
} ReducedDataBin;
```

**Return values**  
None

**Example**  
Refer to the example in the following section for the use of the GetData function.

### 3.6.5 Example to the waveform data functions

The following code example shows how to get the available waveforms, subscribe to a waveform and retrieve the data.

```cpp
//
// program 5: fetch live data - author: Cees J de Vries
// (c) 2005 LDS Test and Measurement
//
#include "stdafx.h"    // standard stuff
#include "rpcsupport.h"    // basic rpc support like initRPC
#include <iostream>    // basic I/O
using namespace std;

#define MAX_NAMESTRING 128
#define MAX_SEGMENTREQUEST 10
#define MAX_PAIRSPERBUFFER 4
#define MAX_BINREQUEST (MAX_SEGMENTREQUEST * MAX_PAIRSPERBUFFER)

char szSystem[MAX_NAMESTRING];
char cAnyKey;
int i, j;
```
int _tmain(int argc, char* argv[])
{
    // use the computer name or IP-address
    strcpy(szSystem, "10.134.144.123");
    stringArray * pwaveForms = new stringArray;
    SegmentInfo * pSegmentInfo = new SegmentInfo
        MAX_SEGMENTREQUEST];
    ReducedDataBin * pBins = new ReducedDataBin
        MAX_BINREQUEST];

    // Initialise RPC, continue when successful
    if (InitRPC(szSystem))
    {
        // fetch available waveforms
        if (GetAvailableWaveforms(pwaveForms) == 0)
        {
            cout << endl << "Available waveforms for
            display:" << endl << endl;
            for (i = 0 ; i < pwaveForms->stringCount ; i++)
            {
                cout << " " << pwaveForms->strings[i] << endl;
            }
            // connect to the first available waveform
            if (SubscribeLiveWaveform(pwaveForms->strings[0],
                MAX_PAIRSPERBUFFER, 0.5,
                WMTM_FreeRunning, 0, &iSubscriberID) == 0)
            {
                // wait until there is data
                Sleep (2500);
                iBinOffset = 0;
                // Get data
                GetData(iSubscriberID, MAX_SEGMENTREQUEST, 
                    MAX_BINREQUEST, pSegmentInfo, 
                    &iResultSegmentCount, pBins, &iResultBinCount);
                // for each returned segment print the info results
                if (iResultSegmentCount > 0)
                {
                    cout << endl;
                    for (i = 0 ; i < iResultSegmentCount ; i++)
                    {
                        cout << "Segment " << i + 1
                        " Info -> ";
                        cout << pSegmentInfo[i].nSegmentLength
                        " ";
                        printf("%e - ", 
                            pSegmentInfo[i].dStartTime);
                        printf("%e\n", 
                            pSegmentInfo[i].dStartTime);
Before you run this program make sure Perception is running, hardware is connected and a recording is active.
When you run this program the result could look like this:

![Screenshot of available waveforms for display]

3.6.5.1 More on live data

When you subscribe to a waveform, a buffer of specified length with the specified number of
min-max pairs is prepared. When this buffer is filled, an additional buffer is created, up to a
maximum of ten buffers. When you fetch the contents of one or more buffers, the buffer queue
is reduced by the number of buffers you retrieve.

In the aforementioned example the following three constants are defined:

```c
#define MAX_SEGMENTREQUEST 10
#define MAX_PAIRSPERBUFFER 4
#define MAX_BINREQUEST (MAX_SEGMENTREQUEST * MAX_PAIRSPERBUFFER)
```

The MAX_SEGMENTREQUEST defines how many segments you want to retrieve in one go at
maximum. Valid values range from 1 (one) to 10 (ten). What you get depends on how many
filled buffers are available at the time of request.

Typically requesting the data will be done in an (almost) endless loop. The buffer mechanism
is then used for the asynchronous transfer of synchronised data.

For each buffer you define the number of min-max pairs MAX_PAIRSPERBUFFER. This value
is used in the SubscribeLiveWaveform function. The number of min-max pairs can be very low
(1 to 10) but also very high (> 1000). This all depends on the processing capabilities of the
client software: it is the client’s responsibility to fetch the data fast enough to prevent overrun
and to do the other things too.

In the GetData function the value MAX_BINREQUEST is used to allocate memory for the
transfer.
For each segment that is returned there is an entry in the SegmentInfo array. Each entry is a structure that has information on the number of min-max pairs in that segment, the time of the first pair and the time of the last pair.

All the waveform data is returned in an array of ReducedDataBin structures. The data is stored sequentially in the array.

Refer to the diagram for a graphical explanation.

Since the data is stored sequentially, the first pair of the first segment has index 0 and the first pair of the second segment has index length of first segment + 0, etc. Note that, although usually the segment lengths will be the same, the length of the retrieved segments within a single function call may have different lengths.

3.6.6 Understanding min-max pairs

The data is supplied as a series of min-max pairs. Each pair gives the minimum and maximum value that has been encountered in a specific interval. Assume a sample rate of 10000 samples per second and a requested 100 pairs in a buffer of 0.5 seconds. This will yield one min-max pair per 50 samples, i.e. the pair gives you the minimum and maximum values encountered in each subsequent 50 samples.

To draw data correctly using min-max pairs you will have to know your math. Min-max pairs are only effective when you use one pair for each X-axis pixel on the screen. For each X-axis pixel you draw a line from the minimum value to the maximum value.
This could result in gaps however, since the minimum value of a pair can be greater than the maximum of a previous pair. Also the maximum value of a pair can be less than the minimum value of the previous pair. To make things even more complicated it is also possible that small values (less than one pixel) are not drawn. The diagram shows some of the possibilities.

In your program you will need to take these deviations into consideration to make sure that the display on the screen is an actual representation of the data.
3.6.7 [Set][Get]SCInputRangeSettings

The input range settings are used to set span, offset and other input related parameters of an input channel:

- **Mask** - Bitmask indicating which settings will be applied/modified or read. This is a combination of InputRangeSI enumeration as detailed in the table below.

- **UpperValue** - Upper (display) boundary of the input range in user units.

- **LowerValue** - Lower (display) boundary of the input range in user units.

- **ActualUpperValue** - Hardware amplifier upper boundary in user units.

- **ActualLowerValue** - Hardware amplifier lower boundary in user units.

- **Span** - Measurement range in user units.

- **Offset** - Deviation from midscale in user units.

- **Gain** - Gain of bridge amplifier.

- **VariableGain** - Allow for non-standard gain factors.

**Syntax**

SetSCInputRangeSettings(szMainframe, szRecorder, szChannel, iMask, dUpper, dLower, dActualUpper, dActualLower, dSpan, dOffset, dGain, iVariableRange)

GetSCInputRangeSettings(szMainframe, szRecorder, szChannel, iMask, &dUpper, &dLower, &dActualUpper, &dActualLower, &dSpan, &dOffset, &dGain, &iVariableRange)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>szMainframe</td>
<td>string</td>
<td>Name or index of mainframe</td>
</tr>
<tr>
<td>szRecorder</td>
<td>string</td>
<td>Name or index of recorder</td>
</tr>
<tr>
<td>szChannel</td>
<td>string</td>
<td>Name or index of channel</td>
</tr>
<tr>
<td>iMask</td>
<td>signed32</td>
<td>Combine the following enumerations to define</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the settings that will be applied (modified) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_Upper = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_Lower = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_ActUpper = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_ActLower = 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_Span = 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_Offset = 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>InputRangeSI_Gain = 64</td>
</tr>
<tr>
<td>dUpper</td>
<td>double</td>
<td>Upper display boundary</td>
</tr>
<tr>
<td>dLower</td>
<td>double</td>
<td>Lower display boundary</td>
</tr>
</tbody>
</table>
**Perception RPC interface**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dActualUpper</td>
<td>double</td>
<td>Hardware amplifier upper boundary in user units (read-only)</td>
</tr>
<tr>
<td>dActualLower</td>
<td>double</td>
<td>Hardware amplifier lower boundary in user units (read-only)</td>
</tr>
<tr>
<td>dSpan</td>
<td>double</td>
<td>Measurement range in user units</td>
</tr>
<tr>
<td>dOffset</td>
<td>double</td>
<td>Deviation from midscale in user units</td>
</tr>
<tr>
<td>dGain</td>
<td>double</td>
<td>Gain factor of bridge amplifier</td>
</tr>
<tr>
<td>iVariableRange</td>
<td>signed32</td>
<td>Allow for non-standard gain factor:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False / No = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>True / Yes = 1</td>
</tr>
</tbody>
</table>

**Return values**

If the function succeeds, the return value is zero.

If the function fails, the return value is nonzero.

### 3.6.8 [Set][Get]RecStorageSettings

The recorder storage settings are used to set storage/acquisition related parameters of a recorder:

- **Mask** - Bitmask indicating which settings will be applied/modified or read. This is a combination of RecorderStorageSI enumeration as detailed in the table below.

- **StorageMode** - One of the storage mode enumerations as detailed in the table below.

- **RecordingMode** - One of the recording mode enumerations as detailed in the table below.

- **RecordingTimeHistory** - Size in seconds of buffer in circular recording modes.

- **PostTriggerRecordingTime** - Amount of data in seconds that will be stored after a trigger occurred.

**Syntax**

SetRecStorageSettings (szMainframe, szRecorder, iMask, eStorageMode, eRecordingMode, dRecordingTimeHistory, dPostTriggerRecordingTime)

GetRecStorageSettings (szMainframe, szRecorder, iMask, &eStorageMode, &eRecordingMode, &dRecordingTimeHistory, &dPostTriggerRecordingTime)
### Parameter Descriptions

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iMask</td>
<td>signed32</td>
<td>Combine the following enumerations to define the settings that will be applied (modified) or read:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecorderStorageSI_StoMode = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecorderStorageSI_RecMode = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecorderStorageSI_History = 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RecorderStorageSI_PostTrigger = 8</td>
</tr>
<tr>
<td>eStorageMode</td>
<td>signed32</td>
<td>Select one of the storage modes as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_None = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Sweeps = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Continuous = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_Dual = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STM_ABA = 4 (as from V3.04)</td>
</tr>
<tr>
<td>eRecordingMode</td>
<td>signed32</td>
<td>Select one of the acquisition modes as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_StopOnFull = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_CircularRecording = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_LimitedRecording = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RM_StopOnTrigger = 3</td>
</tr>
<tr>
<td>dRecordingTimeHistory</td>
<td>double</td>
<td>Size in seconds of buffer in circular recording modes</td>
</tr>
<tr>
<td>dPostTrigger~</td>
<td>double</td>
<td>Amount of data in seconds that will be stored after a trigger occurred</td>
</tr>
<tr>
<td>RecordingTime</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Return values

If the function succeeds, the return value is zero.
If the function fails, the return value is nonzero.
4 Appendix: Using RPC-COM wrapper and C#

A COM wrapper has been designed by HBM to simplify the creation of your own Perception controlling applications at a Windows based PC using the RPC technology. This COM wrapper can be used from a lot of different programming languages, in this appendix we will focus to C#.

4.1 Introduction

C# (pronounced see sharp) is an object-oriented programming language developed by Microsoft as part of their .NET initiative, and later approved as a standard by ECMA and ISO. C# has a procedural, object-oriented syntax based on several other programming languages (most notably Delphi and Java) with a particular emphasis on simplification. The de facto standard implementation of the C# language is Microsoft C# compiler, included in every installation of .NET Framework. The original .NET Framework distributions from Microsoft included several language-to-IL compilers, including the two primary languages: C# and Visual Basic. The bulk of the differences between C# and VB .NET from a technical perspective are syntactic sugar. That is, most of the features are in both languages, but some things are easier to do in one language than another. It should be noted that all .NET programming languages share the same runtime engine, and when compiled produced binaries that are seamlessly compatible with other .NET programming languages, including cross language inheritance, exception handling, and debugging.

In this appendix we will give various examples of how to use the RPC-COM wrapper with C#. The code is developed in the Microsoft Visual Studio development environment (2005 edition).

The configuration looks like the picture below.
4.2 Getting started

- Install latest Perception build, this build also installs the RPC client, server and COM wrapper software on your PC.

- If you want to work on a remote PC you do not need to install Perception, but you should install “Perception COM-RPC”, this installs the COM-RPC wrapper (PerceptionComRpc.exe), help and demo files. If you work on the same machine as Perception you do not need to install this software, however if you like to have the demo files you have to install it. This install can be found on the Perception install CD.

- At a remote PC you do not need a Perception dongle, the only thing you need is the file: "PerceptionComRpc.exe". This file contains the COM/RPC wrapper and is the same as the one installed at the PC where Perception is installed (See folder C:\Program Files\Common Files\HBM\Components).

- The Perception RPC server is only available if the option Remote API of the Perception HASP®4 USB Token has been enabled. This option is listed as Remote API in the Perception menu: Help > About Perception > More... > Options page.

- Be sure the Perception RPC Server is enabled. You should go to the Preferences dialog and enable the RPC server.
4.3 Creating your first RPC-COM client application

- Start Microsoft Visual Studio 2005 or any newer version. You also can use the free of charge Express editions. These free versions have limited functionality but this is no problem for writing Perception Client applications.

- Create a new project using the "Windows Application" template

- Add a reference to PerceptionComRpc. To do this right click at references and add a reference to the COM component “Perception COM Support”

If "Perception Com Support" does not show up in the COM tab of the Add Reference Dialog, then register the "PerceptionComRpc.exe" manually. Most probably this file is located in the folder: C:\Program Files\Common Files\HBM\Components. To register this file you should run the following command:

"C:\Program Files\Common Files\HBM\Components\PerceptionComRpc.exe /RegServer"

- Add a button and two labels to the form:
• Add the following code:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using PerceptionComRpc;

namespace MyFirstCOMClient
{
    public partial class Form1 : Form
    {
        // Define a private member to hold a reference to
        // Perception
        // All the communication to Perception will be done via
        // this member
        private PerceptionCOM m_Perception = null;

        public Form1()
        {
            InitializeComponent();
            // Create an instance of the Perception com server
            m_Perception = new PerceptionCOM();
        }

        private void btnConnect_Click(object sender, EventArgs e)
        {
            // Set the address of the Perception Server machine
            // "localhost" is pointing to your own PC.
            // If however the Perception server is running on
            // another machine then
            // you have to enter the computer name or IP-address of
            // this PC
            m_Perception.SetServerAddress("localhost");
        }
    }
}
```
m_Perception.ConnectToServer();

// Get the acquisition state to check if we have
// communication
ShowAcquisitionState();

private void ShowAcquisitionState()
{
    int Run;
    int OneShot;
    int Stop;
    int Pause;
    int Idle;
    m_Perception.GetAcquisitionState(out Run, out OneShot, 
    out Stop, out Pause, out Idle);
    label1.Text = "Start: " + Run.ToString() + " OneShot: " 
    + OneShot.ToString() + 
    " Stop: " + Stop.ToString() + " Pause: " + 
    Pause.ToString() + 
    " Idle: " + Idle.ToString();
}

Some notes related to the above example:

Line 8 - We tell the system to use the COM wrapper.

Line 16 - We define a private member variable to hold a reference to Perception, all the
communication to Perception will be done via this member.

Line 22 - Create an instance of the Perception COM server.

Line 25 - Event procedure hooked to the connect button click event.

Line 31 - Set the address of the Perception Server machine; "localhost" means that you are
referring to a Perception RPC server at the same machine from where you are running the
client program. If Perception however is running at a different PC then you have to enter the
computer name or its IP-address.

Line 32 - Making a connection to the Perception server.

Line 35 - Call a private procedure to show the acquisition state of perception, we do this to
proof that we have set up a RPC communication link.

Lines 38-50 Showing the acquisition state, first we ask for the acquisition state, Perception
returns how many recorders are in a specific state.
Use the help file to get more information, it shows the following for the function GetAcquisitionState():

```csharp
using PerceptionCOM;

private PerceptionCOM m_Percception = new PerceptionCOM();

m_Percception.SetServerAddress("localhost");

private void ShowAcquisitionState()
{
    // Run
    m_Percception.GetAcquisitionState(out Run, out OneShot, out Stop, out Pause, out Idle);
    if(Run) { Console.WriteLine("Run"); } // OneShot; Stop; Pause; Idle
    if(OneShot) { Console.WriteLine("OneShot"); } // Run; Stop; Pause; Idle
    if(Stop) { Console.WriteLine("Stop"); } // Run; OneShot; Pause; Idle
    if(Pause) { Console.WriteLine("Pause"); } // Run; OneShot; Stop; Idle
    if(Idle) { Console.WriteLine("Idle"); } // Run; OneShot; Stop; Pause
}
```

This help file contains all the details of the RPC interface, it also contains many examples.
4.4 Extending the application with acquisition control

We now go extend the previous example with some buttons to be able to start, stop, single shot, pause and manual trigger the remote Perception

- Open the previous project
- Add the following buttons:

![My First COM Client](image)

- The **Disconnect** button calls the following code:

```csharp
private void btnDisconnect_Click(object sender, EventArgs e)
{
    m_Perception.DisconnectFromServer();
    label1.Text = "Disconnected";
}
```

- The **Start** button calls the following code:

```csharp
private void btnStart_Click(object sender, EventArgs e)
{
    m_Perception.Start();
}
```

- The **Stop** button calls the following code:

```csharp
private void btnStop_Click(object sender, EventArgs e)
{
    m_Perception.Stop();
}
```

- The **S/Shot** button calls the following code:

```csharp
private void btnSingleShot_Click(object sender, EventArgs e)
{
    m_Perception.OneShot();
}
```
• The **Pause** button calls the following code:

```csharp
private void btnPause_Click(object sender, EventArgs e)
{
    m_Perception.Pause();
}
```

• The **Trigger** button calls the following code:

```csharp
private void btnTrigger_Click(object sender, EventArgs e)
{
    m_Perception.Trigger();
}
```

• The **Get Acq. State** button calls the following code:

```csharp
private void btnShowAcqState_Click(object sender, EventArgs e)
{
    ShowAcquisitionState();
}
```

This concludes our C# example.

For more information look into the help file and look at the example which comes with the installation of the COM-RPC software. The main demo for C# is called “C Sharp RPC Client” and shows a lot of the existing RPC functionality.
This demo program looks like:
measure and predict with confidence