DAPIO32 Reference Manual

DAPIO32 function and structure reference

Version 5.00

Microstar Laboratories, Inc.

Microstar Laboratories, Inc.
2265 116th Avenue N.E.
Bellevue, WA 98004
Tel: (425) 453-2345
Fax: (425) 453-3199
http: // www.mstarlabs.com

Part Number MSDAPIO32M500

Contents

1. Introduction

The Data Acquisition Processor from Microstar Laboratories is a complete data acquisition system that occupies one PCI or USB slot in a PC. Data Acquisition Processor systems are suitable for a wide range of applications in laboratory and industrial data acquisition and control.

A Data Acquisition Processor is a computer, separate from the PC, with its own operating system. The operating system on each Data Acquisition Processor is either DAPL 2000 or DAPL 3000. DAPL 2000 is an operating system that runs on most PCI Data Acquisition Processors, while DAPL 3000 is an operating system that runs on all USB Data Acquisition Processors. The DAPIO32 software interface allows a PC application to control the Data Acquisition Processor through its operating system.

About this Document

This document contains a complete reference for the DAPIO32 software interface to Data Acquisition Processors. 32-bit and 64-bit applications use the DAPIO32 interface to communicate with Data Acquisition Processors. The DAPIO32.DLL implements the DAPIO32 interface. The C++ programming language is used in this document to describe the DAPIO32 interface.

There are several implementations of DAPIO32.DLL for different environments. All of the implementations conform to this reference except where there is an explicit statement to the contrary, either in this document or in the `README.TXT` for the product that includes DAPIO32.DLL.

Several manuals, provided with each Data Acquisition Processor, contain information of interest when creating data acquisition applications:

- The hardware manual for each Data Acquisition Processor model contains information about installing and configuring DAP hardware.
- The DAPL 2000 or DAPL 3000 Manual contains a complete software reference for the DAPL operating system.
- The Applications Manual contains Data Acquisition Processor DAPL application examples.

2. DAPIO32 Overview

DAPIO32 provides the interface between applications and the Data Acquisition Processor. Applications communicate with the Data Acquisition Processor through a communication channel structure called the “communication pipe”. An application opens a handle to a pipe and then uses the handle to send and receive data through the pipe. A pipe can be opened for reading or writing only once; once opened for that purpose, the pipe is reserved for access by the application exclusively until the application closes the open handle.

Communication Pipes

DAPIO32 buffers data from and to the Data Acquisition Processor. This buffering structure is called the “communication pipe”. There are communication pipes in the DAPL operating system running on the Data Acquisition Processor; there also are communication pipes in the PC. The communication pipes on both sides are logically connected on a one-to-one basis. The pipes in the PC can be viewed as extensions to the communication pipes on the Data Acquisition Processor. Each connected pair of pipes form a communication channel between the PC application and the Data Acquisition Processor.

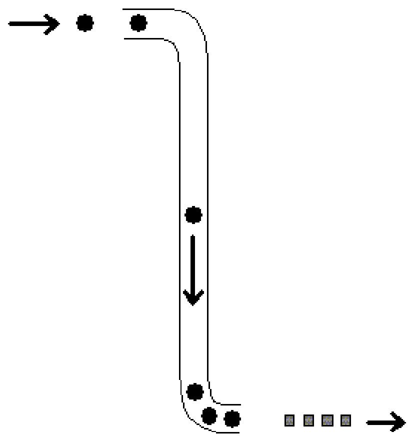
There are four default communication pipes provided for each Data Acquisition Processor.

- `$SysIn` is used for text commands from the host to the DAPL system on the DAP board.
- `$SysOut` is used for text messages returned from the DAPL system to the application on the host system.
- `$BinOut` is used for binary data transfers from the DAPL system and its processing configuration to the application on the host system (typically for returning digitized signal data).
- `$BinIn` is used for binary data transfers from the application on the host system to the processing configuration in the DAPL system (typically, for generating output signals).

The four default communication channels provide most applications with sufficient communication with the Data Acquisition Processor. Additional communication pipes can be pre-configured using the DAPcell Services (the “Data Acquisition” control panel application) under Windows. Alternatively, your applications can request additional communications pipes from DAPcell (or Accel32 under Linux) using the DAPIO32 function `DapComPipeCreate`.

Buffered Data Transfers

Data transfers through buffered communication pipes are a little different from data transfers from a static object such as a file.



- The process that places data into the pipe is separate from the process that takes data out of the pipe. These processes run concurrently. Your application is subject to timing constraints of your operating system, with no hard timing guarantees.
- You cannot tell, when you send data, exactly when the receiver will take the data. You cannot tell, when you receive data, exactly when the data were sent.
- There is storage capacity within the transfer pipe, but it is finite. If the data are not taken, they will backlog somewhere.
- If you take *all* of the data that arrive through the pipe, the amount that you can receive will vary depending on how much was sent, and how much has arrived.
- If you take a *part* of the data that arrive through the pipe, some amount of other as-yet-unseen data can remain within the transfer pipe.
- The groupings of data you take from the stream can be completely different from the groupings as they went in – a hazard or an advantage.

You will need to pay particular attention to receive all data transferred, to take the data in meaningful groups interpreted as the correct data types, and to avoid waiting for data that were not yet sent.

UNC (Universal Naming Convention) Pipe Names

DAPIO32 addresses a pipe on the Data Acquisition Processor using the Universal Naming Convention (UNC). A UNC pipe name consists of three portions, a machine name, a Data Acquisition Processor name, and a pipe name. A UNC name is led by two backslashes with each component delimited by one backslash.

```
\\<Machine>\<DAP>\<Pipe>
```

A remote machine is represented by its unique network machine name. The local machine is denoted by a period. Only the DAPcell / DAPcell Local / Accel32 implementation of DAPIO32 supports remote machine names. All other implementations support only the local machine name (`\\.\`).

The Data Acquisition Processor names are pre-defined as `Dap0`, `Dap1`, ..., and `Dap(N-1)` where `N` is the number of Data Acquisition Processors installed on the system. See the documentation for each DAPIO32.DLL implementation for a description of how DAP names are assigned.

The pipe names also are pre-defined. On the Data Acquisition Processor, two communication pipes are associated with one integer number but differ in transfer directions: input or output. `$SysIn` and `$SysOut` are the default input and output pipes with the number zero while `$BinIn` and `$BinOut` are the default input and output pipes with the number one. DAPIO32 supports a maximum of 32 sets of input and output communication pipes on each Data Acquisition Processor; thus the largest number that can be associated with a pipe is 31. Except for the two default sets, all pre-defined pipe names carry both the pipe number information and the pipe direction information. Following is a list of the 32 supported sets of communication pipes:

```
$SysIn                            $SysOut
$BinIn                            $BinOut
Cp2In                             Cp2Out
Cp3In                             Cp3Out
Cp4In                             Cp4Out
        ...
Cp31In                            Cp31Out
```

Thus, the communication pipe `$SysIn` of the Data Acquisition Processor `Dap0` on the local machine is represented by the UNC name `\\.\Dap0\$SysIn`, and the pipe `$BinOut` of the Data Acquisition Processor `Dap1` on the remote machine `PC101` is referred to as `\\PC101\Dap1\$BinOut`. For example,

```
\\.\Dap0\$SysIn
\\.\Dap0\$SysOut
\\.\Dap0\$BinIn
\\.\Dap0\$BinOut
```

are the four default communication pipes on the Data Acquisition Processor `Dap0` on the local machine, and

```
\\PC101\Dap1\$SysIn
\\PC101\Dap1\$SysOut
\\PC101\Dap1\$BinIn
\\PC101\Dap1\$BinOut
```

are the four default communication pipes on the Data Acquisition Processor `Dap1` on the remote machine `PC101`.

Basic Communication

Every Data Acquisition Processor application performs the following operations:

- opens handles to Data Acquisition Processor com-pipes
- configures Data Acquisition Processor for application specific processing
- performs application specific data input/output through com-pipes
- closes handles to Data Acquisition Processor com-pipes

The following listing of example program S100.CPP shows all four of these phases in the "lifecycle" of a communication pipe connection. Following the application code listing is the DAPL configuration file `S100.DAP` that the program uses to configure the Data Acquisition Processor.

To open com-pipe handles, the S100.CPP application uses the `DapHandleOpen` service. To configure the Data Acquisition Processor, it uses the `DapConfig` service. To read data from the Data Acquisition Processor, it uses the `DapBufferGet` service. To close handles it uses the `DapHandleClose` service.

`S100.CPP` takes 100 readings from a single input channel on the Data Acquisition Processor and prints those data to the console at one line per value, scaled for a ±5 Volt analog input range. To avoid clutter, this example performs minimal error checking.

S100.CPP:

```
   #include <stdio.h>
   #include <dapio32.h>

   void main()
   {
     const  DataCount = 100;
     HDAP  hdapSysGet, hdapSysPut, hdapBinGet;
     short int  asiData[DataCount];
     float   fConverted;
     int  item;
     BOOL  isConfigured;

     // Open communication handles
     hdapSysGet =
       DapHandleOpen("\\\\.\\Dap0\\$SysOut", DAPOPEN_READ);
     hdapSysPut =
       DapHandleOpen("\\\\.\\Dap0\\$SysIn", DAPOPEN_WRITE);
     hdapBinGet =
       DapHandleOpen("\\\\.\\Dap0\\$BinOut", DAPOPEN_READ);
     // Configure the DAP
     DapInputFlush(hdapSysGet);
     DapInputFlush(hdapBinGet);
     isConfigured = DapConfig(hdapSysPut, "S100.DAP");
     // Perform the data transfers
     if  (isConfigured)
     {
       if (sizeof(asiData) ==
         DapBufferGet(hdapBinGet, sizeof(asiData), asiData))
       {
           for (item = 0; item < DataCount; item++)
           {
             fConverted = asiData[item]*5.0f/32768.0f;
             printf("%7.4f\n", fConverted);
           }
        }
     }
     // Close the communication handles
     DapHandleClose(hdapSysGet);
     DapHandleClose(hdapSysPut);
     DapHandleClose(hdapBinGet);
   }
```

Following is the DAPL listing used by S100.CPP to configure the Data Acquisition Processor. S100.DAP configures the Data Acquisition Processor to read 100 values from single-ended analog input pin S0 at 10,000 us per sample, or 100 samples per second, and to send those data back to the PC through communication pipe $BinOut. S100.CPP reads these data using a DapBufferGet service. To change the number of readings, change both the COUNT statement in S100.DAP and the DataCount constant in S100.CPP.

```
S100.DAP:
   // DAPL configuration file for the S100 application
   reset

   idefine  my_sampling
     channels  1
     set ipipe0 s0
     time 10000
     count 100
     end

   pdefine  my_transfers
     copy(ipipe0, $BinOut)
     end

   start
```

Advanced Data I/O

The preceding example uses the simple data input service `DapBufferGet` to read data from the Data Acquisition Processor. The example assumes that whatever data are requested are available at the time of the get operation. While this can be guaranteed in this simple example, it usually is not possible to guarantee data availability in the same way for more advanced applications.

`DapBufferGet`, and every other DAPIO32 simple input/output service, has a built-in 20-second time-out. If the service is unable to process data for more than 20 seconds the service aborts the operation. For get services this means that the Data Acquisition Processor did not send data for more than 20 seconds, for put services this means that the Data Acquisition Processor did not accept data for more than 20 seconds.

Using the advanced input/output services `DapBufferGetEx` and `DapBufferPutEx`, an application developer can configure what data to get or put and can configure the time-out behavior of the services.

If an application can guarantee data availability from the Data Acquisition Processor, use the simple input services. If an application cannot guarantee data availability from the Data Acquisition Processor, use the advanced input service `DapBufferGetEx`.

If an application can guarantee output space to the Data Acquisition Processor, use the simple output services. If an application cannot guarantee output space to the Data Acquisition Processor, use the advanced output service `DapBufferPutEx.`

Linux Support

All DAP communications are handled by DAPIO32 functions provided by the DAPIO32 libraries. Users must never directly call the standard system file I/O functions, such as `open()`, `close()`, `read()`, `write()`, and `ioctl()`.

Following is a brief summary of the functions that are supported in Linux and their behavior, if different in Linux from Win32.

Functions Supported

`DapBufferGet`
`DapBufferGetEx`
`DapBufferPut`
`DapBufferPutEx`
`DapCharGet`
`DapCharPut`
`DapCommandDownload`
`DapComPipeCreate`
`DapComPipeDelete`
`DapConfig`
`DapConfigParamsClear`
`DapConfigParamSet`
`DapConfigRedirect`
`DapHandleClose`
`DapHandleOpen`
`DapHandleQuery`
`DapHandleQueryInt32`
`DapHandleQueryInt64`
`DapInputAvail`
`DapInputFlush`
`DapInputFlushEx`
`DapInt16Get`
`DapInt16Put`
`DapInt32Get`
`DapInt32Put`
`DapLastErrorTextGet`
`DapLineGet`
`DapLinePut`
`DapModuleLoad`
`DapModuleUnload`
`DapOutputEmpty`
`DapOutputSpace`
`DapReset`
`DapStringFormat`
`DapStringGet`
`DapStringPut`
`DapStructPrepare`

Functions Not Supported

`DapBufferPeek`
`DapModuleInstall`
`DapModuleUninstall`
`DapPipeDiskFeed`
`DapPipeDiskLog`
`DapReinitialize`
`DapServerControl`

Functions Modified

The following functions have new, restricted, or different behavior in Linux.

`DapComPipeCreate`:

- Accepts a Unix-style pipe name in addition to a UNC (Universal Naming Convention) name. See the discussion about Accel32 device names.
- The Linux Server currently restricts the maximum pipe size to smaller than 128K bytes.

- Pipes created in Linux are not persistent. They need recreating after a system reboot or after Accel32 for Linux is restarted.

`DapHandleOpen`:
- Accepts a Unix-style Accel32 device name in addition to a UNC name. See the discussion about Accel32 device names.
- Does not support the `DAPOPEN_DISKIO` open attribute.

`DapHandleQuery`:
- The following queries keys are not supported in Linux:
  “ServerEnumerate”
  “DiskFeedEnable”
  “DiskLogEnable”
  “ModuleInstallEnabled”
  “Transports”
  “BindTransport”
  “DapDiskIoCount”
  “DapDiskIoStatus”

Unix style Accel32 device names

The DAPIO32 interface in Linux accepts a Unix-style device name to refer to a target device the Accel32 Server manages. The interface also accepts UNC-style names, as used in Windows.

Three types of device targets are valid in Accel32: the server device (the Server itself), a DAP device, or a pipe device. For example, a handle can be obtained by using `DapHandleOpen` to obtain access to the Server, to a DAP that the Server manages, or to a specific pipe on a DAP.

To refer to the server, use the name "`/`".

To refer to a DAP, use the name "`/<dapname>`", such as "`/Dap0`".

To refer to a pipe on a DAP, use the name "`/<dapname>/<pipename>`", such as "`/Dap0/$SysIn`".

For example, the following statement opens `$SysOut` on `Dap1` for reading.

```
hSysOut = DapHandleOpen("/Dap1/$SysOut", DAPOPEN_READ);
```

And the next statement creates the a binary output communication pipe on `Dap0`, called `Cp2Out`, with default attributes.

```
if (!DapComPipeCreate("/Dap0/Cp2Out"))
{
... handle error ...
}
```

3. C++ Application Programming

This chapter discusses design strategies for applications that access data from Data Acquisition Processor boards, including both the DAP and xDAP families. Though the coding examples are in C++, the same strategies will apply to any host application platform or programming language.

The "DAP Development Software" that comes on any edition of the DAPtools Software CD, or an equivalent downloaded image from the Web, enables C or C++ applications to communicate with and operate a Data Acquisition Processor using the DAPIO API. In practice, just about everything in a Windows system uses dynamic link library (DLL) files in one way or another, and can access DAPIO32.DLL. Various higher-level programming languages require special configurations or *wrapper functions* to cross through from their higher-level notations to the lower-level DLL conventions. C++, being a relatively lower-level language, is better than most for accessing DLL functions directly.

When you use a DAP board, it is like one computer communicating with another one. You don't have direct access to hardware of that other computer — so you must communicate through the API functions. In many ways, this is a big advantage, because there are things that low-level device drivers cannot do, such as exist as shared network resources, coordinate multiple boards, stream data directly to disk drives, etc.

Software Components

The application running on your PC host will interact with other software components also residing in your host system.

- The DAPcell Services
  This software implements the DAPIO programming interface that your application uses. This is a running process that performs services to move data, command lines, and messages between your application and the DAPL operating system.
- DAPIO32.DLL
  This module defines the DAPIO programming interface functions that your application uses to access DAP boards. There is a 32-bit and a 64-bit version. You can operate one DAP board or multiple boards through the same interface.
- DAPIO32.H
  This header file is included by each source code module that uses DAPIO functions.
- DAPIO32.LIB
  This module is linked into compiled applications to give them run-time access to the DAPIO32.DLL functions. Actually, there are two versions of this file, one supporting 32-bit and the other supporting 64-bit.

There are other software elements that run on the DAP boards. DAPIO functions expect these to be present and running.

- The DAPL System
  Depending on your equipment type, the DAPL 2000 system or the DAPL 3000 system runs the data acquisition hardware independently. You will send a configuration script to the DAPL system to configure the manner that it captures data and generates output signals. You also specify the processing that you need. The processing might be as simple as transferring captured data through a communication channel to the host, but it can also include sophisticated data selection and DSP filtering operations, sensor calibration corrections, and so forth. The script can be sent one command line at a time, or copied directly from a file.

- Processing Tasks
  Executable processing task modules, downloaded to your DAPL system each time your host system boots up, provide the processing functions that run on a DAP to modify and transfer data. You will specify at least one processing task, to do something with the data you capture or generate.

Some exceptions:

iDSC boards have their own programming interface. It is similar but supports a different set of features. This is covered in the iDSC Reference Manual.

The Linux system uses a separate Accel32 kernel driver that supports most of the DAPIO features, but not those implemented by a server process.

Transfers of data go through logical data channels, called communication pipes. There is more information about communication pipes in the Overview Chapter, and in particular you need to know about the four communication pipes set up by default.

- Use `$SysIn` to send commands to the DAPL system.
- Use `$SysOut` to receive messages and formatted text from the DAPL system.
- Use `$BinOut` to transfer binary data to the host from a processing task.
- Use `$BinIn` to send binary data from the host to a processing task.

Installation

Install your Microsoft compiler software.

It seems like this should be straightforward, but for some reason getting the compiler set up and working right sometimes seems like the hardest part. Make sure that your compiler is completely operational before continuing.

Install your DAPtools Development software.

If you have downloaded the CD image, first run the extractor program, which will place all required data and the `setup` program on your computer disk drive.

If you have not installed the DAP board, do so according to the DAP Install Guide.

As part of a normal DAP board installation, you will run the setup.exe program to install the DAPcell software. This will provide all of the basic software you need to run applications, and it will register a copy of the `DAPIO.DLL` in your Windows system.

To prepare for application development, run the installer program again, but this time select the DAP Development software. This will place the programming interface files you need to compile your applications, including the library and header files, on your disk drive.

After the installation is completed, you should be able to find the DAP development files in the installation folder that you specified. Assuming that it is the default installation location on your local c: drive, you should find the following.

- Example source files in subdirectory `DapDev\Examples\C`
  It is a useful exercise to build everything as provided, to make sure that this is working properly, before you try building your own application code.

- `DapDev\Import\C\DAPIO32.H` header file
  You will need to include this in each source code file that uses DAPIO functions. You can examine the header file DAPIO32.H to see the declarations of data structures and functions.
- `DapDev\Import\LIB\MC\x86\DAPIO32.LIB` and `DapDev\Import\LIB\MC\x64\DAPIO32.LIB` library files
  Depending on whether you are compiling a 32-bit or 64-bit application, the appropriate file must be linked into your application so that it knows what functions are available in the DAPIO32.DLL file.

Configure your compiler environment

Configure your compiler environment so that it knows

- how to locate the `DAPIO32.H` file when it is compiling your application code. This file is typically referenced by a compiler command line "*I option*" so that the compiler can find it.
- how to locate the `DAPIO32.LIB` file when it is linking your binary application code together to build the executable file. This file is typically listed on the compiler command line with other parameters that are passed to the linker utility.

If you use the examples provided with the DAP Development software, and install the software in the normal locations, the makefile for the examples show the required references to these files.

Determine your onboard processing configuration

Your DAP board will only provide the data that you configure it to send. The configuration of the DAP board is not an extensive process, but it is a precise one. If any detail is wrong, nothing seems to happen – giving few clues. You can validate your DAPL configuration script, verify that your DAP equipment operates correctly, and verify that your signals are valid, using a utility such as DAPstudio. If your configuration commands work correctly when operated independently, they will work exactly the same way under the control of your application.

Configure for your application build

Once you have your compiler generally working, with your DAP software installed and working, set up your project to begin development for your new software.

The DAP Connection Life Cycle

After you have established a basic framework for your application – so that the application compiles, executes, and then terminates successfully – you will next want to include the four stages of the "DAP connection life cycle," as outlined briefly in the Overview Chapter.

If you think of a DAP device as a resource "object," the management life cycle for this object should be something like the following.

- Use the `DapHandleOpen` function to connect to each DAP board and each communication channel you need to use, establishing the direction as *`input`* (to the DAP) or *`output`* (from the DAP), and receive the `handles` that allow further access.
- Use the *`handle variables`* to perform various configuration activities to prepare for running.
- Use the *`handle variables`* to start DAP processing and to perform data transfer operations.
- Use the `DapHandleClose` function to release all of the DAP handles obtained during the initialization.

These phases mesh well with the typical stages of a Windows application.

Initialization

Early in the program initialization sequence, allocate “handle” variables for selecting DAP boards and accessing the communication pipe connections. If you are using several communication pipes, you might want to define a pipe management object to organize them. Most applications, however, use the `$BinOut` file channel almost exclusively after the initialization phase, so extra packaging might be unwarranted.

```
// Reserving handles for board and communication access
  HDAP  hDAP0
  HDAP  hCmdSend;
  HDAP  hDataRecv;
  HDAP  hDataSend;
  HDAP  hMsgRecv;
  …
```

Near the end of the initialization phase, before you are ready to start the main application event polling loop, use the handle variables to establish connections to the data transfer channels. Use the UNC naming conventions as discussed in the Overview Chapter to identify which DAP board and which communication channel you are processing.

The processing to open the DAP communication channels will look something like the following.

```
// Reserve a DAP board for the application
  hDAP0 =  DapHandleOpen("\\\\.\\dap0",DAPOPEN_WRITE)
  if  (hDAP0==0)
    error("Cannot reserve DAP0 for this application");
  DapReset(hDAP0);

// Open DAP communication channels
  hCmdSend  =   DapHandleOpen("\\\\.\\dap0\\$SysIn",DAPOPEN_WRITE)
  if  (hCmdSend==0)
    error("Error opening DAP text input handle");
  hDataRecv = DapHandleOpen("\\\\.\\dap0\\$BinOut",DAPOPEN_READ)
  if  (hDataRecv==0)
    error("Error opening DAP binary data output handle");
```

In communication pipe names, “input” and “output” are from the point of view of the DAP board. Thus, you will open a `$BinIn` pipe to write to the DAP board and a `$BinOut` pipe to read from the DAP board. You can select handle names that are more intuitive for coding your application. The input and output option codes are defined in the `DAPIO32.h` file.

You can use UNC identifier strings like the following to open channels to a second DAP board, residing on a different host machine, to operate the boards in parallel.

```
”\\\\PC101\\Dap1\\$SysIn”
”\\\\PC101\\Dap1\\$SysOut”
”\\\\PC101\\Dap1\\$BinIn”
”\\\\PC101\\Dap1\\$BinOut”
```

Notice that when using the UNC path names, each backslash character is represented in the text strings as a backslash pair. This is a syntactic feature of C++, which use a single backslash as an "escape character" notation to prefix special character codes.

Most applications will find the predefined communication pipes sufficient, but there are some cases where additional communication pipes simplify things a lot.

- A special channel can route "messages" directly to one specific processing task.
- Separate channels can serve to keep data with dissimilar data groupings and rates separated, avoiding data backlogs and making data management easier.
- Information blocks can be passed in one pipe, to indicate how to interpret the bulk data from another pipe.

To use the additional pipes, you must first configure them, either using the *Browser* tab in the DAPcell *"Data Acquisition"* control panel application or by calling a DAPIO `DapComPipeCreate` function from your application. Use one of the reserved names for the new communication pipe.

```
Cp2In  …  Cp31In          Cp2Out  …  Cp31Out
```

Certain advanced applications will send special status queries to a DAP board, not specific to any one of the data channels. For these queries, use a special kind of DapOpen call in which the UNC path does not list a particular communications pipe.

```
    hQuerySend  =   DapHandleOpen("\\\\.\\dap0",DAPOPEN_QUERY)
```

Clearing prior operations

Unknown previous operations could have left various old data, or possibly a running configuration. Whatever this prior activity might have been, you don't want it to clutter your current application. So one of the first things done in the initialization sequence is an operation to clear the DAP. This stops any ongoing DAP processing, removes past configurations, and clears away any unwanted old data. This combination of things occurs so often that a special shorthand function is provided. Use it for each DAP board.

```
DapReset( hDAP0 );
```

Now that the DAP is clear and can no longer spawn new undesired data, flush any data that accumulated in buffer memory on the host side.

```
DapInputFlush( hDataRecv );
```

Repeat for each communication channel that the application needs to use.

Downloading a DAP configuration

You must configure your DAP board, or it will not know what data to generate or transfer. It is typically a good idea to configure the DAP immediately after opening the communication channels, to avoid "DAP startup" time lags later, when other activity begins.

You can think of configuration commands this way: they tell the DAP what it *will do*. It doesn't actually do any of this until the START command. (This is a little oversimplified, but most commands show no observable effects in terms of data input or output until you run the configuration.) Applications that need to get the DAP working as soon as possible can include a START command at the end of the configuration.

Full details of configuration commands are provided in the DAPL 2000 Manual or the DAPL 3000 Manual. Most applications will specify an input procedure section, defining how to capture data samples, and a processing procedure section, defining what to do with them. Some applications will specify an output procedure section, defining signal generation.

There are two strategies for downloading the configuration. The first is to send the information one command line at a time. For example, the following sends an input processing configuration this way.

```
DapLinePut(hCmdSend, "IDEFINE   MySampling");
DapLinePut(hCmdSend, "CHANNELS   2");
DapLinePut(hCmdSend, "SET  IPIPE0   D0");
DapLinePut(hCmdSend, "SET  IPIPE1   D1");
DapLinePut(hCmdSend, "TIME   25.0");
DapLinePut(hCmdSend, "END");
```

This strategy is effective when significant changes to the configuration are required in response to run-time information entered by the application user. But most of the time, the configuration never needs to change, and it is much easier to collect configuration lines in a file and download them all in one operation.

```
DapConfig(hCmdSend, "C:\DAPapps\MyApp\config.dap");
```

As an example of using this method, suppose the MySampling configuration was set up previously. The file config.dap specifies the following DAPL configuration commands. These define processing that will transfer two channels of input sample data (as defined by the MySampling input configuration) to the host application.

```
// Configuration lines to make the DAP upload data
PDEFINE   MyTransfers
COPY( IPIPES(0,1), $BinOut )
END

START
```

It is not a one-way-or-the-other choice. You can send some things one line at a time, and other things from a pre-configured file. What matters is the sequence of lines as they are received by the DAPL system, from the RESET operation to the START operation.

For applications that need to start the DAP activity with minimum delay, it is useful to put the START command at the end of the configuration file, as shown in the previous example. But other applications might want to defer starting the data collection until a user clicks a graphical "Start button" in an application screen. For this case: remove the START command from the file, and instead use the following command in response to the user's click.

```
// User has clicked… start the data acquisition.
DapLinePut(hCmdSend, "START");
```

The possibilities for what you can configure using the DAPL commands are almost unlimited. There are many examples of DAPL command scripts on the Microstar Laboratories Web site, as well as the manuals for your products and the DAPL reference manuals.

Efficient Run-Time Processing

When you start your DAPL configuration, it will generate data predictably, in the manner that you defined. The data will start streaming through the communication channels. In contrast, application timing will be anything but certain. Your application runs at those moments the operating system scheduler deems appropriate. The delays are unpredictable, depending on various other system activity (graphic displays, desktop applications, disk activity, mouse pointer management, etc.).

A "stream of data in a channel" is almost like a "stream of water in a pipe." If you pour a liter of water into a pipe, and then pour another liter of water into the pipe, what will you get out of the other end? Depending on the timing, you might get the first liter, you might get both liters… or you might get the first one and part of the second. The data in a communication pipe are somewhat better behaved than water, because the data will always emerge in exactly the same order as they went in. However, you are not guaranteed to receive the data in any particular amount at any predictable time. Your task is to keep pace, and not lose anything.

There are various strategies that you can use for your data transfers, depending on the goals of your application.

One Value At A Time

The idea is that at each opportunity the scheduling loop will call a primitive function that extracts one unit of data from the communications pipe buffers. Do this often enough and you can capture any amount of data.

```
// Call this at every opportunity to obtain the next data.
// This idea is probably doomed!
void   GetValueFromDap( HDAP * hDataRecv )
{
    short int *value;
    while  (  ! DapInt16Get( hDataRecv, &value ) )  /* WAIT... */ ;
    return;
}
```

There are also corresponding functions `DapInt32Get` and `DapStringGet` for receiving higher precision integers or text messages in a similar manner.

While this idea seems very easy, it has some severe problems. The part about "do this often enough" is not under your control. Each delivery of one value must pass through operating system software layers for device control, system data transport, I/O buffering, and the application interface. You will be doing very well if you complete 100 input transactions per second this way. Even worse… what happens if arrival is delayed for some reason? Your application will *hang* in this function.

There are times when this strategy is helpful. For example, if you have a communication pipe that never contains anything except emergency warning flags, you expect the pipe to be empty. This case is handled efficiently, and you continue immediately with other normal processing. If something does appear, handle the emergency condition directly. *For general-purpose data transfers, however, this idea is usually a disaster.*

One Block At A Time

You can't make your operating system more efficient, but if you can move many values (for example, 1000) with each operation, then the overhead per transported value reduces dramatically, and you can move a lot of data.

```
#define   BUFFER_LENGTH   1000

// Call this to receive a new block of input data.
int   GetBlockFromDap( HDAP * hDataRecv, short int * buffer )
{
    int  Nreceived =
         DapBufferGet( hDataRecv, BUFFER_LENGTH, (void *)buffer );
    return  Nreceived;
}
```

This looks like a great solution. You gain great efficiency from a larger block transfer. If the buffer cannot be completely filled, or if some kind of error condition arises, you can determine this from the function return value, so that it is clear what to do with the data that arrive in your storage area.

The problem is timing. Blocks of data take some time to collect and organize. The `DapBufferGet` function should attempt to return a block of the size you request, if it can, but it has no way to know how much time to allow for this. It can wait up to 20 seconds to fill the request, and even this might not be enough. During this time, your application is effectively blocked from executing, and will be unresponsive.

This strategy can be successful if you know that your data will arrive in one big block, with no excessive delay, and with no other processing needed until the data block arrives. Otherwise, you are probably better off using a different strategy.

Take Everything Available – the TDapBufferGetEx structure

Sometimes you don't really need to worry about how the data arrive, you just need to move everything with great efficiency. The "take everything" strategy is very effective for this. A classic example is the problem of streaming very large amounts of data to disk storage efficiently for data logging.

At each opportunity, attempt to receive a full data block. *If you do:* deliver the data block for other processing and immediately attempt to receive another block. *If you do not:* this is probably the last of the available data. Deliver whatever data you do receive to other processing, and then return to the caller. Repeat these as many times as possible, but if things get really busy, take a break now and then to let the rest of the application have a chance to run.

To control your data transfers, several configuration parameters are needed. These parameters can be configured once and then used repeatedly by defining a structure of type `TDapBufferGetEx`, as declared in the DAPIO32.H header file. Here is an example. There are 16 data channels to sample, so data transfers are deferred until a set is available covering each channel. Transfers are done only in complete 16-channel groups.

```
// Prepare buffer control structure
int const   NCHAN = 16;
int const   NBUFFER = NCHAN*128;
  short int   iStorage[NBUFFER];
TDapBufferGetEx   BufControl;
DapStructPrepare( BufControl, sizeof(TDapBufferGetEx) );
```

```
// At least 16 samples
BufControl.iBytesGetMin = NCHAN*sizeof(short int);
   // As many as 128 of these groups
BufControl.iBytesGetMinax = NBUFFER*sizeof(short int);
// Always groups of 16 samples
BufControl.iBytesMultiple =  NCHAN*sizeof(short int);
```

Since this strategy is for processing lots of data, the rest of the application must attempt to cover all other activity very quickly and concentrate on being ready for data arrival. So assume that if there is any waiting to do, it will be done in the functions that receive the data stream. Allow reasonable time intervals for data arrival, but if data do not arrive quickly, take the opportunity to let the application do other things.

```
BufControl.dwTimeWait  = 5;        // Milliseconds to wait for first new data
BufControl.dwTimeOut    = 20;      // Milliseconds maximum before returning
```

If data blocks happen to be buffered and ready, this will grab them and return the data quickly. Up to 20 milliseconds are allowed to obtain a complete buffer load. If some data arrive, but not enough to completely fill the buffer within 20 milliseconds, the data transfers are probably "caught up" so it is OK to leave and let other application activity run for a while. But in the worst case, 5 buffer loads could require nearly 100 milliseconds. After this much time, the main scheduling loop should be given a chance to take care of other things such as file management, display, use controls, and so forth.

After each call to the following function, the main polling loop learns about how much new data has arrived and can take appropriate action.

```
// Call this to fetch and deliver any new data that arrive.
// Report the number of new values.
int   ReceiveAvailableData( HDAP * hDataRecv, short int * buffer )
{
    int  total = 0;
    int  received;
    int  bytes;

    // Exit this loop if 5 buffer loads have been collected (up to 100mS)
    for ( int iRepeat=0; iRepeat<5; ++iRepeat )
    {
         bytes = DapBufferGetEx(hDataRecv, &BufControl, (void *)buffer);
         // Deliver any data received
         if (bytes > 0)
         {
              received = (bytes / sizeof(short int));
              DeliverData(buffer,received);
              total += received;
         }
        // Exit this loop if the last buffer was not completely filled
        if ( bytes < BufControl->iBytesGetMax )
              break;
    }
    return total;
}
```

Your application will determine what the application-specific `DeliverData` function does with the data.

This example used a loop to guarantee that regardless of how busy the data collection, the main application loop will occasionally get control for other activity. Another strategy that advanced programmers can consider is using a separate thread to control the time allocated to data transfers and the main application loop.

Don't Wait, Take What You Need

This strategy can be useful when receiving small "tagged" data blocks, of the sort that occur when there are multiple activities on the DAP board producing small amounts of data. The "tag" is just a small data block preceding each group of actual data, describing the type and the amount of actual data to follow. After you have the tag and know the block size, you want to fetch the data for one block exactly, to avoid additional tags or fragments carrying over from one activity to the next. If a full block is not available, try again later.

You can do this by *polling* the available contents of the transfer channel, and modifying the control configuration so that only data for a complete block are delivered, without waiting for any new data to arrive. Set up a buffer control structure, but leave two critical fields empty.

```
// Buffer control structure prototype for adjustable-size data blocks
   short int   iBlock[MAXBLOCK];
TDapBufferGetEx   BufControl;
DapStructPrepare( BufControl, sizeof(TDapBufferGetEx) );
BufControl.iBytesMultiple =  NCHAN*sizeof(short int);
```

Unlike the *Take Everything strategy*, data arrival could be slow and very irregular. Waiting for data is seldom going to be productive, so wait for data as little as possible, and leave most of the timing control to the main application loop. The input processing is less efficient, but the data management is as simple as possible.

```
BufControl.dwTimeWait  = 2;       // Milliseconds to wait for first new data
BufControl.dwTimeOut   =  20;     // Milliseconds total before returning
```

Now your application code will first call a function to check whether the next tag is available. If not, return without waiting for anything.

```
// Call this to fetch available new tag data - otherwise, try again later
BOOL    FetchTag( HDAP * hDataRecv, short int * type, short int * length )
{
    BOOL  result;
    // Verify that tag data are ready before continuing
    result = (DapInputAvail(hDataRecv) >= 2*sizeof(short int));
    if  ( result )
    {
       // This is very easy, but a blocked transfer is actually more efficient
       DapInt16Get(hDataRecv,&type);
       DapInt16Get(hDataRecv,&length);
    }
    return result;
}
```

When the tag arrives, the main polling loop learns the amount of data in the following data block. By setting the minimum and the maximum size equal to this block size, it is guaranteed to receive exactly this data. Check whether the data for this block have arrived. If so, load the data block. If not, wait until later.

```
// Call this to load the predetermined block size into the buffer area.
// If a complete block is not available -- try again later.
BOOL    FetchAdjustableBlock(
      HDAP * hDataRecv,  short int length, short int * buffer )
{
    BOOL  result;
    // DANGER - are you sure the block fits completely in buffer memory?
    result = ( DapInputAvail(hDataRecv) >= length*sizeof(short int) );
    if   (result)
    {
        // Block data ready and buffered. Now fetch it.
        // Make local copy of control 'template' and supply missing fields
        TDapBufferGetEx   LocalBufControl = BufTemplate;
        LocalBufControl.iBytesGetMin = length*sizeof(short int);
        LocalBufControl.iBytesGetMax = length*sizeof(short int);
        DapBufferGetEx(hDataRecv, &LocalBufControl, (void *)buffer);
    }
    return result;
}
```

Beware that long data blocks might not fit in the buffer memory available for the application. If this happens, the `DapInputAvail` function will never succeed, and your application will never see any of the data. For transfers of a few hundred samples, this should be safe. For a few thousand samples, probably not. For efficiently streaming large amounts of data, you need long blocks and a different strategy.

Take Block If Available

This strategy can be very effective when you know that your data source generates fixed size blocks with periods of delay between. If any new data are available, you know it means a complete new block is coming, and you want to process it; otherwise you do not want to wait. Data are taken in units of complete blocks. The rate must be low enough that your application has no difficulty keeping pace.

Configure a control parameter block of type `TDapBufferGetEx` with minimum and maximum transfer sizes set equal to each other and equal to the length of one block. (The `iBytesMultiple` field will not matter as long as it is something reasonable.) For this example, suppose that each data block is a 2048-value block produced by a an FFT, with the results in a 32-bit floating point data type.

```
int const   NBUFFER = 2048;
  float    fStorage[NBUFFER];
TDapBufferGetEx   BufControl;
DapStructPrepare( BufControl, sizeof(TDapBufferGetEx) );
BufControl.iBytesGetMin = NBUFFER*sizeof(float);
BufControl.iBytesGetMinax = NBUFFER*sizeof(float);
BufControl.iBytesMultiple = sizeof(float);
```

If any data have arrived, we want to allow plenty of time to load everything. But if there are no data, we don't want to wait. The pace is not fast, and if we see no immediate data it is safe to try again later.

```
BufControl.dwTimeWait  = 2;        // Milliseconds to wait for first new data
BufControl.dwTimeOut   =  100;    // Milliseconds maximum before returning
```

Data are processed one block at a time, so there is no need to call a `DeliverData` function; the data can be safely processed directly from the buffer storage.

```
// Call this to fetch one new data block.
BOOL   ReceiveFFTBlock( HDAP * hDataRecv, float * buffer )
{
    bytes = DapBufferGetEx(hDataRecv, &BufControl, (void *)buffer);
    if (bytes > 0)
         return true;
    else
         return false;
}
```

Take All Blocks Available

This strategy is really a combination of the buffer control configuration described in the preceding *"Take One Block"* section with the algorithm described in the *"Take Everything Available"* section before that. It is a very safe and efficient strategy when data blocks are produced faster than the Windows polling loop can ask for them.

Take Most Recent Block

This strategy is effective for applications that need to record data on a time schedule defined by an external process. Most applications are limited to one of two ways to do this.

1. Stream the data into the host so that if an external request arrives, the current information necessary to satisfy the request is available. This is rather wasteful of resources because most of the data and management effort will end up being discarded.
2. Wait until the external request arrives. Send a request for data acquisition, wait for data to be collected, wait for data to be transferred, process the data, and deliver the response. By the time this is done, the readings are no longer current.

With a DAP board to help, there is a third option: a combination of these two strategies, with the host and DAP processing sharing the load.

3. Let the DAP continuously collect data at a high rate, keeping the most current readings in memory. When an external request arrives, the host requests the "most recent block" of data. By the time the application can post a request to receive the response data, the DAP typically has completed the transfer, making the results seem instantaneous.

Some special processing is required on the DAP board so that it sends data only on request, rather than streaming it continuously. This special processing is provided by the `MRBLOCK` (most recent block) processing command, which must be included in the DAPL configuration sent to the DAPL system. (`MRBLOCK` is not provided with all versions of the DAPL system, but if your version does not have it, a downloadable module containing the `MRBLOCK` command is available on request, compatible with all current DAP and xDAP models.) You also will need to open a connection to the `$BinIn` communication pipe in your initialization section.

The DAPL processing will look something like the following.

```
// Processing to generate “most recent block” reports
PDEFINE   MRBsend
  // Various processing here will generate the data
  …
  // Collect blocks of results
  NMERGE( 1, item1, 1, item2, 16, item3, … , report)
  // Send only the most recent block, on request received from $BinIn
  MRBLOCK( report, $BinIn, BLOCKSIZE, $BinOut )
END
```

Now your application can periodically request data in the following manner.

```
// Call this each time the external system requests a new status report.
//
void   GetMostRecentStatus( HDAP * hRequestSend, HDAP * hDataRecv,
     void * buffer )
{
    // Send the request for 1 block to the DAP processing
    DapInt16Put(hRequestSend,1)
    // Expect an immediate response that delivers one full block
    DapBufferGet( hDataRecv, BUFFER_LENGTH, buffer );
    return;
}
```

Data Access Considerations

You must construct your application so that it polls the data transfers frequently enough to keep pace with all of the data that the DAPL system is configured to send. Furthermore, other processing must remove the data from your buffers to free the storage. This can lead to interesting challenges for demanding, high-rate applications:

- How can you organize memory buffers efficiently, so that the DAP data can be received and processed without too much conflict?
- How can you organize your processing threads so that other processing is not stalled while awaiting data arrival from the DAP?

These are ordinary data management issues that you would face with any Windows application. The only difference is that the DAP can produce data so quickly that the data streams overwhelm your application if processing is not timely and efficient.

When you configure your data storage, it will typically receive data in a *multiplexed sequence* – one value from channel 0, one value from channel 1, one value from channel 2, ... and on to the value from channel N-1. Then this starts over with channel 0. These channel-by-channel values are what you will find stored in your buffer memory in consecutive locations. You can prepare for this if you wish, as in the following example with 10 channels of sampled data and a buffer that can retain up to 400 of these groups.

```
//DAPL configuration:
    COPY(IP(0..9),$BinOut)


//Host software
short int   iMyStorage[4000];
short int   (* pMyArray) [400][10];
```

```
...
    pMyArray = (int (* )[400][10])(&iMyStorage[0]);
```

Now, when the data buffer is returned, the data are addressable in the `iMyStorage` area either as a raw sequence of 4000 samples, or as `pMyArray` with 400 groups of 10 sample values, depending on which pointer you use.

```
for  (int ichannel=0; ichannel<9; ++ichannel)
    {
        value = pMyArray[igroup][ichannel];
        ....
    }
```

On the other hand, suppose that you apply processing in the DAPL system that organizes the data in a different manner, sending data in contiguous blocks, not mixed channel-by-channel. Here is an example of DAPL processing that can produce this kind of data organization.

```
//DAPL processing configuration: produces 5 blocks of 512 terms
PDEFINE   spectra
    FFT(5,10,0,ip0,PFFT0)
    FFT(5,10,0,ip1,PFFT1)
    FFT(5,10,0,ip2,PFFT2)
    FFT(5,10,0,ip3,PFFT3)
    FFT(5,10,0,ip4,PFFT4)
    BMERGE(PFFT0,PFFT1,PFFT2,PFFT3,PFFT4,512,$BinOut)
END
```

This kind of data organization can also be supported in your application code.

```
//Host software
short int   iMyStorage[5120];
short int   (* pMyArray) [10][512];
...
    pMyArray = (int (* )[10][512])(&iMyStorage[0]);

    for  (int iterm=0; iterm<512; ++iterm)
    {  value = pMyArray[itransform][iterm];
        ....
    }
```

Application Termination and Cleanup

It is likely that when you are ready to stop your application, the DAP board remains fully configured, and possibly still running. If you leave the board in this condition, the DAPL system on the board will not know whether it is safe to stop operation or whether your application is just experiencing a temporary delay. If the board continues running after your application is closed, this can leave the DAP board inaccessible by other applications – even trying to start the same application again might fail. Leaving the DAP running can also cause unexpected data to remain in the transfer pipes, creating a possibility that the stale data will be seen in the data channel later.

For these reasons, it is important to shut down the activity on the DAP board before closing the application. First, stop the processing so there is no conflict or delay as other things are shut down.

```
DapLinePut(hCmdSend, “STOP”);
```

Then issue commands to close the DAP communication channels, closing every communication channel that you opened.

```
  result = DapHandleClose(hCmdSend);
  if  (result==-1)
    error("Error closing DAP command handle");
  result = DapHandleClose(hMsgRecv);
  if  (result==-1)
    error("Error closing DAP message output handle");
  result = DapHandleClose(hBinRecv);
  if  (result==-1)
    error("Error closing DAP data input handle");
```

Before terminating your application, clear everything in the DAPL configuration. This removes all of the existing configurations and purges any remaining unused data from communication pipes.

```
DapReset(hDAP0);
DapHandleClose(hDAP0);
```

Strictly speaking, if your application is “well defended” in your startup sequence, it will not fall victim to the residual effects of prior processing. But there is no point in leaving possible hazards for other applications.

Summary of Basic DAPIO Functions

Here are the DAPIO functions illustrated in this section. Look them up in the DAPIO function reference section of this manual for complete details.

`DapHandleOpen`	Opens a handle to a DAP communication pipe, specified using Microsoft networking UNC (Universal Naming Convention) path, specifying **read** or **write** access, receiving the DAP handle in return.
`DapInputFlush`	Clear data from the specified communication channels, in case any data remains unprocessed from prior operations.
`DapConfig`	Send a text file of configuration commands to the specified DAP.
`DapReset`	Stop all DAP activity, clear away any unsent data left over in memory, and clear away any existing configurations.
`DapHandleClose`	Release a connection to the specified DAP communication channel.
`DapStructPrepare`	Initialize fields of a DAP control block prior to other use.
`DapInputAvail`	Test whether data are available in the communication channel without suspending the thread to wait for data arrival.
`DapBufferGetEx`	Transfer a block of sample data from the DAPL system into application buffer memory.
`DapLinePut`	Transfer a line of command text to the DAPL system
`DapInt16Get`	Receive the first short int (16-bit) value from transfer buffer memory.

4. DAPIO32 Interface Reference

The following pages provide reference information for the structures and functions included in the DAPIO32 Interface.

DAPIO32 Structure Reference

Structure Reference

TDapBufferGetEx
TDapBufferPeek
TDapBufferPutEx
TDapCommandDownload
TDapHandleQuery
TDapIoInt64
TDapPipeDiskFeed
TDapPipeDiskLog

Structure Usage

Applications use structures to pass information to and from several of the DAPIO32 functions. To ensure correct operation, application code must always fully initialize a structure before passing it to a DAPIO32 function.

Structure Initialization

The DAPIO32 interface provides a function and a template to facilitate structure initialization. Application code can use either the function or the template `DapStructPrepare` to do the initial setup of the structure, and then set the fields relevant to the particular operation to required values.

For example:

```
    . . .
    {
        TDapCommandDownload dcd;
        DapStructPrepare(dcd);

        dcd.hdapSysPut                  = hDapSysPut;
        dcd.hdapSysGet                  = hDapSysGet;
        dcd.pszCCFileName               = pszFile;
        dcd.iCCStackSize                = 1000;
    . . .
    }
```

The preceding code fragment initializes the `dcd` structure to zero and sets the *`iInfoSize`* field, using `DapStructPrepare` template. It then initializes all other non-zero required fields for the desired operation. This sequence of steps should be observed as a normal practice for any DAPIO32 structure initialization.

Binary Compatibility

Applications compiled with an older DAPIO32 interface will typically run with a newer DAPIO32.DLL. The `README.TXT` file included with each release of DAPIO32.DLL lists the versions of the DAPIO32 interface supported by that DLL. Code modification and recompilation are only required if the version of the interface that was previously used by an application is no longer supported, or if an application needs to take advantage of additional features offered by the newer interface.

Alphabetical Structure Reference

Following is a complete alphabetical listing of all DAPIO32 structures.

See Also

DAPIO32 Function Reference

TDapBufferGetEx

The `TDapBufferGetEx` structure defines the behavior of the `DapBufferGetEx` function.

```
typedef struct tag_TDapBufferGetEx {
    int iInfoSize;                          // Size of this structure
    int iBytesGetMin;                       // Minimum bytes to get
    int iBytesGetMax;                       // Maximum bytes to get
    int iReserved1;                         // Not used; must be zero
    unsigned long dwTimeWait;               // Time interval to wait for new data
    unsigned long dwTimeOut;                // Total time for entire operation
    int iBytesMultiple;                     // Bytes to get is a multiple of this
    } TDapBufferGetEx;
```

Members

iInfoSize
Specifies the size of this information structure.

iBytesGetMin
Specifies the minimum number of bytes to get. It can be zero or a positive integer that is a multiple of *iBytesMultiple*.

iBytesGetMax
Specifies the maximum number of bytes to get. It must be greater than or equal to *iBytesGetMin* and a multiple of *iBytesMultiple*.

dwTimeWait
Specifies the longest time in milliseconds to wait for new data to arrive. If no new data arrive in this amount of time, the service aborts the operation. A value of zero means “do not wait”.

dwTimeOut
Specifies the longest time in milliseconds to complete the entire operation. If the operation fails to complete in this amount of time, the service aborts the operation. When this member is non-zero, it takes precedence over *dwTimeWait*. A value of zero means do not time-out (wait indefinitely if necessary).

iBytesMultiple
Specifies that the number of bytes to get for *iBytesGetMin* and *iBytesGetMax* be restricted to a multiple of this value.

Remarks

An application must fully initialize `TDapBufferGetEx` before passing it to `DapBufferGetEx`. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

See Also

`DapBufferGetEx`, `DapStructPrepare`

TDapBufferPeek

The `TDapBufferPeek` structure defines the behavior of the `DapBufferPeek` function.

```
typedef struct tag_TDapBufferPeek {
    int iInfoSize;                    // Size of this structure
    int iBytesGetMin;                 // Minimum bytes to get
    int iBytesGetMax;                 // Maximum bytes to get
    unsigned long bmMode;             // Bit mode flag
    unsigned long dwTimeWait;         // Time interval to wait for new data
    unsigned long dwTimeOut;          // Total time for entire operation
    int iBytesMultiple;               // Bytes to get is a multiple of this
    void *pBuffer;                    // Pointer to user-supplied buffer
    TDapInt64 i64OffsetRequested;     // Requested offset of data
    TDapInt64 i64OffsetResult;        // Actual offset of data returned
    int iBytesResult;                 // Number of bytes returned
    } TDapBufferPeek;
```

Members

iInfoSize
Specifies the size of this information structure.

iBytesGetMin
Specifies the minimum number of bytes to get. It can be zero or a positive integer that is a multiple of *iBytesMultiple*.

iBytesGetMax
Specifies the maximum number of bytes to get. It must be greater than or equal to *iBytesGetMin* and a multiple of *iBytesMultiple*.

bmMode
Specifies the mode of the operation, *dbpk_Relative* or *dbpk_Absolute*.

The relative mode requires that the last unit of data (see description of *iBytesMultiple* for the definition of data unit) be the most recent available data unit in the target, or, if the value of *i64OffsetRequested* is non-zero, the data unit *i64OffsetRequested* back from the most recent available data unit.

The absolute mode (default) requires that data start at the offset specified in *i64OffsetRequested*.

dwTimeWait
Specifies the longest time in milliseconds to wait for new data to arrive. If no new data arrive in this amount of time, the service aborts the operation. A value of zero means "do not wait".

dwTimeOut
Specifies the longest time in milliseconds to complete the entire operation. If the operation fails to complete in this amount of time, the service aborts the operation. When this member is non-zero, it takes precedence over *dwTimeWait*. A value of zero means do not time-out (wait indefinitely if necessary).

iBytesMultiple
Specifies the size of the smallest transferable data unit as well as its alignment offset. A data unit always starts at an offset that is the multiple of this value. A value of zero means no user-specified unit size and alignment,

which is equivalent to the size and alignment of the underlining target data width. If non-zero, this value must be a multiple of the target data width.

For pipes, the data width is the width of the data type of the pipe. For disk files, the data width is always one.

pBuffer

Points to a user-specified buffer. The buffer must be at least *iBytesGetMax* in size.

i64OffsetRequested

This value must be a multiple of the data unit size, that is, a multiple of the value of *iBytesMultiple*.

In relative mode, this field must be zero or negative. If this is zero, the last unit of requested data is the most recent unit available. If this is a negative number, the requested data units shift back in time by this offset. For pipes, the maximum requested data length (*iBytesGetMax*) plus the possible shift length specified by this field must be no longer than the maximum pipe size. If the requested data is not available yet, waiting may occur. If a time-out has occurred before all data becomes available, whatever available is returned.

In absolute mode, this is the requested offset of the first data unit. If the requested data is available, the data returned is exactly the request. If some or all of the requested data no longer exist in the target, a block of the next available data unit is returned. If the request refers to data in the future, waiting may occur. If a time-out occurs, a block of data smaller than requested may be returned. In all cases, the offset of the first data unit returned is in *i64OffsetResult*. Following are examples for pipe history reading. Disk file reading is analogous to it except that there is no restriction on the request imposed by the pipe maximum size.

For example, in relative mode with data unit size (*iBytesMultiple*) of 2 and current last byte of data at offset 999 or 1000,

a 100 bytes of data (50 units) with *i64OffsetRequested* of 0 will be bytes from offset 900 to 999, and

a 100 bytes of data (50 units) with *i64OffsetRequested* of -100 will be bytes from offset 800 to 899.

A request for 100 bytes of data (50 units) with *i64OffsetRequested* of -1000 will return nothing if no wait time or too short of a wait time is specified. Otherwise, it returns one or more units up to *iBytesGetMax* from offset 0 or larger if at least one unit of data shows up during the waiting period.

In absolute mode with unit size (*iBytesMultiple*) of 2, *i64OffsetRequested* 1000 and the pipe maximum size 10000 bytes,

if the current last byte of data is at offset between 1099 and 10999, a 100 bytes (50 units) of data is returned from offset 1000 to 1099, and

if the current last byte of data is at offset 11099, a 100 bytes (50 units) of data is returned from offset 1100 to 1199.

If the current last byte of data is at offset 99 and a wait time is specified, a 100 bytes (50 units) of data may be returned from offset 1000 or larger. If a time-out occurs, less than 50 units may be returned. If no wait time is specified or the wait time specification is not long enough for any data to show up, nothing is returned.

i64OffsetResult

This output field carries the offset of the first data unit returned in the user-provided buffer. If no data is returned, this carries the offset of the most recent available unit in the target. This can be undefined if less than one unit of data exists in the target.

If the *iBytesGetMin* and *iBytesGetMax* of a request are both zero, this field returns the same value as it does for a request for one data unit.

iBytesResult

This output field carries the number of bytes of data in the return buffer. A positive value is always a multiple of *iBytesMultiple* and is never larger than *iBytesGetMax*. A value of zero indicates that nothing in the target meets the request, even though at least one unit (*iBytesMultiple*) of data does exist. In this case, *i64OffsetResult* contains the offset of the most recent available unit in the target. If less than one unit of data exists in the target, this value is -1 and *i64OffsetResult* is undefined.

Remarks

An application must fully initialize `TDapBufferPeek` before passing it to `DapBufferPeek`. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

See Also

`DapBufferPeek`, `DapStructPrepare`

TDapBufferPutEx

The `TDapBufferPutEx` structure defines the behavior of the `DapBufferPutEx` function.

```
typedef struct tag_TDapBufferPutEx {
    int iInfoSize;                       // Size of this structure
    int iBytesPut;                       // Number of bytes to put
    unsigned long dwTimeWait;            // Time interval to wait for space
    unsigned long dwTimeOut;             // Total time for entire operation
    int iBytesMultiple;                  // Bytes to put is a multiple of this
    int iReserved1;                      // Not used; must be zero
    } TDapBufferPutEx;
```

Members

iInfoSize

Specifies the size of this information structure.

iBytesPut

Specifies the number of bytes to put. It must be a multiple of *iBytesMultiple*.

dwTimeWait

Specifies the longest time in milliseconds to wait for available space to put data. If no space is available in this amount of time, the service aborts the operation.

dwTimeOut

Specifies the longest time in milliseconds to complete the entire operation. If the operation fails to complete in this amount of time, the service aborts the operation. When this member is non-zero, it takes precedence over *dwTimeWait*.

iBytesMultiple

Specifies that the number of bytes to put for *iBytesPut* be restricted to a multiple of this value.

Remarks

An application must fully initialize `TDapBufferPutEx` before passing it to `DapBufferPutEx`. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

The value of *iBytesPut* must be an integral multiple of *iBytesMultiple*; otherwise, an error occurs.

A zero value of *iBytesMultiple* is treated the same as one. The value of *iBytesMultiple* cannot be larger than the maximum pipe buffer size on the PC side (converted to bytes); otherwise, an error occurs.

See Also

`DapBufferPutEx`, `DapStructPrepare`

TDapCommandDownload

The `TDapCommandDownload` structure is used to define the behavior of the `DapCommandDownload` function.

```
typedef struct tag_TDapCommandDownload {
    int iInfoSize;                        // Size of this structure
    HDAP hdapSysPut;                      // $SysIn handle
    HDAP hdapSysGet;                      // $SysOut handle
    const char *pszCCFileName;            // Custom command filename
    const char *pszCCName;                // Custom command name
    int iCCStackSize;                     // Custom command stack size
    } TDapCommandDownload;
```

Members

iInfoSize
Specifies the size of the structure.

hdapSysPut
Handle open to `$SysIn` on a DAP board.

hdapSysGet
Handle open to `$SysOut` on a DAP board or zero.

pszCCFileName
File name for custom command.

pszCCName
Name for custom command or `NULL`.

iCCStackSize
Stack size for custom command.

Remarks

`TDapCommandDownload` must be fully initialized before calling the `DapCommandDownload` function. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

hdapSysPut must be open with write access to a com-pipe which is connected to `$SysIn` on a DAP board.

hdapSysGet may be open with read access to a com-pipe which is connected to `$SysOut` on the same DAP board.

pszCCFileName is the name and path to a custom command file to download. It may also be used to provide the name of the custom command.

pszCCName is the name for the custom command. It may be up to 11 characters and must consist of valid letters for a DAPL command name (A-Z_ followed by A-Z0-9_).

pszCCName may be `NULL`. In this case, the first 11 characters, excluding the extension, of *pszCCFileName* are used for the custom command name. The first 11 characters of *pszCCFileName* must form a valid DAPL symbol name.

iCCStackSize is the size in bytes for the custom command stack. *iCCStackSize* must be at least 1000; if it is not the size is increased automatically to 1000.

If *hdapSysGet* is 0, `DapCommandDownload` will not attempt to synchronize communication with the DAP board and will just download assuming that the DAP board is ready. This can be faster than fully synchronized download but it is quite dangerous. In most cases it is best to set *hdapSysGet* to a pipe open to `$SysOut` for a DAP board.

DAPL error handling is the responsibility of the caller if *hdapSysGet* is 0 because `DapCommandDownload` is unable to query the DAP board for error information.

When *hdapSysGet* is 0, it is critical to turn off the DAPL `OPTIONS SYSINECHO`, `TERMINAL`, and `PROMPT`. Otherwise, DAPL echoes data back to the application and the application may hang because it is not reading the data.

See Also

`DapCommandDownload`, `DapStructPrepare`

TDapHandleQuery

The `TDapHandleQuery` structure is used to define the behavior of the `DapHandleQuery` function.

```
typedef struct tag_TDapHandleQuery {
    int iInfoSize;                         // Size of this structure
    const char *pszQueryKey;               // Pointer to a query key string
    union {                                // Query result union
        unsigned long dw;                  // 32-bit return value
        char *psz;                         // Address of buffer for result strings
        void *pvoid;                       // Address of buffer for eResultType
     } QueryResult;                        //   data
    int iBufferSize;                       // Size of the buffer
    int eResultType;                       // Data type of return value
    } TDapHandleQuery;
```

Members

iInfoSize
Specifies the size of this structure.

pszQueryKey
Points to a null-terminated query key string.

QueryResult
Receives the result of the query. This is a union of *QueryResult.dw*, *QueryResult.psz* and *QueryResult.pvoid*. `DapHandleQuery` returns 32-bit binary values in *QueryResult.dw*. It returns strings in an application-supplied buffer addressed by *QueryResult.psz*. It returns data of the requested type in an application-supplied buffer addressed by *QueryResult.pvoid* if it receives *eResultType* specified as any value other than DAPIO_NONE.

iBufferSize
Specifies the size of the application-supplied buffer addressed by *QueryResult.psz* or *QueryResult.pvoid*. Specify zero to obtain the result in *QueryResult.dw* as a 32-bit binary value.

eResultType
Specifies the data type of the return value in an application-supplied buffer addressed by *QueryResult.pvoid.*

DAPIO_NONE	No type specified. If the application supplies a buffer, the service will return a string in the buffer addressed by *QueryResult.psz.*
DAPIO_BINARY	The service returns data in binary format in the buffer addressed by *QueryResult.pvoid.*
DAPIO_SZ	The service returns a string in the buffer addressed by *QueryResult.pvoid.*

`DAPIO_MULTI_SZ`	The service returns as a double null-terminated list of strings in the buffer addressed by *`QueryResult.pvoid`*.
`DAPIO_VARIANT`	A sub-set of the Windows `VARIANT` type that includes only scalar values. No pointer or reference types are supported. The service returns the result in the buffer addressed by *`QueryResult.pvoid`*. The buffer must be at least as large as the Windows API `VARIANT` data type.

Remarks

An application must fully initialize the `TDapHandleQuery` structure before calling the `DapHandleQuery` function. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

The member *`pszQueryKey`* must point to a null-terminated query key string.

The member *`iBufferSize`* must be set either to zero or to a positive value. If *`iBufferSize`* is zero, the result is a 32-bit binary value returned in *`QueryResult.dw`*. If *`iBufferSize`* is positive, it specifies the size, in bytes, of an application-supplied buffer. The application must initialize *`QueryResult.psz`* or *`QueryResult.pvoid`* to point to the buffer. The result is a double null-terminated list of character strings if *`eResultType`* is `DAPIO_NONE`; otherwise, the result is in the format as specified by *`eResultType`*.

A query can request the `DAPIO_VARIANT` result type for all keys that can return a scalar result. For queries whose result type is not clear in advance, use `DAPIO_VARIANT`.

To use this type, the caller must supply a buffer at least as large as the size of a `VARIANT` object. A typical practice is to allocate a `VARIANT` object and initialize it to empty by calling the Windows API `VariantInit` before passing it to the query through the `TDapHandleQuery` structure. Upon a successful return, check the type of the return variant and then access the result through the corresponding field of the variant. The following segment of code illustrates the use of this type.

```
// Allocate local query structures
VARIANT var;
TDapHandleQuery Q;

// Initialize the structures for a query
VariantInit(&var);
DapStructPrepare(Q);
Q.pszQueryKey = “DaplMemTotal”;
Q.iBufferSize = sizeof(var);
Q.QueryResult.pvoid = &var;
Q.eResultType = DAPIO_VARIANT;

// Make the query
if (DapHandleQuery(Handle, &Q))
    {
    // In this case, the result type is long, known in
    // advance. Simply access the result through
    // var.lVal. Otherwise, something like the following
    // should be done:
    // switch (var.vt) {
    //   case VT_I2: // access var.iVal
    //   case VT_I4: // access var.lVal
    //   case VT_R4: // access var.fltVal
    //   ...
    // }
    }
else
    {
    // handle error
    }
```

See Also

DapHandleQuery, DapStructPrepare

TDapIoInt64

DAPIO32 uses `TDapIoInt64` to represent a 64-bit integer type.

The `TDapIoInt64` definition as declared in `DAPIO32.h` is as follows:

```
#ifdef M_DapIoNoInt64
typedef struct tag_TDapIoInt64 {
   unsigned long dwLowPart;
   unsigned long dwHighPart;
} TDapIoInt64;
#else
   typedef __int64 TDapIoInt64;
#endif
```

Examples

Following are two examples about how to initialize the *`i64MaxCount`* field in the `TDapPipeDiskLog` structure to a value of 4294967296. The first example is for environments that support the `__int64` data type while the second example is for environments that do not support the `__int64` data type.

```
// Example 1: __int64 type is supported
#include <dapio32.h>

TDapPipeDiskLog dpdl;

DapStructPrepare(dpdl);
dpdl.i64MaxCount = 4294967296;


// Example 2: __int64 type is NOT supported
#define M_DapIoNoInt64 1
#include <dapio32.h>


TDapPipeDiskLog dpdl;

DapStructPrepare(dpdl);
dpdl.i64MaxCount.dwLowPart = 0;
dpdl.i64MaxCount.dwHighPart = 1;
```

TDapPipeDiskFeed

The `TDapPipeDiskFeed` structure defines the behavior of the `DapPipeDiskFeed` function.

```
typedef struct tag_TDapPipeDiskFeed {
    int iInfoSize;                          // Size of this structure
    unsigned long bmFlags;                  // Flags to control feeding behavior
    const char *pszFileName;                // Pointer to a name string
    unsigned long dwFileShareMode;          // Define how the disk file is read
    unsigned long dwFileFlagsAttributes; // Set attributes of the logfile
    unsigned long dwBlockSize;              // Size of block to read
    TDapIoInt64 i64MaxCount;                // Maximum number of bytes to read
    unsigned long dwReserved[16];           // Not used; must be zero
    } TDapPipeDiskFeed;
```

Members

iInfoSize
Specifies the size of this structure.

bmFlags
Specifies various disk-reading options.

pszFileName
Points to a null-terminated string that specifies the name of the data file to read.

dwFileShareMode
Specifies the file share properties of the disk data file.

dwFileFlagsAttributes
Specifies additional file attributes.

dwBlockSize
Specifies the minimum amount of data (in bytes) to read from the data file at one time when enough data are available. Use this field to optimize disk transfer. The default value is 8192.

i64MaxCount
Specifies the maximum number of bytes to feed. The default value is zero.

Remarks

An application must fully initialize `TDapPipeDiskFeed` before calling `DapPipeDiskFeed`. Use `DapStructPrepare` to prepare the structure before setting the specific fields of interest.

bmFlags specifies one of the following disk-feed options:

`dpdf_ServerSide`	The data file to be read resides on the same side of the network connection as the DAP board. The default is to assume the file resides on the same side of the network as the application (client side).

`dpdf_FlushBefore`	Flush the output data pipe before beginning the feeding session. Default action is to NOT flush the pipe before feeding.
`dpdf_ContinuousFeed`	The DAP board re-reads the data file from the beginning as soon as the end-of-file is reached. The default action is to stop feeding data at the end of the file.
`dpdf_BlockTransfer`	Instruct the `DapPipeDiskFeed` function to open the file with no intermediate buffering or caching and to access the file in a special way that is highly dependent on the target disk attributes to improve performance. To receive the expected performance improvement, use this option in conjunction with a very large *dwBlockSize* value (such as 1048576).

The member *pszFileName* points to a null-terminated string that specifies the name of the data file to read. The name can consist of a relative path or an absolute path. When `dpdf_ServerSide` is set, the service interprets the name from the server machine.

The *pszFileName* may contain a UNC file name using a network share name on the DAPcell server. Using this, a user on a client may select a file on the DAPcell server through the share without any knowledge of the server side disk layout. The DAPcell server will translate the share name to a local server side path so that it can read the file locally, allowing for high performance.

For example, if the DAPcell server name is `DAPcell1` and the data directory on the DAPcell server is `c:\data`, which is shared as `DATA`, then setting *pszFileName* to `\\DAPcell1\DATA\DataFile.dat` will cause the DAPcell server to read the file `c:\data\DataFile.dat` on the DAPcell server.

The member *dwFileShareMode* specifies the file share properties of the disk data file. The values allowed are:

`0`	The file cannot be used by another process.
`DAPIO_FILE_SHARE_READ`	The file can be read by another process.
`DAPIO_FILE_SHARE_WRITE`	The file can be written to by another process.

The member *dwFileFlagsAttributes* specifies additional file attributes. The possibilities are:

`DAPIO_FILE_ATTRIBUTE_NORMAL`	No special attributes.
`DAPIO_FILE_ATTRIBUTE_READONLY`	The file is read-only.
`DAPIO_FILE_ATTRIBUTE_ENCRYPTED`	The data in the file is encrypted.
`DAPIO_FILE_FLAG_SEQUENTIAL_SCAN`	Can be used to optimize the transfer of large blocks of data. Most applications will not need this flag.

The member *i64MaxCount* specifies the maximum number of bytes to feed. The default of zero causes feeding to continue until the end of the file is reached or, if `dpdf_ContinuousFeed` is set, indefinitely until the handle used to initiate the `DapPipeDiskFeed` command is closed using `DapHandleClose`.

If the end of the file is reached before the *i64MaxCount* value is reached, and `dpdl_ContinuousFeed` is not set, the I/O is terminated.

`TDapIoInt64` is by default `__int64` (64-bit integer). See the description of `TDapIoInt64` for more information if a particular compiler does not support the `__int64` type.

See the `DapPipeDiskFeed` reference for interactions between this field and the `TDapBufferPutEx` structure's *iBytesMultiple* field.

Version

The `DapPipeDiskFeed` service is only available in DAPcell Server and DAPcell Local Server version 4.00 or later. It is not available in DAPcell Basic Server.

See Also

`TDapIoInt64`, `DapPipeDiskFeed`, `DapStructPrepare`, `TDapBufferPutEx`

TDapPipeDiskLog

The TDapPipeDiskLog structure defines the behavior of the DapPipeDiskLog function.

```
typedef struct tag_TDapPipeDiskLog {
    int iInfoSize;                          // Size of this structure
    unsigned long bmFlags;                  // Flags to control logging behavior
    const char *pszFileName;                // Pointer to a null-terminated string
    unsigned long dwFileShareMode;          // Define how the logfile is shared
    unsigned long dwOpenFlags;              // Define how to open the logfile
    unsigned long dwFileFlagsAttributes; // Set attributes of the logfile
    TDapIoInt64 i64MaxCount;                // Maximum number of bytes to log
    unsigned long dwBlockSize;              // Size of block to write
    unsigned long dwReserved[16];           // Not used; must be zero
    } TDapPipeDiskLog;
```

Members

iInfoSize
Specifies the size of this information structure.

bmFlags
Specifies various logging options.

pszFileName
Points to a null-terminated string that specifies the name of the primary disk log file and the name of a possible mirror disk log file.

dwFileShareMode
Specifies the file share properties of the disk log file.

dwOpenFlags
Specifies how file opening is to be handled.

dwFileFlagsAttributes
Specifies additional file attributes.

i64MaxCount
Specifies the maximum number of bytes to log. The default value is zero.

dwBlockSize
Specifies the minimum amount of data (in bytes) to write to the log file at one time. This field is provided for disk transfer optimization. The default value is 8192.

Remarks

bmFlags specifies one of the following disk-log options:

dpdl_ServerSide	Logging is to take place on the same side of the network connection as the DAP board. If not specified, logging will take place on the application (client) side of the network connection.

dpdl_FlushBefore	Flush the input data pipe before beginning the logging session. Default action is to NOT flush the pipes before logging. This should be done before issuing the START command to DAPL or data may be lost.
dpdl_FlushAfter	Flush the input pipe after the logging session has terminated. Default action is to NOT flush the pipes after logging. To ensure proper operation, an application should issue a STOP command to the DAP board before closing the disk logging session or the input pipe may fill up, causing flushing to fail.
dpdl_MirrorLog	Enable mirror logging. Mirror logging creates a copy of the logged data in another file.
dpdl_AppendData	Allows new data to be appended to an existing file. The only *dwOpenFlags* that can be used for appending are DAPIO_OPEN_ALWAYS and DAPIO_OPEN_EXISTING.
dpdl_BlockTransfer	Instruct the DapPipeDiskLog function to open the file with no intermediate buffering or caching and to access the file in a special way that is highly dependent on the target disk attributes to improve performance. This transfer mode, however, adds overhead to slow rate transfer with small buffers. It should only be used when necessary with a very large *dwBlockSize* value (such as 1048576 and above).

The member *pszFileName* points to a null-terminated string that specifies the name of the primary disk log file and the name of a mirror disk log file as well, if mirror logging is enabled. Separate multiple file names with semi-colons.

Currently, only one mirror file is allowed. Both files must be on the same side of the DAPcell/DAPcell Local service (the PC application side or the DAP side). When dpdl_ServerSide is set, the files named in *pszFileName* are interpreted from the server machine.

The *pszFileName* may contain a network share on the DAPcell server. Using this, a file on the DAPcell server may be selected by the client through the share without any knowledge of the server side disk layout. The DAPcell server will translate the share name to a local server-side path so that it can log to the file locally, allowing for high performance.

For example, if the DAPcell server name is DAPcell1 and the data directory on the DAPcell server is c:\data, which is shared as DATA, then setting *pszFileName* to \\DAPcell1\DATA\DataFile.dat will cause the DAPcell server to log data to the file c:\data\DataFile.dat on the DAPcell server.

The member *dwFileShareMode* specifies the file share properties of the disk log file. The values allowed are:

0	The file cannot be used by another process.
DAPIO_FILE_SHARE_READ	The file can be read by another process.
DAPIO_FILE_SHARE_WRITE	The file can be written to by another process.

The *dwOpenFlags* member specifies how file opening is to be handled. The possibilities are:

`DAPIO_CREATE_NEW`	Create a new file. Creation fails if the file already exists.
`DAPIO_CREATE_ALWAYS`	Create a new file. If the file already exists, it is overwritten.
`DAPIO_OPEN_ALWAYS`	Open an existing file. If the file does not exist, it will be created.
`DAPIO_OPEN_EXISTING`	Open an existing file without resetting permissions. Opening fails if the file does not exist.

The *dwFileFlagsAttributes* member specifies additional file attributes. The possibilities are:

`DAPIO_FILE_ATTRIBUTE_NORMAL`	No special attributes.
`DAPIO_FILE_ATTRIBUTE_ENCRYPTED`	The data in the file is encrypted.
`DAPIO_FILE_FLAG_WRITE_THROUGH`	Write through any intermediate caching and go directly to disk.
`DAPIO_FILE_FLAG_SEQUENTIAL_SCAN`	Can be used to optimize the transfer of large blocks of data. Most applications will not need this flag.

The member *i64MaxCount* specifies the maximum number of bytes to feed. The default zero causes logging to continue indefinitely until the handle used to initiate the `DapPipeDiskLog` command is closed using `DapHandleClose`. See the `DapPipeDiskLog` description for interactions between this field and the `TDapBufferGetEx` *iBytesMultiple* field.

`TDapIoInt64` is by default `__int64` (64-bit integer). See the description of `TDapIoInt64` for more information if a particular compiler does not support the `__int64` type.

After one of its target disks has failed or become full, `DapPipeDiskLog` will keep logging to other target disks, if there are any. Even when all target disks have failed or become full, `DapPipeDiskLog` will keep reading data from the source Data Acquisition Processor until either the log request count is covered or the log handle is closed.

Version

The `DapPipeDiskLog` service is only available in DAPcell Server and DAPcell Local Server version 4.00 or later. It is not available in DAPcell Basic Server.

See Also

`TDapIoInt64`, `DapPipeDiskLog`, `DapStructPrepare`, `TDapBufferGetEx`

DAPIO32 Function Reference

A Data Acquisition Processor simplifies writing data acquisition applications by handling data buffering, real-time control, and real-time processing. Programs in the PC are responsible for user interaction, graphics, and disk logging.

Function Reference

`DapBufferGet`	`DapInt16Get`
`DapBufferGetEx`	`DapInt16Put`
`DapBufferPeek`	`DapInt32Get`
`DapBufferPut`	`DapInt32Put`
`DapBufferPutEx`	`DapLastErrorTextGet`
`DapCharGet`	`DapLineGet`
`DapCharPut`	`DapLinePut`
`DapCommandDownload`	`DapModuleInstall`
`DapComPipeCreate`	`DapModuleLoad`
`DapComPipeDelete`	`DapModuleUninstall`
`DapConfig`	`DapModuleUnload`
`DapConfigParamsClear`	`DapOutputEmpty`
`DapConfigParamSet`	`DapOutputSpace`
`DapConfigRedirect`	`DapPipeDiskFeed`
`DapHandleClose`	`DapPipeDiskLog`
`DapHandleOpen`	`DapReinitialize`
`DapHandleQuery`	`DapReset`
`DapHandleQueryInt32`	`DapServerControl`
`DapHandleQueryInt64`	`DapStringFormat`
`DapInputAvail`	`DapStringGet`
`DapInputFlush`	`DapStringPut`
`DapInputFlushEx`	`DapStructPrepare`

Function Overview

The DAPIO32 interface provides a complete set of functions for communicating with a Data Acquisition Processor. Each function falls into one of several categories.

Most DAPIO32 services require a handle to identify the target of the operation. Handles are obtained using the `DapHandleOpen` service.

Communication with a Data Acquisition Processor through DAPIO32.DLL is established by opening one or more handles to communication pipes. Commands and data are transferred through the communication pipes using DAPIO32 services.

In Windows, any 32-bit/64-bit programming language that supports DLL calls can use the DAPIO32.DLL services.

Category	Services
Handle services	`DapHandleOpen`, `DapHandleClose`
Basic I/O services	`DapBufferGet`, `DapBufferPut`, `DapCharGet`, `DapCharPut`, `DapInputFlush`, `DapInt16Get`, `DapInt16Put`, `DapInt32Get`, `DapInt32Put`, `DapLineGet`, `DapLinePut`, `DapStringFormat`, `DapStringGet`, `DapStringPut`
Advanced I/O services	`DapBufferGetEx`, `DapBufferPeek`, `DapBufferPutEx`, `DapInputFlushEx`
Disk I/O services	`DapPipeDiskFeed`, `DapPipeDiskLog`
Initialization services	`DapReinitialize`, `DapReset`, `DapStructPrepare`
Configuration services	`DapCommandDownLoad`, `DapConfig`, `DapConfigParamsClear`, `DapConfigParamSet`, `DapConfigRedirect`, `DapComPipeCreate`, `DapComPipeDelete`
Module services	`DapModuleInstall`, `DapModuleLoad`, `DapModuleUninstall`, `DapModuleUnload`
Server services	`DapServerControl`
Information services	`DapLastErrorTextGet`, `DapHandleQuery`, `DapHandleQueryInt32`, `DapHandleQueryInt64`, `DapInputAvail`, `DapOutputEmpty`, `DapOutputSpace`

Most DAPIO32 services require a handle to identify the target of the operation. Handles are obtained using the `DapHandleOpen` service.

An application communicates with a Data Acquisition Processor through DAPIO32.DLL by opening one or more handles to communication pipes. It sends and receives commands and data through the communication pipes using DAPIO32 services.

Any 32-bit Windows programming language or application that can call DLL functions can use the DAPIO32.DLL services.

Data I/O Time-out

All of the basic I/O services have a built-in time-out. If a service is unable to process data for more than 20 seconds, the service aborts the operation. For get services, this means that the Data Acquisition Processor did not send data for more than 20 seconds. For put services, this means that the Data Acquisition Processor did not accept data for more than 20 seconds. An application cannot change the basic I/O time-out.

The advanced I/O services allow an application to set the time-out to fit the application needs.

Alphabetical Function Reference

Following is a complete alphabetical listing of all DAPIO32 services.

See Also

DAPIO32 Structure Reference

DapBufferGet

The `DapBufferGet` function reads a block of data from the target pipe.

```
int __stdcall DapBufferGet(
    HDAP hAccel,                    // Open handle to the target pipe
    int iLength,                    // Number of bytes to read
    void *pvBuffer                  // Address of buffer to receive data
    );
```

Parameters

hAccel

Specifies an open handle to the target pipe. Requires a handle opened with read access. The target pipe must be an output pipe from the Data Acquisition Processor.

iLength

Specifies the number of data bytes to read.

pvBuffer

Points to the buffer that receives data.

Return Values

If the function succeeds, the return value is the number of bytes actually read. The result is in the range zero to *iLength*.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapBufferGet` attempts to read all requested data from the target pipe. It will wait for up to 20 seconds per new data item. If there are no data available for more than 20 seconds, it returns with the number of bytes read so far.

An application which can not guarantee data availability should avoid waiting for data by using `DapBufferGetEx`
.

See Also

`DapBufferGetEx`, `DapInputAvail`

DapBufferGetEx

The `DapBufferGetEx` function reads a block of data from the target pipe. It allows an application to specify a range of data bytes to read. It will read as many data bytes as possible within the specified range. It also allows an application to specify two time-out parameters to control the behavior of the function. The function will return immediately with the data read so far if a time-out occurs.

```
int __stdcall DapBufferGetEx(
    HDAP hAccel,                          // Open handle to the target pipe
    const TDapBufferGetEx *pGetInfo,      // Address of get structure
    void *pvBuffer                        // Address of buffer to receive data
    );
```

Parameters

hAccel

Specifies the open handle to the target pipe. Requires a handle opened with read access. The target pipe must be an output pipe from the Data Acquisition Processor.

pGetInfo

Points to a `TDapBufferGetEx` structure that passes the parameters of the get operation into the function.

pvBuffer

Points to the buffer that receives data.

Return Values

If the function succeeds, the return value is the number of data bytes actually read.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

The `DapBufferGetEx` function is an extended version of the `DapBufferGet` function. It allows a minimum number of requested bytes *iBytesGetMin*, a maximum number of requested bytes *iBytesGetMax*, a multiple size for the requested bytes *iBytesMultiple*, a maximum time interval in milliseconds to wait for new data *dwTimeWait*, and a maximum time-out interval in milliseconds for the entire operation *dwTimeOut*. These parameters are passed through the `TDapBufferGetEx` structure, pointed to by *pGetInfo*.

The allocated buffer that receives data, pointed to by *pvBuffer*, must be at least *iBytesGetMax* in size. If the allocated buffer is not large enough, there may be random failures.

iBytesGetMin must be greater than or equal to zero, and *iBytesGetMax* must be greater than or equal to *iBytesGetMin*. Both *iBytesGetMin* and *iBytesGetMax* must be an integral multiple of *iBytesMultiple*.

iBytesMultiple must be an integral multiple of the communication pipe width referenced by *hAccel*. A zero value of *iBytesMultiple* is treated the same as the communication pipe width.

The value of *iBytesMultiple* must be less than or equal to (1) the PC side maximum pipe buffer size in bytes minus 1024 or (2) the PC side maximum pipe buffer size in bytes minus the DAP side blocking size in bytes, whichever is smaller. If `iBytesMultiple` does not satisfy these requirements, the first condition causes an error, and the second condition may cause a deadlock. Since the second condition is not checked, it is the application's responsibility to guarantee that this never happens.

`DapBufferGetEx` returns when one of the following conditions occurs:

- `iBytesGetMin` is satisfied
- `dwTimeOut` is reached (`dwTimeOut` > 0)
- no new data arrive during `dwTimeWait`

When `DapBufferGetEx` returns, it returns with all available data up to `iBytesGetMax`. It could return anything from zero to `iBytesGetMax`, but the number of bytes returned is always an integral multiple of `iBytesMultiple`. If no data are available (which includes not enough data to satisfy `iBytesMultiple`) when `DapBufferGetEx` returns, it returns zero.

If `iBytesGetMin` is available before `dwTimeOut` or `dwTimeWait` is reached, `DapBufferGetEx` returns with all available data up to `iBytesGetMax`. If `iBytesGetMin` is zero, `DapBufferGetEx` returns with all available data up to `iBytesGetMax` without waiting.

`dwTimeOut > dwTimeWait`
If `dwTimeOut` is greater than `dwTimeWait`, `DapBufferGetEx` waits up to `dwTimeOut` to satisfy `iBytesGetMin` before returning. The length of time `DapBufferGetEx` waits depends on whether any data arrive during `dwTimeWait` intervals. If some data arrive during `dwTimeWait`, `DapBufferGetEx` waits again for `dwTimeWait`, and this continues until `dwTimeOut` is reached or `iBytesGetMin` is satisfied, whichever occurs first. If, at any point before `dwTimeOut` is reached or `iBytesGetMin` is satisfied, no data arrive during an interval of `dwTimeWait`, `DapBufferGetEx` returns immediately.

`dwTimeOut <= dwTimeWait and dwTimeOut > 0`
If `dwTimeOut` is greater than zero but smaller than or the same as `dwTimeWait`, `dwTimeOut` takes precedence over `dwTimeWait`. `dwTimeWait` is not used.

`dwTimeOut = 0 and dwTimeWait > 0`
`dwTimeOut` of zero means never time-out if some data arrive during `dwTimeWait`. The affect of this is waiting indefinitely to satisfy `iBytesGetMin` as long as some data arrive during `dwTimeWait` intervals. If, at any point before `iBytesGetMin` is satisfied, no data arrive during any interval of `dwTimeWait`, `DapBufferGetEx` returns immediately.

`dwTimeOut = 0 and dwTimeWait = 0`
`dwTimeOut` of zero and `dwTimeWait` of zero means returning immediately with all available data up to `iBytesGetMax`.

See Also

`TDapBufferGetEx`, `DapBufferGet`

DapBufferPeek

The `DapBufferPeek` function peeks the pipe or the disk log file associated with the handle for data. This operation does not affect the data integrity of the pipe or disk file.

```
BOOL __stdcall DapBufferPeek(
   HDAP hAccel,                      // Peek handle
   TDapBufferPeek *pPeekInfo         // Pointer to a peek information block
   );
```

Parameters

hAccel

Identifies the handle of the target.

pPeekInfo

Pointer to a peek information block. The block defines the behavior of the function and carries its return values. See the description of `TDapBufferPeek` structure for more information.

Return Values

Returns true if the operation is successful. Data is returned in the user-supplied buffer pointed to by *pPeekInfo->pBuffer*. The size of return data is in *pPeekInfo->iBytesResult*. The starting offset of data is in *pPeek->i64OffsetResult*.

Returns false if an error has occurred. All the other return values in the peek information block are undefined. To determine the cause of an error, use the `DapLastErrorTextGet` function.

Remarks

If the handle is a normal pipe handle, the target is the pipe. The `DapBufferPeek` function peeks the history of the pipe if it is still available. Using a query handle allows multiple readers to peek the history of the pipe at the same time.

If the handle is a disk I/O handle, the target is the primary disk file of the current active disk I/O session. The `DapBufferPeek` function reads data from the disk file and possibly buffered data yet to be written to the disk without affecting the on-going disk I/O. This feature is not available in DAPcell Basic Server.

Data transfer is always done in data units. The size of a data unit is equal to the value specified in *pPeekInfo->iBytesMultiple* or the underlining target data type width if *pPeekInfo->iBytesMultiple* is zero.

When `DapBufferPeek` returns successfully, the return value in *pPeekInfo->iBytesResult* can be a positive integer, zero, or -1.

When it is positive, this value is typically between *iBytesGetMin* and *iBytesGetMax*, but can be smaller if a time-out has occurred. The starting offset or return data is in *pPeek->i64OffsetResult*.

If *pPeekInfo->iBytesResult* is zero, no data is returned. This implies that none of the data requested is available. The field *pPeek->i64OffsetResult* carries the offset of the most recent unit of data available in the target.

A value of -1 returned in the *pPeekInfo->iBytesResult* indicates that less than one unit of data exists in the target. In this case, *pPeek->i64OffsetResult* is undefined.

Version

This service is only available in DAPcell version 4.14 or later. Not all functions are available in DAPcell Basic Server.

See Also

`TDapBufferPeek`, `DapBufferGetEx`, `DapHandleOpen`, `DapPipeDiskFeed`, `DapPipeDiskLog`

DapBufferPut

The `DapBufferPut` function writes a block of data to the target pipe.

```
int __stdcall DapBufferPut(
    HDAP hAccel,                          // Open handle to the target pipe
    int iLength,                          // Number of data bytes in the block
    const void *pvBuffer                  // Address to the block of data
    );
```

Parameters

hAccel

Specifies the open handle to the target pipe. Requires a handle opened with write access. The target pipe must be an input pipe to the Data Acquisition Processor.

iLength

Specifies the number of data bytes to write.

pvBuffer

Points to the block of data to write.

Return Values

If the function succeeds, the return value is the number of data bytes actually written. The result is in the range zero to *iLength*.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapBufferPut` attempts to write the entire block of data to the target pipe. It will wait for up to 20 seconds for space to place each data item. If there is no space for more than 20 seconds, it returns with the number of bytes written so far.

An application that cannot guarantee space in the com-pipe should avoid waiting for space by using `DapBufferPutEx`.

See Also

`DapOutputSpace`

DapBufferPutEx

The `DapBufferPutEx` function writes a block of data to the target pipe. It allows an application to specify two time-out parameters to control the behavior of the function. The function will return immediately if a time-out occurs.

```
int __stdcall DapBufferPutEx(
    HDAP hAccel,                              // Open handle to the target pipe
    const TDapBufferPutEx *pPutInfo,          // Address of put structure
    const void *pvBuffer                      // Address of buffer to provide data
    );
```

Parameters

hAccel
: Specifies the open handle to the target pipe. Requires a handle opened with write access. The target pipe must be an input pipe to the Data Acquisition Processor.

pPutInfo
: Points to the `TDapBufferPutEx` structure that passes the parameters of the put operation into the function.

pvBuffer
: Points to the buffer that provides data.

Return Values

If the function succeeds, the return value is the number of data bytes actually written.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This function is an extension of the `DapBufferPut` function.

The function tries to put the number of data bytes into the target pipe as specified in the member *iBytesPut* of the `TDapBufferPutEx` structure. If it succeeds completely, its return value is equal to the value of *iBytesPut*. The value of *iBytesPut* must be an integral multiple of *iBytesMultiple*.

The function will wait for space if the target pipe becomes full before it completes writing all data. In this case, the two members of the `TDapBufferPutEx` structure, *dwTimeOut* and *dwTimeWait*, determine the behavior of the function. If the put operation fails to complete in *dwTimeOut* milliseconds, or if the pipe remains full for *dwTimeWait* milliseconds, the function returns immediately.

The return value is then the number of bytes actually written up to the point where the service aborted the operation. It can be zero or any integral multiple of *iBytesMultiple* less than *iBytesPut*. An application can check the return value to determine if a time-out has occurred.

See Also

`TDapBufferPutEx`, `DapBufferPut`

DapCharGet

The `DapCharGet` function reads a single character from a DAP com-pipe.

```
BOOL __stdcall DapCharGet(
    HDAP hAccel,                    // Open handle to the target pipe
    char *pch                       // Location to receive character
    );
```

Parameters

hAccel
Handle to DAP com-pipe.

pch
Pointer to the location to receive the character.

Return Values

If the function succeeds, the return value is TRUE; the service read the character.

If the function fails, the return value is FALSE. Something is wrong with the handle or *pch* is `NULL`. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapCharPut

The `DapCharPut` function writes a single character to a DAP com-pipe.

```
BOOL __stdcall DapCharPut(
    HDAP hAccel,                        // Open handle to the target pipe
    char ch                             // Character to write
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

ch

Character to write to DAP com-pipe.

Return Values

If the function succeeds, the return value is TRUE; the service wrote the character.

If the function fails, the return value is FALSE. Something is wrong with the handle. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapCommandDownload

The `DapCommandDownload` function downloads a custom command to DAPL.

Note: Use `DapCommandDownload` to load 16-bit custom command binaries to a DAP board. Do not use this command to load 32-bit command modules to a DAP board. Install 32-bit command modules using the Data Acquisition Processor control panel application. For advanced application use `DapModuleInstall` and related services.

```
BOOL __stdcall DapCommandDownload(
   const TDapCommandDownload *pdcdl  // Pointer to structure
   );
```

Parameters

pdcdl

A pointer to a `TDapCommandDownload` structure that describes the command to download.

Return Values

If the function succeeds, the return value is TRUE; the service downloaded the custom command without error.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Use `DapStructPrepare` to fully initialize the `TDapCommandDownload` structure before calling this function.

Downloads a custom command binary to a DAP board. The command binary must be developed using the Developer's Toolkit for DAPL and must be compiled for the operating system on the DAP board (DAPL 4.x or DAPL 2000).

Contact your Microstar Laboratories product supplier for information on the Developer's Toolkit for DAPL if you wish to develop custom commands.

This is a `DESTRUCTIVE` operation. It resets the DAP board before performing the download operation. This means that all user-defined symbols are erased after a call to this function (See documentation for the DAPL `RESET` command). An application should not count on this behavior, however, because it is not guaranteed in future versions of this service.

See Also

`TDapCommandDownload`

DapComPipeCreate

The `DapComPipeCreate` function instructs DAPIO32 to create a communication pipe channel between the PC and a Data Acquisition Processor. It physically creates pipes both on the PC and on the Data Acquisition Processor. This function can be destructive.

```
BOOL __stdcall DapComPipeCreate(
    const char *pszPipeInfo                 // Address of pipe info string
    );
```

Parameters

pszPipeInfo

Points to a null-terminated string that specifies the name and attributes of the pipe channel to create.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This function requires exclusive access to the target DAP. `DapComPipeCreate` fails if the target DAP has been opened with read or write access, or if any of its pipes has been opened with any access.

The pipe information string consists of the UNC pipe name of the pipe to create and an optional list of attributes.

A UNC pipe name takes the form of `\\<PcName>\<DapName>\<PipeName>`. `<PcName>` is replaced by the computer name of the host PC or "." if the host PC is local. `<DapName>` is replaced by `Dap0`, `Dap1`, ..., or `DapX`, …, based on the number of Data Acquisition Processors installed on the system and which Data Acquisition Processor the operation is intended for. `<PipeName>` must be one of the following:

```
$SysIn                          $SysOut
$BinIn                          $BinOut
Cp2In                           Cp2Out
Cp3In                           Cp3Out
Cp4In                           Cp4Out
      ...
Cp31In                          Cp31Out
```

These pre-defined names carry the information of a target Data Acquisition Processor pipe number as well as the transfer direction. On the Data Acquisition Processor, each communication pipe is associated with an integer number and is declared as either input or output. `$SysIn` and `$SysOut` are the default input and output pipes with the number of zero while `$BinIn` and `$BinOut` are the default input and output pipes with the number of one. A maximum of 32 pipes on each Data Acquisition Processor are supported; therefore, the largest number that can be associated with a Data Acquisition Processor communication pipe is 31.

When DapComPipeCreate creates a communication channel, it first creates a communication pipe in the PC. Then it creates the paired communication pipe on the Data Acquisition Processor. After a successful return from this function, the target communication pipe is fully functional. On ISA Data Acquisition Processors where the DAPL command CPIPE is available, it is not necessary or appropriate to create the paired communication pipe on the Data Acquisition Processor separately using the command.

An optional pipe attribute list can follow the pipe name as part of the information string. The list must be enclosed in square brackets. The list allows applications to specify pipe attributes for both the PC side and the Data Acquisition Processor side. A vertical bar separates the DAP side attributes on the right from the PC side attributes on the left. If the right side is empty, the vertical bar can be omitted. (Note: there was an older syntax for the attribute list used prior to this interface 2.0. The older syntax is still supported for compatibility, but a mixed use of the old and new syntax will be rejected.)

A summary of the optional pipe attribute list syntax is present below:

```
[... | ...]
```

The PC side attributes are specified to the left of the vertical bar. They can be absent if default values are assumed. The supported attributes are:

`type = xxx` --- `xxx` is one of the types listed on the right:	"`byte`" "`word`" "`long`" "`float`" "`double`" "`text`" The type attribute describes the data type of the pipe for both PC and DAP sides. It also implies "`width`": "`byte`" and "`text`" are 1 byte, "`word`" is 2 bytes, "`long`" and "`float`" are 4 bytes, and "`double`" is 8 bytes. When it's absent, type defaults to "`word`" (note that not all types are supported by all versions of DAPL 2000).
`maxsize = <integer>`	Maximum pipe buffer size in unit of elements. The maximum size converted to bytes has to be at least 1024. No upper limit is imposed at the syntax level. The actual upper limit is system dependent. When it's absent, it assumes some default value that, in most cases, offers better performance.

The DAP side attributes are specified to the right of the vertical bar. They can be absent if default values are assumed. If none of the attributes is specified, the vertical bar can also be absent. The supported attributes are:

`maxsize = <integer>`	Maximum pipe buffer size in unit of elements. The maximum size converted to bytes has to be at least 1024. No upper limit is imposed at the syntax level. The upper limit is DAP model dependent. When it's absent, it defaults to that on the PC side.

`blocking = <integer>`	Number of elements available in the pipe buffer before data are transmitted to the PC side pipe buffer. The amount of data in each transfer is always an integral multiple of this value. This attribute is only valid for non-text DAP output pipes. Blocking size must be larger than zero and smaller than the pipe buffer size on either side. The upper limit is also DAP model dependent. When it's absent, blocking size defaults to one.

If a pipe identical to the pipe to create already exists when this function is called, the function returns success without doing anything. Two pipes are considered identical if their names are identical and their pipe attributes are identical as well.

Otherwise, the function creates a new pipe or overrides the existing pipe. This operation is `DESTRUCTIVE` if it overrides the existing pipe that is being used. In this case, application data and configuration information on the target Data Acquisition Processor are lost. For this reason, `DapComPipeCreate` is typically called at the very beginning of the system initialization.

Once communication pipes are created, they are persistent even across system reboots until explicitly removed using `DapComPipeDelete`. To maintain consistency between the PC and the Data Acquisition Processor, it is important not to create or destroy communication pipes without calling `DapComPipeCreate` or `DapComPipeDelete`.

Example

Following is an example of creating a communication channel that is connected to the communication pipe `Cp4In` on the Data Acquisition Processor `DAP0` on the local machine. It is an input binary pipe to the Data Acquisition Processor. The pipe type is "`word`" (width= 2), and the maximum pipe size is 2048 in the PC and 4096 on the Data Acquisition Processor.

```
DapComPipeCreate("\\\\.\\Dap0\\Cp4In [type=word maxsize=2048 | maxsize=4096]");
```

See Also

`DapComPipeDelete`

DapComPipeDelete

The `DapComPipeDelete` function instructs DAPIO32 to remove the communication channel that bears the specified UNC name. This function can be destructive.

```
BOOL __stdcall DapComPipeDelete(
    const char *pszPipeInfo                // Address of pipe info string
    );
```

Parameters

pszPipeInfo

Points to a null-terminated string that specifies the name of the pipe channel to delete.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This function requires exclusive access to the target DAP. `DapComPipeDelete` fails if the target DAP has been opened with read or write access, or if any of its pipes have been opened with any access.

`DapComPipeDelete` removes the communication channel that bears the same name as is specified in the pipe information string. It takes the same pipe information string as `DapComPipeCreate` does, but ignores the optional pipe attribute list.

If the target communication channel does not exist, this function returns success without doing anything.

Otherwise, this function physically removes the paired pipes from the PC and from the Data Acquisition Processor. This operation is `DESTRUCTIVE`. All data in the target pipe are lost. If the target pipe is being used at the time of removal, application and configuration data on the target Data Acquisition Processor are lost as well.

See Also

`DapComPipeCreate`

DapConfig

The `DapConfig` function sends a file to a DAP com-pipe with parameter substitution.

```
BOOL __stdcall DapConfig(
    HDAP hAccel,                    // Open handle to the target pipe
    const char *pszDaplFilename     // Name of file to send
    );
```

Parameters

hAccel
Handle to DAP com-pipe open for writing.

pszFilename
Name of file to send to DAP com-pipe.

Return Values

If the function succeeds, the return value is TRUE; the complete file was successfully sent to DAP com-pipe.

If the function fails, the return value is FALSE; something went wrong sending the file. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Sends a DAPL command file to the DAPL interpreter through the com-pipe addressed by *hAccel*.

If the file contains replaceable parameters of the form `%N` where `N` is a number in the range 1 to 100, replaces occurrences of `%N` with either default text set in the DAPL file itself or with text set by a program using the `DapConfigParamSet` service.

If the file contains any errors in replaceable parameters, the processing of the file is aborted at a line boundary and an error is reported. That is, complete lines are sent to the DAP com-pipe, but the complete file may not be sent to the DAP com-pipe.

Default parameters in a DAPL file are set using the '//;%DEFAULT' token.

It takes one of two forms:

```
//;%DEFAULT %N=TextWithoutSpaces
//;%DEFAULT %N="text with spaces"
```

The DAPL interpreter considers all text after semi-colon (//) up to the end of line a comment. This is true also of the '//;%DEFAULT' token; `DEFAULT` is only processed by `DapConfig` if it is a comment to the DAPL interpreter.

A default parameter definition may be defined on the same line where it is used. That means that the definition actually follows the use of the parameter on the line. This, however, can be quite useful since it lets you see the definition where it is used.

For example:

```
IDEF A
  CHANNELS 1
  TIME %45   //;%DEFAULT %45=1000
  . . .
```

will set the time for input procedure `A` to 1000 us if there is no program definition. If there is a program definition, it takes precedence over the default. A default parameter may not be defined on a line following the line where it is used.

If for a given parameter `%N` both a program-defined definition and a default definition exists, the program-defined definition takes precedence.

So, in the example above, if the program uses `DapConfigParamSet` (45, 500) and then calls `DapConfig`, the time for input configuration `A` will be 500 us, not the 1000 us set by the default.

There is only one set of parameters for a given process that uses the DAPIO32 DLL. This includes systems with multiple DAP boards, so to send files to more than one DAP board without interactions in the default parameters between them call `DapConfigParamsClear` before calling `DapConfig`.

Parameter definitions persist across multiple calls to `DapConfig`. This means that the default definitions in one DAPL file can affect the text sent to a DAP board using a second DAPL file depending on the order in which the files are sent.

See Also

`DapConfigParamsClear`, `DapConfigParamSet`, `DapConfigRedirect`

DapConfigParamsClear

The `DapConfigParamsClear` function clears all program-defined parameters and default parameters.

```
BOOL __stdcall DapConfigParamsClear(
    void
    );
```

Parameters

None.

Return Values

If the function succeeds, the return value is TRUE; the parameters were cleared.

If the function fails, the return value is FALSE. Something went wrong clearing parameters. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Clears all program-defined parameters and default replaceable parameters.

While this function returns a result, it currently never returns failure. Future versions may return failure, so it is worth checking the result.

See Also

`DapConfigParamSet`

DapConfigParamSet

The `DapConfigParamSet` function initializes a program-defined parameter.

```
BOOL __stdcall DapConfigParamSet(
    int iParamNumber,                    // Parameter number (1 - 100)
    const char *pszParam                 // Parameter
    );
```

Parameters

iParamNumber

Parameter number in the range 1 to 100.

pszParam

Null-terminated string for parameter.

Return Values

If the function succeeds, the return value is TRUE; the parameter was set.

If the function fails, the return value is FALSE; something went wrong setting the parameter. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

If *pszParam* is `NULL`, the parameter is cleared.

Program-defined parameters take precedence over default parameters set in the DAPL file.

See Also

`DapConfig`, `DapConfigParamsClear`

DapConfigRedirect

The `DapConfigRedirect` function redirects the output of `DapConfig` to a text file rather than sending the configuration file to a DAP com-pipe.

```
BOOL __stdcall DapConfigRedirect(
    const char *pszFilename                    // Destination filename
    );
```

Parameters

pszFilename

Name of destination file or `NULL`.

Return Values

If the function succeeds, the return value is TRUE; *pszFilename* was opened.

If the function fails, the return value is FALSE; *pszFilename* could not be opened.

Remarks

If *pszFilename* is `NULL` or `""`, `DapConfigRedirect` closes any existing redirection.

`DapConfigRedirect` is meant primarily for testing. It allow an application developer to redirect output of a DAPL configuration file through the parameter substitution performed by `DapConfig` without sending anything to the DAP board.

It is also sometimes useful to redirect output to a file and then send the processed file line by line to the DAP board. One use for this is in an interactive environment where you wish to display the echo from the DAPL interpreter as the lines are sent to it.

See Also

`DapConfig`

DapHandleClose

The `DapHandleClose` function releases a handle previously opened with `DapHandleOpen`.

```
BOOL __stdcall DapHandleClose(
    HDAP hAccel                                   // The handle to close
    );
```

Parameters

hAccel

Specifies the handle to close. It must be a handle previously returned by `DapHandleOpen`.

Return Values

If the function succeeds, the return value is TRUE,

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

See Also

`DapHandleOpen`

DapHandleOpen

The `DapHandleOpen` function returns a handle to the target with the specified name. The target can be one of the following: a server PC, a Data Acquisition Processor on a server, or a communication pipe on a Data Acquisition Processor.

```
HDAP __stdcall DapHandleOpen(
   const char *pszAccelName,          // The UNC name of the pipe to open
   unsigned long ulOpenFlags          // Open attributes
   );
```

Parameters

pszAccelName

Points to a UNC target name string that specifies the target to open.

ulOpenFlags

Specifies the desired access to acquire. An application can acquire read access to an output pipe from a Data Acquisition Processor, write access to an input pipe to a Data Acquisition Processor, or query access to a server PC, a Data Acquisition Processor, or a pipe.

Values	Description
DAPOPEN_READ	Specifies the read access to a pipe. Data can only be read from the pipe.
DAPOPEN_WRITE	Specifies the write access to a pipe. Data can only be written to the pipe.
DAPOPEN_QUERY	Specifies the query access to a server PC, a Data Acquisition Processor, or a pipe. A handle opened with query access can only be used to retrieve static information about the target the handle is associated with.
DAPOPEN_DISKIO	Specifies the disk I/O access with a pipe.

Return Values

If the function succeeds, the return value is an open handle to the specified target.

If the function fails, the return value is a `NULL` handle (of value zero). Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

An application can open a communication pipe with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute. Opening a pipe with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute reserves the pipe for exclusive use. No one can open the same pipe again with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute until the application that owns the handle closes it by calling `DapHandleClose`.

An application can also open a server or a DAP with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute. Opening a server or a DAP with either attribute reserves the target for exclusive use. Once the target is reserved, no one can open it again or open any target under it with either attribute until the handle is closed. For example, the function `DapReinitialize` requires a server handle opened with the `DAPOPEN_WRITE` attribute, to reload DAPL to all DAPs on a server, or a DAP handle opened with the `DAPOPEN_WRITE` attribute, to reload DAPL to a DAP.

An application can open a pipe, a Data Acquisition Processor, or a server PC with the `DAPOPEN_QUERY` attribute. This handle can only be used with the `DapHandleQuery` function to retrieve static information about the target the handle is associated with. Opening a handle with the `DAPOPEN_QUERY` attribute does not prevent any other application from opening the same target again for any purpose. Under DAPcell /DAPcell Local /DAPcell Basic Server, a handle opened with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute can also be used with `DapHandleQuery`.

With DAPcell/DAPcell Local Server, an application can open a pipe with the `DAPOPEN_DISKIO` attribute. This handle can be used later to initiate and terminate a direct pipe disk I/O session using either the `DapPipeDiskLog` or `DapPipeDiskFeed` function. It can also be used to query disk I/O status using the `DapHandleQuery` function. This attribute is not available in DAPcell Basic Server.

Examples

To acquire write access to the input pipe `$BININ` on the Data Acquisition Processor `DAP0` on the local machine, use the following syntax:

```
hdapBinPut = DapHandleOpen("\\\\.\\Dap0\\$BinIn",
  DAPOPEN_WRITE);
```

To acquire query access to the Data Acquisition Processor `DAP0` on the local machine, use the following syntax:

```
hDAP0 = DapHandleOpen("\\\\.\\Dap0", DAPOPEN_QUERY);
```

To acquire query access to the remote server on `PC16`, use the following syntax:

```
hPC16 = DapHandleOpen("\\\\PC16", DAPOPEN_QUERY);
```

To acquire direct disk log access to the remote pipe `$BinOut` on PC16 from a DAPcell service, use the following syntax:

```
hLog = DapHandleOpen("\\\\PC16\\$BinOut", DAPOPEN_DISKIO);
```

See Also

`DapHandleClose`, `DapHandleQuery`

DapHandleQuery

The `DapHandleQuery` function queries for information about a target the handle is associated with. There are four types of handles: a `NULL` handle, a handle to a server PC, a handle to a Data Acquisition Processor, and a handle to a communication pipe on a Data Acquisition Processor. Querying about a `NULL` handle gives information that is not specific to a particular server, a particular Data Acquisition Processor, or a particular pipe.

```
BOOL __stdcall DapHandleQuery(
   HDAP hAccel,                         // The handle to query about
   TDapHandleQuery *pHandleInfo         // Address of query structure
   );
```

Parameters

hAccel

Identifies the handle to query about.

pHandleInfo

Points to a `TDapHandleQuery` structure that passes the query key to the function and receives queried information from the function.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

An application must initialize the `TDapHandleQuery` structure correctly before calling the `DapHandleQuery` function. Use *DapStructPrepare* to prepare the structure before setting the specific fields of interest.

The *iBufferSize* field must always be set to either zero or the byte size of an application-supplied buffer. If *iBufferSize* is set to zero, the query result is returned in the structure field *QueryResult.dw* as a 32-bit binary value. No application-supplied buffer is necessary. If *iBufferSize* is non-zero, the query result is returned in a buffer. An application must supply the buffer to receive the result. In this case, *iBufferSize* must be the byte size of the buffer and *QueryResult.psz* or *QueryResult.pvoid* must point to the buffer. The result type depends on the value specified in *eResultType*. The buffer must be large enough to account for any terminating null characters required by result types such as strings.

The handle to be queried can be a handle opened with any one of the following attributes, `DAPOPEN_READ`, `DAPOPEN_WRITE`, or `DAPOPEN_QUERY`. However, opening a handle with the `DAPOPEN_READ` or the `DAPOPEN_WRITE` attribute reserves the target for exclusive use. If the handle is only used for querying, it is typically opened with the `DAPOPEN_QUERY` attribute.

To query the disk I/O status in a DAPcell/DAPcell Local service, the handle must be opened with the `DAPOPEN_DISKIO` attribute.

Query Keys: NULL Handle

With a `NULL` handle, the function supports the following query keys:

"ClientVersion"

Query key to obtain the version of the DAPIO32 client software. The query result is a 32-bit binary value. This query does not require an open handle. The `hAccel` parameter can be `NULL`.

"ServerEnumerate"

Query key to obtain a list of DAPcell servers that the client can access in a networked environment. The query result is a list of server names. Each name is a null-terminated string, with the last name terminated by two null characters. The server name is represented by the host computer name led by two backslashes. This query does not require an open handle. The *hAccel* parameter can be `NULL`.

This key can be optionally followed by a set of options, each of which is enclosed in square brackets. The key and the options belong to the same query string and must reside inside the same double quotation marks, separated only by blanks. The supported options are *[provider=xxx]*, *[domain=xxx]*, and *[transport=xxx]*. When one or more options are present, the query function will use the specified transport, if there is one, to enumerate server PCs under the specified network provider and/or domain only. This can significantly speed up the enumeration process under a large and complicated network structure by forcing the search within a restricted range. Following is an example key string with three options: *"ServerEnumerate [provider=Microsoft Windows Network] [domain=mydomain] [transport=ncacn_ip_tcp]"*.

Query Keys: Open Server Handle

With an open server handle, the function supports the following query keys:

"DapEnumerate"

Query key to obtain a list of Data Acquisition Processors available on a target server. The query result is a list of names of the Data Acquisition Processors. Each name is a null-terminated string, with the last name terminated by two null characters.

"DiskFeedEnabled"

Query key to determine whether remote disk data feeding is enabled. Returns a 32-bit binary integer result, indicating the state of disk feeding permission currently configured. The valid return values are 0, 1, and 2 for the states of "`disabled`", "`restricted`", and "`normal`" respectively.

"DiskLogEnabled"

Query key to determine whether remote disk logging is enabled. Returns a 32-bit binary integer result, indicating the state of disk logging permission currently configured. The valid return values are 0, 1, and 2 for the states of "`disabled`", "`restricted`", and "`normal`" respectively.

"IsRemote"

Query key to determine whether the server is remote or local. Returns a 32-bit binary Boolean result indicating if this server associated with the handle is remote.

"ModuleInstallEnabled"

Query key to determine whether remote clients are allowed to install modules. Returns a 32-bit binary Boolean result, indicating if the server allows remote clients to install modules.

"ServerName"

Query key to obtain the name of the server PC. The query result is a string. "`DAPcell Basic`", "`DAPcell Local`", and "`DAPcell`" are among the name strings that can be returned.

"ServerOs"
Query key to obtain the operating system under which the server PC is running. The query result is a string. For example, the returned string "`Windows 7`" indicates that the server PC is running under the Microsoft Windows 7 operating system.

"ServerOsSystemType"
Query key to return a string of either "*x86*" or "*x64*" indicating the system type of the operating system under which the server is running. This query is available on servers with version 7.00 and later.

"ServerSystemType"
Query key to return a string of "*x86*" if the server is running under Windows WOW64 environment; otherwise, an empty string. This query is available on servers with version 7.00 and later.

"ServerVersion"
Query key to obtain the version of the server software. The query result is a 32-bit binary value.

"Transports"
Query key to obtain a list of network transports that both the client and the target server support. The query result is a list of transport names. Each name is a null-terminated string, with the last name terminated by double null characters.

Query Keys: Open DAP Handle

With an open DAP handle, the function supports the following query keys:

"DapModel"
Query key to obtain the hardware model of the target Data Acquisition Processor. The query result is a string that consists of both the hardware series and hardware model information, such as DAP5200a/626.

"DapName"
Query key to obtain the full product name of the target Data Acquisition Processor. The query result is a string that consists of the hardware series, hardware model, and any additional variant information which does not affect the software characteristics of the DAP board, such as DAP5200a/626-01. The *"DapModel"* key does not return the variant portion of a DAP board's product name unless that information affects the software characteristics of the board. In the preceding example, a query with the *"DapModel"* key would return DAP5200a/626. On most DAP boards the variant portion of the product name is not present and so *"DapName"* will be identical to *"DapModel"*.

"DapOs"
Query key to obtain the type of DAPL operating system running on the target DAP board. The result is a string of either `DAPL2000` or `DAPL`.

"DapSerial"
Query key to obtain the hardware serial number of the target Data Acquisition Processor. The query result is a decimal number string.

"DaplErrorMsg"
Query key to obtain the queued DAPL error message. The query result is a string. This query requires DAPL 2000 operating system version 1.23 and later.

"DaplErrorNum"
Query key to obtain the queued DAPL error number. The query result can be either binary or a string based on the request. This query requires DAPL 2000 operating system version 1.23 and later.

"DaplErrorText nnnn"

Query key to obtain a DAPL error message text, where *nnnn* is a four-digit DAPL error number. The query result is a string. This query requires DAPL 2000 operating system version 1.23 and later.

"DaplEventLog [target=syslog] [n | start end | id=<hexid> | id=<hexid> n]"
"DaplEventLog [target=knllog] [n | start end]"

Query key that returns a list of formatted event log messages in chronological order. The list is terminated by two null characters. The query can have two options: a target option and a range option. If the target option is absent, it defaults to *[target=syslog]*. If no range option is present, the range defaults to all. If the option *[n]* is present, it returns the most recent n messages. If *[start end]* is present, it returns messages in the specified index range relative to the most recent one. If *[id=<hexid>]* is present, it returns all messages starting from the specified one-based event ID in hexadecimal. If *[id=<hexid> n]* is present, it returns n messages starting from the specified event ID. The last two options are not available if the target is knllog. If messages in the requested range are not all available, it returns whatever are available. The query fails if the user-supplied query buffer is too small for all the requested messages.

This key is only available in DAPL 3000.

"DaplInputAnalogGains"

Query key to return a list of supported analog input gains. The return gains are floating point values in ASCII string format. Each supported gain occupies a line, separated by a carriage return.

This key is available in DAPL 2000 versions higher than 2.53 and in DAPL 3000. DapHandleQuery returns false if the key is used with earlier versions of DAPL.

"DaplInputAnalogVRanges"

Query key to return a list of supported analog input voltage ranges. The returned ranges are pairs of floating point values in ASCII string format. Each supported pair occupies a line, separated by a carriage return.

This key is available in DAPL 2000 versions higher than 2.53 and in DAPL 3000. DapHandleQuery returns false if the key is used with earlier versions of DAPL or if the target DAP does not support software selectable voltage ranges.

"DaplInputChannelGroupSize <n>"

Query key to obtain the DAPL input channel group size. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

An optional integer parameter *<n>* can be present to specify a zero-based index of the entry into an input channel group size array if the target DAP supports multiple input channel group sizes. Specifying the index of 0 always retrieves the default group size, which is identical to the result of specifying no index. If the index is out of range, the query fails. Repeatedly calling this query with incrementing indices until the call fails enumerates all group sizes the target DAP supports. This option requires DAPL 2000 operating system version 2.06 and later.

"DaplInputChannelConfigurationCountMax"

Query key to obtain the DAPL maximum input channel configuration size. This number is the maximum value that can be used in a DAPL IDEFINE command. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

"DaplInputCount [target=<configuration list>|]"*

Query key to obtain the latest sample count(s) of the named input configuration(s). If the *[target=...]* option is absent, it returns the sample count of the system default configuration. The count is returned as a 64-bit unsigned integer in either binary or string format. If the query buffer is not present, it returns the lower 32 bits of the count in *QueryResult.dw*. If *[target=*]* is specified, it returns sample counts of all input configurations that are or have been active. If `[target=<configuration list>]` is specified where `<configuration list>` is a comma-separated list of input configuration names, it returns the sample counts of all the listed configurations. In both cases, the query result is a double-null-terminated multi-string in the format of `<configuration name>:<count>`.

This query requires DAPL 2000 operating system version 1.23 and later. The use of `[target=…]` option requires DAPL 3000 operating system versions 1.10 or later.

"DaplInputMasterDivideMax"

Query key to return a 32-bit integer of the largest synchronous input sampling clock divisor the board supports. A divisor divides the input sampling clock rate on the master board by its value, to provide the slave board connected to it with a clock of reduced rate at the synchronous input clock output pin on J13. On boards that do not support this capability, the query returns a value of 1.

This key is only available in DAPL 3000.

"DaplInputScanTimeMin [update=active|inactive] [igroup=n] [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL minimum input scan time in nanoseconds. If the `[update=active|inactive]` option is specified, the value returned is the minimum input scan time while the output updating is active or is inactive, depending on the selection. The default is *inactive*. An optional parameter `[igroup=n]` can be specified to request for the minimum input scan time with input channel group size equal to *n*. This is useful if the target DAP supports more than one input channel group size. If the option is not present, the default group size is assumed. If the specified group size is not valid for the target DAP, the query fails.

The query result is a number in either binary or string format based on the request. The option `[datatype=…]` selects the data type of the number. If not specified, it defaults to `double`. The option `[phase=…]` selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires the DAPL 3000 operating system.

"DaplInputScanTimeMax [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL maximum input scan time in nanoseconds. The result is a number in either binary or string format based on the request. The option `[datatype=…]` selects the data type of the number. If not specified, it defaults to *double*. The option `[phase=…]` selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires DAPL 3000 operating system.

"DaplInputScanTimeIncrement [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL input scan time increment in nanoseconds. The result is a number in either binary or string format based on the request. The option `[datatype=…]` selects the data type of the number. If not specified, it defaults to *double*. The option `[phase=…]` selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires DAPL 3000 operating system.

"DaplMemFree"

Query key to obtain the total free heap memory on a Data Acquisition processor. The query result can be either binary or a string based on the request. This query requires DAPL 2000 operating system version 1.23 and later.

"DaplMemTotal"

Query key to obtain the total heap memory on a Data Acquisition processor. The query result can be either binary or a string based on the request. This query requires DAPL 2000 operating system version 1.23 and later.

"DaplMonitorData [type=temperature] [target=cpu|monitor]"

Query key to return a 32-bit integer result in Celsius. If both *[type=temperature]* and *[target=cpu]* are present, the result is the temperature of the CPU. If both *[type=temperature]* and *[target=monitor]* are present, the result is the temperature of the monitor chip. If only *[type=temperature]* is present or no options are present, it defaults to *[type=temperature] [target=cpu]*. In all other cases, it generates a DAPL error. If no monitor is available on the target DAP, the query fails with an error of unsupported function.

This query requires DAPL 3000 operating system.

"DaplName"

Query key to return an ASCII string of the name of the DAPL operating system, such as "*DAPL3000*".

This query requires DAPL 3000 operating system.

"DaplOutputChannelConfigurationCountMax"

Query key to obtain the DAPL maximum output channel configuration size. This number is the maximum value that can be used in a DAPL ODEFINE command. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

"DaplOutputChannelGroupSize"

Query key to obtain the DAPL output channel group size. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

"DaplOutputCount [target=<configuration list>|]"*

Query key to obtain the latest update count(s) of the named output configuration(s). If the *[target=...]* option is absent, it returns the update count of the system default configuration. The count is returned as a 64-bit unsigned integer in either binary or string format. If the query buffer is not present, it returns the lower 32 bits of the count in *QueryResult.dw*. If *[target=*]* is specified, it returns update counts of all input configurations that are or have been active. If *[target=<configuration list>]* is specified where *<configuration list>* is a comma separated list of output configuration names, it returns the update counts of all the listed configurations. In both cases, the query result is a double-null-terminated multi-string in the format of *<configuration name>:<count>*.

This query requires DAPL 2000 operating system version 1.23 and later. The use of *[target=...]* option requires DAPL 3000 operating system version 1.10 or later.

"DaplOutputScanTimeMin [sample=active|inactive] [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL minimum output scan time in nanoseconds. If the *[sample=...]* option is specified, the value returned is the minimum output scan time while input sampling is active or is inactive, depending on the selection. The default is *inactive*.

The query result is a number in either binary or string format based on the request. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *double*. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires DAPL 3000 operating system.

"DaplOutputScanTimeMax [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"
Query key to obtain the DAPL maximum output scan time in nanoseconds. The result is a number in either binary or string format based on the request. The option *[datatype=...]* selects the type of the number. If not specified, it defaults to *double*. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires DAPL 3000 operating system.

"DaplOutputScanTimeIncrement [phase=cfgtime|iotime] [datatype=uint32|uint64|float|double]"
Query key to obtain the DAPL output scan time increment in nanoseconds. The result is a number in either binary or string format based on the request. The option *[datatype=...]* selects the type of the number. If not specified, it defaults to *double*. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to *cfgtime*.

This query requires DAPL 3000 operating system.

"DaplOverflowCount [target=<configuration list>|]""*
Query key to obtain the input overflow count(s) of the named input configuration(s). If the *[target=...]* option is absent, it returns the overflow count of the system default configuration. The count is returned as a 64-bit unsigned integer in either binary or string format. If the query buffer is not present, it returns the lower 32 bits of the count in *QueryResult.dw*. If *[target=*]* is specified, it returns overflow counts of all input configurations that have been overflowed. If *[target=<configuration list>]* is specified where *<configuration list>* is a comma-separated list of input configuration names, it returns the overflow counts of all the listed configurations. In both cases, the query result is a double-null-terminated multi-string in the format of *<configuration name>:<count>*.

This query requires DAPL 2000 operating system version 1.23 and later. The use of *[target=...]* option requires DAPL 3000 operating system versions 1.10 or later.

"DaplSampleResolutionAnalog"
Query key to obtain the DAPL analog input resolution in bits. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

"DaplSampleTimeIncrement [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"
Query key to obtain the DAPL input time increment in nanoseconds. The result is a 32-bit unsigned integer in either binary or string format based on the request. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware. If it is absent the option defaults to *cfgtime*. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *uint32*.

This query requires DAPL 2000 operating system version 1.30 and later. The options are only available in DAPL 3000.

"DaplSampleTimeMaxAnalog [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"
Query key to obtain DAPL maximum analog input sampling time in nanoseconds. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in *QueryResult.dw*, the lower 32 bits of the value are returned. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring

the hardware. If it is absent the option defaults to *cfgtime*. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *uint64*.

This query requires DAPL 2000 operating system version 1.30 and later. The options are only available in DAPL 3000.

"DaplSampleTimeMaxDigital [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL maximum digital input sampling time in nanoseconds. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in *QueryResult.dw*, the lower 32 bits of the value are returned. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware. If it is absent the option defaults to *cfgtime*. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *uint64*.

This query requires DAPL 2000 operating system version 1.30 and later. The options are only available in DAPL 3000.

"DaplSampleTimeMinAnalog [update=active/inactive] [igroup=n] [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL minimum analog input sampling time in nanoseconds. If the [update=...] option is specified, the value returned is the minimum analog input sampling time while output updating is active or is inactive, depending on the selection. The default is *inactive*. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in *QueryResult.dw*, the lower 32 bits of the value are returned. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware. If it is absent the option defaults to *cfgtime*. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *uint64*.

This query requires DAPL 2000 operating system version 1.30 and later. Options *[phase=...]* and *[datatype=...]* are only available in DAPL 3000.

An optional parameter *[igroup=n]* can be specified to request for the minimum analog input sampling time with input channel group size equal to *n*. This is useful if the target DAP supports more than one input channel group size. If the option is not present, the default group size is assumed. If the specified group size is not valid for the target DAP, the query fails. This option requires DAPL 2000 operating system version 2.06 and later.

"DaplSampleTimeMinDigital [update=active/inactive] [igroup=n] [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL minimum digital input sampling time in nanoseconds. If the *[update=...]* option is specified, the value returned is the minimum digital input sampling time while output updating is active or is inactive, depending on the selection. The default is *inactive*. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in *QueryResult.dw*, the lower 32 bits of the value are returned. The option *[phase=...]* selects the query purpose, *cfgtime* for configuration definition purpose and *iotime* for the purpose of configuring the hardware. If it is absent the option defaults to *cfgtime*. The option *[datatype=...]* selects the data type of the number. If not specified, it defaults to *uint64*.

This query requires DAPL 2000 operating system version 1.30 and later. Options *[phase=...]* and *[datatype=...]* are only available in DAPL 3000.

An optional parameter *[igroup=n]* can be specified to request for the minimum digital input sampling time with input channel group size equal to *n*. This is useful if the target DAP supports more than one input channel group size. If the option is not present, the default group size is assumed. If the specified group size is not valid for the target DAP, the query fails. This option requires DAPL 2000 operating system version 2.06 and later.

"DaplSupports [<tag>]"

Query key to return a 32-bit binary Boolean result indicating if `<tag>` is supported, where `<tag>` is a DAPL system command name. For example, "`DaplSupports [calibrate]`" queries DAPL for support of the `calibrate` command. For configuration commands such as `idefine`, `odefine`, and `pdefine`, a second configuration command name can follow. "`DaplSupports [idefine channels]`" is a valid key for support of the input configuration command `channels`. The `<tag>` accepts all abbreviations of a command name that DAPL accepts.

This query requires DAPL 2000 version 2.52 and later.

"DaplSymbol <nnn>"

Query key to obtain the serial numbers of currently defined DAPL symbols. All DAPL symbols including DAPL commands, built-in pipes, user-defined symbols, and custom commands have a unique non-zero serial number associated with them. This serial number is defined for the life of a symbol and can be used to access that symbol regardless of changes to any of its other attributes. Each of these serial numbers is found and returned as an element in a `DWORD` (32-bit integer) array supplied by the `TDapHandleQuery` structure. Since the number of symbols is not fixed, a single array may not be sufficient to hold all available symbols. For this reason an optional starting index *"nnn"* is specified in the key. The resulting array contains serial numbers starting with the closest one greater than *"nnn"*. When `<nnn>` is absent, zero is assumed. The query will return as many serial numbers as the provided array buffer can hold, or fewer than the array capacity if there are fewer than that many between *"nnn"*+1 and the last one available. In the latter case, a zero item immediately follows the last serial number and terminates the array. The return array is not sorted, but it is guaranteed that the last serial number is the largest in the array. To get all the serial numbers, call the query repeatedly using the last serial number in the array as the next value of *"nnn"* until the terminating zero item value is encountered.

The query's `eResultType` should be defined as `DAPIO_BINARY`, and the `QueryResult.pvoid` and `iBufferSize` fields should be set for the serial number array. A buffer of any size larger than one array entry is valid for the query. To avoid repeatedly calling the query function to get all serial numbers, a reasonably large array such as 1024 entries long can be used. This query requires DAPL 2000 operating system version 2.00 and later.

"DaplSymbol <name> [property=serial]"

Query key to obtain the serial number of a currently defined DAPL symbol given its name *"name"*. The serial number is returned as an unsigned long integer in the field `QueryResult.dw` or in the supplied buffer pointed to by the `QueryResult.pvoid` field of the `TDapHandleQuery` structure. If a buffer is supplied, the `eResultType` can be any of the supported types. If an error occurs, call `DapLastErrorTextGet` to get the cause of error. This query requires DAPL 2000 operating system version 2.00 and later.

"DaplSymbol <serial> [property=name]"

Query key to obtain the name of a currently defined DAPL symbol given its serial number *"serial"*. The name is returned as a null-terminated string in the buffer pointed to by the `QueryResult.psz` field of the `TDapHandleQuery` structure. The `eResultType` field should be `DAPIO_SZ`. A buffer must be supplied that can hold the longest possible symbol name - currently, 24 bytes including the null terminator. This query requires DAPL 2000 operating system version 2.00 and later.

"DaplSymbol <serial|name> [property=buildversion]"

Query key to return the build version of the module associated with the specified name or symbol serial number. The return value is a null-terminated string in the form of "`d.dd`" such as "`1.00`". If the build version information is not present, it returns "`not available`". If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

"DaplSymbol <serial|name> [property=capabilities]"

Query key to return the capabilities of the object associated with the specified name or symbol serial number. Only the DAPL kernel module currently supports this property. The return value is a double-null-terminated multi-string, each null-terminated string of which represents a capability. When no capabilities are available, an empty string is returned. The only capability currently supported is "*removable device*" if the target is a USB DAP and the name of the object is in the query key is "*DAPL*".

This query is only available in DAPL 3000.

"DaplSymbol <serial|name> [property=copyright]"

Query key to return the copyright string of the module associated with the specified name or symbol serial number. The return value is a null-terminated string. If the copyright information is not present, it returns "`not available`". If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

"DaplSymbol <serial|name> [property=description]"

Query key to return the description string of the module associated with the specified name or symbol serial number. The return value is a null-terminated string. If the description information is not present, it returns "`not available`". If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

"DaplSymbol <serial|name> [property=entries]"

Query key to obtain the number of entries in the object represented by the DAPL symbol, given its serial number *"serial"* or name *"name"*. The entry count is an unsigned long integer returned either in the field *QueryResult.dw* or in the supplied buffer pointed to by the *QueryResult.pvoid* field of the `TDapHandleQuery` structure. If a buffer is supplied, the *eResultType* can be any of the supported types.

Not all DAPL symbols support this query. The ones that support it are DAPL pipes and triggers. This query requires DAPL 2000 operating system version 2.03 and later.

"DaplSymbol <serial|name> [property=fileversion]"

Query key to return the file version of the module associated with the specified name or symbol serial number. The return value is a null-terminated string in the form of "*d.dd*" such as "*1.20*". If the file version information is not present, it returns "`not available`". If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

"DaplSymbol <serial|name> [property=ifversion]"

Query key to return the interface version of the module associated with the specified name or symbol serial number. The return value is a null-terminated string in the form of "*d.dd*" such as "*2.00*". If the interface version information is not present, it returns "`not available`". If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

“DaplSymbol <serial|name> [property=maxsize]”

Query key to obtain the maximum size of the DAPL object associated with the symbol given its serial number *“serial”* or name *“name”*. The *maxsize* is an unsigned long integer returned in the field *QueryResult.dw* or in the supplied buffer pointed to by the *QueryResult.pvoid* field of the TDapHandleQuery structure. If a buffer is supplied, the *eResultType* can be any of the supported types.

Not all DAPL symbols support this query. The one that supports it is a DAPL pipe. This query requires DAPL 2000 operating system version 2.00 and later.

“DaplSymbol <serial|name> [property=references]”

Query key to obtain the reference count of the DAPL symbol given its serial number *“serial”* or name *“name”*. The reference count is a long integer returned either in the field *QueryResult.dw* or in the supplied buffer pointed to by the *QueryResult.pvoid* field of the TDapHandleQuery structure. If a buffer is supplied, the *eResultType* can be any of the supported types. This query requires DAPL 2000 operating system version 2.00 and later.

“DaplSymbol <serial|name> [property=type]”

Query key to obtain the type of a currently defined DAPL symbol given its serial number *“serial”* or name *“name”*. The type is returned as a null-terminated string in the buffer pointed to by *QueryResult.psz* field of the TDapHandleQuery structure. The *eResultType* field should be DAPIO_SZ. A buffer must be supplied that can hold the type string including the null terminator. This query requires DAPL 2000 operating system version 2.00 and later.

“DaplSymbol <serial|name> [property=value]”

Query key to obtain the value of the DAPL object associated with the symbol given its serial number *“serial”* or name *“name”*. The value is returned in the buffer pointed to by the *QueryResult.pvoid* field of the TDapHandleQuery structure. The *eResultType* field can be either DAPIO_BINARY or DAPIO_SZ or DAPIO_VARIANT. DAPIO_VARIANT or DAPIO_SZ is recommended unless the data type of the scalar result is known in advance.

Not all DAPL symbols support this query. The ones that support it are DAPL variables, constants, and strings. This query requires DAPL 2000 operating system version 2.00 and later.

“DaplSymbol <serial|name> [property=version]”

Query key to return the version string of the module associated with the specified name or symbol serial number. The return value is a null-terminated string in the form of “*interface: x.xx, build: y.yy*” such as “*interface: 2.00, build: 1.00*”. If the version information is not present, it returns “*not available*”. If the name or serial number is not associated with a module, the query fails with an error message.

This query is only available in DAPL 3000.

“DaplSymbol <serial|name> [property=width]”

Query key to obtain the width of the DAPL object associated with the symbol given its serial number *“serial”* or name *“name”*. The width is a long integer returned either in the field *QueryResult.dw* or in the supplied buffer pointed to by the *QueryResult.pvoid* field of the TDapHandleQuery structure. If a buffer is supplied, the *eResultType* can be any of the supported types.

Not all DAPL symbols support this query. The ones that support it are DAPL pipes, variables, constants, and vectors. This query requires DAPL 2000 operating system version 2.00 and later.

“DaplUnderflowCount [target=<configuration list>|]”*

Query key to obtain the underflow count(s) of the named output configuration(s). If the *[target=...]* option is absent, it returns the underflow count of the system default configuration. The count is returned as a 64-bit unsigned integer in either binary or string format. If the query buffer is not present, it returns the lower 32 bits of the count in `QueryResult.dw`. If `[target=*]` is specified, it returns the underflow counts of all output configurations that have stopped prematurely. If `[target=<configuration list>]` is specified where `<configuration list>` is a comma-separated list of output configuration names, it returns the underflow counts of all the listed configurations. In both cases, the query result is a double-null-terminated multi-string in the format of `<configuration name>:<count>`.

This query requires DAPL 2000 operating system version 1.23 and later. The use of `[target=…]` option requires DAPL 3000 operating system versions 1.10 or later.

"`DaplUpdateResolutionAnalog`"
Query key to obtain the DAPL analog output resolution in bits. The result is a 32-bit unsigned integer in either binary or string format based on the request. This query requires DAPL 2000 operating system version 1.30 and later.

"`DaplUpdateTimeIncrement [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]`"
Query key to obtain the DAPL output time increment in nanoseconds. The result is a 32-bit unsigned integer in either binary or string format based on the request. The option `[phase=…]` selects the query purpose, `cfgtime` for configuration definition purpose and `iotime` for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to `cfgtime`. The option `[datatype=…]` selects the type of the number. If not specified, it defaults to `uint32`.

This query requires DAPL 2000 operating system version 1.30 and later. The use of options requires DAPL 3000.

"`DaplUpdateTimeMaxAnalog [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]`"
Query key to obtain the DAPL maximum analog output updating time in nanoseconds. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in `QueryResult.dw`, the lower 32 bits of the value are returned. The option `[phase=…]` selects the query purpose, `cfgtime` for configuration definition purpose and `iotime` for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to `cfgtime`. The option `[datatype=…]` selects the type of the number. If not specified, it defaults to `uint64`.

This query requires DAPL 2000 operating system version 1.30 and later. The use of options requires DAPL 3000.

"`DaplUpdateTimeMaxDigital`
`[phase=cfgtime|iotime][datatype=uint32|uint64|float|double]`"
Query key to obtain the DAPL maximum digital output updating time in nanoseconds. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in `QueryResult.dw`, the lower 32 bits of the value are returned. The option `[phase=…]` selects the query purpose, `cfgtime` for configuration definition purpose and `iotime` for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to `cfgtime`. The option `[datatype=…]` selects the type of the number. If not specified, it defaults to `uint64`.

This query requires DAPL 2000 operating system version 1.30 and later. The use of options requires DAPL 3000.

"`DaplUpdateTimeMinAnalog [sample=active/inactive]`
`[phase=cfgtime|iotime][datatype=uint32|uint64|float|double]`"
Query key to obtain the DAPL minimum analog output updating time in nanoseconds. If the `[sample=…]` option is specified, the value returned is the minimum analog output updating time while input sampling is active or is inactive, depending on the selection. The default is `inactive`. The result is a 64-bit unsigned

integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in `QueryResult.dw`, the lower 32 bits of the value are returned. The option `[phase=…]` selects the query purpose, `cfgtime` for configuration definition purpose and `iotime` for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to `cfgtime`. The option `[datatype=…]` selects the type of the number. If not specified, it defaults to `uint64`.

This query requires DAPL 2000 operating system version 1.30 and later. The use of options `[phase=…]` and `[datatype=…]` requires DAPL 3000.

"DaplUpdateTimeMinDigital [update=active/inactive] [phase=cfgtime|iotime][datatype=uint32|uint64|float|double]"

Query key to obtain the DAPL minimum digital output updating time in nanoseconds. If the `[sample=…]` option is specified, the value returned is the minimum digital output updating time while input sampling is active or is inactive, depending on the selection. The default is `inactive`. The result is a 64-bit unsigned integer in either binary or string format based on the request. If the return value is expected as a 32-bit binary value in `QueryResult.dw`, the lower 32 bits of the value are returned. The option `[phase=…]` selects the query purpose, `cfgtime` for configuration definition purpose and `iotime` for the purpose of configuring the hardware I/O clock intervals. If not specified, it defaults to `cfgtime`. The option `[datatype=…]` selects the type of the number. If not specified, it defaults to `uint64`.

This query requires DAPL 2000 operating system version 1.30 and later. The use of options `[phase=…]` and `[datatype=…]` requires DAPL 3000.

"DaplVersion"

Query key to obtain the version of the DAPL operating system. The query result can be either a binary value of a string. If the result is binary, the major version is the hundreds digit of the value, the minor version is the tens digit and last digit represents the micro version. If the result is a string, it takes the format of "`<Major>.<Minor><Micro>`". This query requires DAPL 2000 operating system version 1.23 and later.

"DaplWarnMsg"

Query key to obtain the queued DAPL warning message. The query result is a string. This query requires DAPL 2000 operating system version 2.03 and later.

"DaplWarnNum"

Query key to obtain the queued DAPL system warning number. The query result can be either binary or a string based on the request. This query requires DAPL 2000 operating system version 2.03 and later.

"PipeEnumerate"

Query key to obtain a list of communication pipes configured for a Data Acquisition Processor. The query result is a list of pipe names. Each name is a null-terminated string, with the last name string terminated by two null characters.

Query Keys: Open Pipe Handle

With an open pipe handle, the function supports the following query keys:

"PipeAttribute"

Query key to obtain the attributes of the target communication pipe. The query result is a string that resembles the attribute list used in a call to the `DapComPipeCreate` function. This query requires an open pipe handle.

"PipeDirection"

Query key to obtain the direction of a pipe. The query result is a string of either "`input from DAP`" or "`output to DAP`". This query requires an open pipe handle.

“PipeMaxSize”

Query key to obtain the maximum buffer size of a pipe. The query result is a 32-bit binary value. This query requires an open handle to the target communication pipe.

“PipeType”

This key is a synonym to the key *“PipeDirection”*. Use *“PipeDirection”* in new applications.

“PipeWidth”

Query key to obtain the width of a pipe. The query result is a 32-bit binary value. This query requires an open handle to the target communication pipe.

Query Keys: Open Pipe Handle with the DAPOPEN_DISKIO Attribute

With an open pipe handle with the DAPOPEN_DISKIO attribute, the function supports the following query keys:

“DapDiskIoCount [file=n]” “DapDiskIoCount [dapio]”

Query key to obtain the number of bytes transferred between a data file and a DAP pipe (or vice versa) during a disk I/O session.

If the *iBufferSize* field is non-zero, the return value is a 64-bit integer stored in the application-supplied buffer pointed to by *QueryResult.pvoid*. The *eResultType* field determines the result format. The buffer must be large enough to hold the result; otherwise the query fails. If the *iBufferSize* field is zero, the lower 32 bits of the result is stored in *QueryResult.dw*.

When used in conjunction with a DapPipeDiskLog command, the option *[file=n]* returns the number of bytes read from or written to a file; *n=0* returns the primary log file count, *n=1* returns the first mirror file count, and so on. If no option is given, *“[file=0]”* is assumed. If this option is specified for a mirror file that does not exist, an error is generated.

When the *“[dapio]”* option is given, this number will be the total number of bytes transferred to or from the pipe buffer.

At any moment, data transferred to or from the pipe buffer is not necessarily equal to the data transferred from or to the disk file until after the disk I/O is completed.

“DapDiskIoStatus [file=n]” “DapDiskIoStatus [dapio]”

Query key to obtain the status of a DapPipeDiskLog or DapPipeDiskFeed command. The result can be either an integer status code or string. If the *iBufferSize* field of the TDapHandleQuery structure is set to zero, the resulting status code is stored in the structure’s *QueryResult.dw* field. If *iBufferSize* is set to any other value, the status is stored in the application-provided buffer pointed to by the *QueryResult.pvoid* or *QueryResult.psz* field. When the *eResultType* field is set to DAPIO_BINARY the resulting status code is a 32-bit integer value in the buffer. Any other *eResultType* results in the status string stored in the buffer pointed to by *QueryResult.psz*.

The possibilities for the status code and its associated string are:

• ddios_Active - the disk I/O session is active and data are being transferred. Returns the string “The disk I/O is active.”

• ddios_Completed - the requested number of items have been successfully transferred with no errors. Returns the string “The disk I/O is completed.”

• ddios_FileError - a file has been closed because of an error before the disk I/O is terminated. Returns the string “Disk I/O file error:” followed by the details of the last file closure. To determine which file has been closed, each file must be polled separately.

• `ddios_Aborted` - the disk I/O session has been terminated due to a fatal error. Returns the string "`The disk I/O has been aborted - `" followed by the details of the session termination.

When used in conjunction with a `DapPipeDiskLog` command, the option *"`[file=n]`"* retrieves the status of an individual file; *`n=0`* specifies the primary log file, *`n=1`* specifies the first mirror file, and so on. When this option is specified for a mirror file that does not exist, an error is generated. When no option is used, *"`[file=0]`"* is assumed

The option *"`[dapio]`"* retrieves the status of the I/O to or from the pipe buffer.

Query Keys: Any Open Handle

With any open handle, the function supports the following query keys:

"`BindTransport`"

Query key to obtain the description of the network transport the handle uses to communicate with the server. The query result is a string. This key requires DAPIO32 interface version 2.12 or later.

"`HandleName`"

Query key to obtain the name of the open handle. The query result is a string that resembles the UNC name used in opening the handle.

See Also

`TDapHandleQuery`, `DapComPipeCreate`, `DapHandleQueryInt32`, `DapHandleQueryInt64`

DapHandleQueryInt32

The `DapHandleQueryInt32` function provides a more convenient interface than `DapHandleQuery` for queries that yield 32-bit integer results.

```
int __stdcall DapHandleQueryInt32(
    HDAP hAccel,                        // Handle to query about
    const char * pszKey,                // Pointer to a query key string
    int *pi32Result                     // Pointer to a 32-bit storage result
    );
```

Parameters

hAccel
Identifies the handle to query about.

pszKey
Points to a null-terminated query key string.

pi32Result
Points to a 32-bit storage to receive data.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapHandleQueryInt32` is implemented as a wrapper to `DapHandleQuery` for convenience. The function applies to all queries that yield 32-bit integer results. See `DapHandleQuery` for more information.

See Also

`DapHandleQuery`, `DapHandleQueryInt64`

DapHandleQueryInt64

The `DapHandleQueryInt64` function provides a more convenient interface than `DapHandleQuery` for queries that yield 64-bit integer results.

```
int __stdcall DapHandleQueryInt64(
    HDAP hAccel,                    // Handle to query about
    const char * pszKey,            // Pointer to a query key string
    TDapIoInt64 *pi64Result         // Pointer to a 64-bit storage result
    );
```

Parameters

hAccel
Identifies the handle to query about.

pszKey
Points to a null-terminated query key string.

pi64Result
Points to a 64-bit storage to receive data.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapHandleQueryInt64` is implemented as a wrapper to `DapHandleQuery` for convenience. The function applies to all queries that yield 64-bit integer results. See `DapHandleQuery` for more information.

See Also

`DapHandleQuery`, `DapHandleQueryInt32`, `TDapIoInt64`

DapInputAvail

The `DapInputAvail` function gets the number of data bytes available for reading in the target pipe.

```
int __stdcall DapInputAvail(
    HDAP hAccel                          // Open handle of the target pipe
    );
```

Parameters

hAccel

Specifies the open handle to the target pipe. The target pipe must be an output pipe from the Data Acquisition Processor.

Return Values

If the function succeeds, the return value is the number of data bytes available in the target pipe.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Unlike its counterpart in DAPL, this function always returns a count in bytes regardless of the actual width of pipe elements.

The number returned by this function is the number of data bytes already buffered by the server. An application is safe to read that number of data bytes from the pipe without ever being blocked.

For efficient data transfer, it is usually best to use `DapBufferGetEx`, either with minimum bytes to read of zero or a time-out, and avoid use of `DapInputAvail`. `DapBufferGetEx` returns the actual number of bytes read and so lets the application know what data have been transferred. The reason to avoid `DapInputAvail` is that, when it is used, each block transfer usually requires two calls to the server, one to determine what data are available and one to read the data. Using `DapBufferGetEx` alone only requires one call to the server for basically the same operation.

When an application uses the network feature of DAPcell Server, it is particularly important to reduce the number of calls to the server, since calls to the server can be dispatched over the network. The overhead of dispatching a call over the network can be expensive.

See Also

`DapBufferGetEx`

DapInputFlush

The `DapInputFlush` function flushes an output pipe from the Data Acquisition Processor by discarding its data from the PC memory. It returns the number of bytes discarded.

```
int __stdcall DapInputFlush(
    HDAP hAccel                             // Open handle of the target pipe
    );
```

Parameters

hAccel

Handle to the target pipe. The target pipe must be an output pipe from the Data Acquisition Processor.

Return Values

If the function succeeds, the return value is the number of bytes flushed.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

There is a built-in 20-second time-out; if it is unable to flush all data from *hAccel* after the time-out, it returns failure (-1). If no data are found after a delay of 100 ms, `DapInputFlush` returns successfully with the number of bytes flushed so far.

For more control over the flush operation, use `DapInputFlushEx`.

Example

The following code flushes the $binout on the PC.

```
HDAP hBinout = DapHandleOpen("\\\\.\\dap0\\$binout, DAPOPEN_READ);
Int BytesFlushed;
…
BytesFlushed = DapInputFlush(hBinout);
…
```

See Also

`DapInputFlushEx`, `DapOutputEmpty`

DapInputFlushEx

The `DapInputFlushEx` function flushes an output pipe from the Data Acquisition Processor by discarding its data from the PC memory. It always returns control to the caller, even when it is unable to flush all data from the target pipe. `DapInputFlushEx` is an extended version of the `DapInputFlush` service.

```
BOOL __stdcall DapInputFlushEx(
    HDAP hAccel,                        // Open handle to the target pipe
    unsigned long dwTimeOut,            // Maximum time to flush
    unsigned long dwTimeWait,           // Maximum time to wait
    unsigned long *pdwFlushed           // Address of the number flushed
    );
```

Parameters

hAccel
Specifies the handle to the target pipe. The handle must have read access. The target pipe must be an output pipe from the Data Acquisition Processor.

dwTimeOut
Specifies the maximum amount of time in milliseconds within which the flushing operation should complete.

dwTimeWait
Specifies the minimum amount of time in milliseconds for which `DapInputFlushEx` should guarantee that the target pipe remains empty to claim success.

pdwFlushed
Points to a 32-bit variable that receives the number of data bytes actually flushed. If the caller does not need the number of bytes flushed, set this parameter to `NULL`.

Return Values

If the function succeeds, the return value is TRUE. The variable that *pdwFlushed* points to contains the total number of data bytes flushed.

If the function fails, the return value is FALSE. The variable that *pdwFlushed* points to contains the actual number of data bytes flushed. Call `DapLastErrorTextGet` to retrieve additional information about the error.

If *pdwFlushed* is `NULL`, no count of flushed data are returned.

See Also

`DapInputFlush`, `DapOutputEmpty`

DapInt16Get

The `DapInt16Get` function reads a single 16-bit integer from a DAP com-pipe.

```
BOOL __stdcall DapInt16Get(
    HDAP hAccel,                    // Open handle to the target pipe
    short *pi                       // Location to receive integer
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

pi

Pointer to location to receive integer.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapInt16Put

The `DapInt16Put` function writes a single 16-bit integer to a DAP com-pipe.

```
BOOL __stdcall DapInt16Put(
    HDAP hAccel,                    // Open handle to the target pipe
    short i                         // 16-bit integer
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

i

Integer to write to DAP com-pipe.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapInt32Get

The `DapInt32Get` function reads a single 32-bit integer from a DAP com-pipe.

```
BOOL __stdcall DapInt32Get(
    HDAP hAccel,                    // Open handle to the target pipe
    long *pl                        // Location to receive integer
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

pl

Pointer to location to receive integer.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapInt32Put

The `DapInt32Put` function writes a single 32-bit integer to a DAP com-pipe.

```
BOOL __stdcall DapInt32Put(
    HDAP hAccel,                         // Open handle to the target pipe
    long l                               // 32-bit integer
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

l

Integer to write to DAP com-pipe.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

DapLastErrorTextGet

The `DapLastErrorTextGet` function retrieves the message text of the calling thread's last error, including custom DAPIO32-specific errors.

```
char * DapLastErrorTextGet(
    char *pszError,                 // Points to application-supplied buffer
    int iLength                     // Size of the buffer
    );
```

Parameters

pszError
: Points to an application-supplied buffer that receives the error message.

iLength
: Specifies the size of the application supplied buffer.

Return Values

If the function succeeds, the return value is the pointer to a null-terminated message text string.

If the function fails, the return value is the pointer to a null string.

Remarks

DAPIO32-specific errors are defined as custom errors in the file DAPIO32.DLL. `DapLastErrorTextGet` checks the error code and retrieves the error message text from DAPIO32.DLL, if it is a custom DAPIO32-specific error.

An application can also use Win32 API's `GetLastError` and `FormatMessage` to retrieve the last error text message. The application needs to check the `Custom` bit of the error code returned by `GetLastError` and tell `FormatMessage` to retrieve messages from DAPIO32.DLL if that bit is set.

DapLineGet

The `DapLineGet` function gets a string of characters from the DAP com-pipe with user-specified time-out.

```
int __stdcall DapLineGet(
    HDAP hAccel,                          // Handle to target pipe
    int iLength,                          // Length of pszBuffer
    char *pszBuffer,                      // Pointer to buffer
    unsigned long dwTimeWait              // Maximum time to wait
    );
```

Parameters

hAccel
Handle to DAP com-pipe from which to read a null-terminated string.

iLength
Length of *pszBuffer* including space for the null character.

pszBuffer
Pointer to buffer to store null-terminated string.

dwTimeWait
Time, in milliseconds, which the com-pipe must remain empty to return without a complete string. May be zero.

Return Values

If the function succeeds, returns the number of characters read from the Data Acquisition Processor excluding line-feed characters. The result includes the terminating carriage-return if one is encountered; the carriage-return is not placed in *pszBuffer*. Zero indicates that *dwTimeWait* milliseconds elapsed without reading any characters.

If the function fails, the return value is -1. Either the handle is invalid or *pszBuffer* is `NULL`. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapLineGet` reads characters from the com-pipe *hAccel* and places them in *pszBuffer* up to the first carriage-return character.

If no characters are available from the com-pipe for more than *dwTimeWait* milliseconds, returns with whatever characters have been read so far in *pszBuffer* and a count of the number of characters read.

If more than *iLength - 1* characters are encountered before a carriage-return is found returns *iLength - 1* characters. It is possible to determine whether a carriage-return character was encountered by comparing the returned count to the length of the string returned in *pszBuffer*.

Line-feed characters are ignored and are never placed in the string.

See Also

`DapCharGet`, `DapStringGet`

DapLinePut

The `DapLinePut` function writes a string of characters to a Data Acquisition Processor com-pipe and terminates the string with a carriage-return.

```
int __stdcall DapLinePut(
    HDAP hAccel,                    // Open handle to the target pipe
    const char *psz                 // Character string
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

psz

Null-terminated string to write to DAP com-pipe. *psz* may be `NULL`. In this case just a carriage-return is sent to the DAP com-pipe.

Return Values

If the function succeeds, the return value is the number of characters actually written to the DAP com-pipe, including the terminating carriage-return.

If the function fails, the return value is 0 (zero). Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Before appending the carriage-return, `DapLinePut` strips all line-feed and carriage-return characters from the right side of the string.

`DapLinePut` only checks the end of the string for carriage-return/line-feed characters.

To send a string to the DAP com-pipe without modification, use `DapStringPut`.

See Also

`DapStringPut`

DapModuleInstall

The `DapModuleInstall` function installs a module to the DAP board(s) associated with the handle.

```
BOOL __stdcall DapModuleInstall(
    HDAP hAccel,                        // install handle
    const char *pszModPath,             // module file name
    unsigned long bmFlags,              // bit flags of installation behavior
    void *pDapList                      // reserved
    );
```

Parameters

hAccel

Identifies the DAP board(s) on which to install the module. If this is a DAP handle, it installs the module to the target DAP board. If this is a server handle, installs the module to all DAP boards on the server.

pszModPath

File name of module to install.

bmFlags

Bit flags telling the service how to proceed with the installation. The following flags are available.

Values	Description
`dmf_NoCopy`	Instruct the service not to copy the file to the default module directory created by the SETUP program. The default is to copy it. If the handle is a remote handle, the service ignores this flag and always copies the file.
`dmf_NoReplace`	Instruct the service not to replace an existing installed module. The default is to replace it.
`dmf_NoLoad`	Instruct the service not to load the module to the target DAP board(s) after installation. The default is to load.
`dmf_ForceRegister`	Instruct the service to force installation of the module. In this case, the service installs the module even if some of the modules it depends on are not installed. The default is not to force the installation.
`dmf_ForceLoad`	Instruct the service to force the loading of the module to the target DAP board(s). In this case, the operation may be destructive. The service may reset or even reload DAPL in order to complete the request. The default is not to force loading.
`dmf_OsDAPL2000`	Instruct the service to carry out the operation only if the target is running DAPL 2000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.

`dfm_OsDAPL3000`	Instruct the service to carry out the operation only if the target is running DAPL 3000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.

`pDapList`
Reserved. Must be `NULL`.

Return Values

Returns true if the installation is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This operation is persistent. Once a module is installed, it stays until the module is uninstalled.

The service loads the module to the DAP board(s) unless `dmf_NoLoad` is specified. If installation fails, the loading will not proceed.

When `dmf_ForceRegister` is specified, the installation may result in an inconsistent configuration because it ignores possible absence of the dependencies the module requires. When `dmf_ForceLoad` is specified, the operation may be destructive, in which case data on the target DAP board(s) are lost.

See Also

`DapModuleLoad`, `DapModuleUninstall`, `DapModuleUnload`, `DapReinitialize`

DapModuleLoad

The `DapModuleLoad` function loads a module to the DAP board(s) associated with the handle.

```
BOOL __stdcall DapModuleLoad(
    HDAP hAccel,                          // load handle
    const char *pszModPath,               // filename of module
    unsigned long bmFlags,                // bit flags of load behavior
    void *pDapList                        // reserved
    );
```

Parameters

hAccel

Identifies the DAP board(s) to which to load the module. If this is a DAP handle, loads the module to the target DAP board only. If this is a server handle, loads the module to all DAP boards on the server.

pszModPath

Filename of module to load.

bmFlags

Bit flags telling the service how to proceed with the loading. The following flags are available.

Values	Description
`dmf_NoReplace`	Instruct the service not to replace an existing loaded module. The default is to replace it.
`dmf_ForceLoad`	Instruct the service to force the loading of the module to the target DAP board(s). In this case, the operation may be destructive. The service may reset or even re-load DAPL in order to complete the request. The default is not to force loading.
`dmf_OsDAPL2000`	Instruct the service to carry out the operation only if the target is running DAPL 2000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.
`dfm_OsDAPL3000`	Instruct the service to carry out the operation only if the target is running DAPL 3000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.

pDapList

Reserved. Must be `NULL`.

Return Values

Returns true if the loading is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This function loads the specified module without installing it into the system. Therefore, the effect is not persistent. Once the target DAP board(s) are reinitialized, it is necessary to load the module again if it is not installed.

This function does not try to resolve module dependencies. If module dependencies exist, the caller is responsible to load each dependency module first in the right order before loading the target module. A module cannot be loaded until all its dependency modules are loaded.

When `dmf_ForceLoad` is specified, the operation may be destructive, in which case data on the target DAP board(s) are lost.

See Also

`DapModuleInstall`, `DapModuleUninstall`, `DapModuleUnload`, `DapReinitialize`

DapModuleUninstall

The `DapModuleUninstall` function uninstalls a module from the DAP board(s) associated with the handle.

```
BOOL __stdcall DapModuleUninstall(
    HDAP hAccel,                      // uninstall handle
    const char *pszModName,           // module name
    unsigned long bmFlags,            // bit flags of uninstallation behavior
    void *pDapList                    // reserved
    );
```

Parameters

hAccel

Identifies the DAP board(s) from which to uninstall the module. If this is a DAP handle, uninstalls the module from the target DAP board only. If this is a server handle, uninstalls the module from all DAP boards on the server.

pszModName

Name of module to uninstall.

bmFlags

Bit flags telling the service how to proceed with the uninstall operation. The following flags are available.

Values	**Description**
`dmf_NoLoad`	Instruct the service not to unload the module from the target DAP board(s) after uninstall. The default is to unload.
`dmf_ForceRegister`	Instruct the service to force uninstall of the module even if modules that depend on it are installed. The default is not to force uninstall.
`dmf_ForceLoad`	Instruct the service to force unloading the module from the target DAP board(s) even if it is being used in DAPL. The default is not to force unloading.
`dmf_RemoveDependents`	Instruct the service to remove dependents of the module during forced uninstall and unload. The default is not to remove dependents.
`dmf_OsDAPL2000`	Instruct the service to carry out the operation only if the target is running DAPL 2000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.
`dmf_OsDAPL3000`	Instruct the service to carry out the operation only if the target is running DAPL 3000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.

pDapList
Reserved. Must be `NULL`.

Return Values

Returns true if the uninstall is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This operation is persistent. Once a module is uninstalled, it is gone until the module is reinstalled. When `dmf_ForceRegister` is specified, the operation may result in an inconsistent configuration because the service ignores the possible presence of installed modules that depend on it.

The service unloads the module from the DAP boards after the uninstall unless `dmf_NoLoad` is specified. If uninstall fails, the unloading will not proceed.

When `dmf_ForceLoad` is specified, the operation may be partially destructive to the target DAP board(s) in order to complete the request. The result of a partially destructive operation is equivalent to issuing a “`RESET`” DAPL command to those DAP boards.

See Also

`DapModuleInstall`, `DapModuleLoad`, `DapModuleUnload`, `DapReinitialize`

DapModuleUnload

The `DapModuleUnload` function removes the module from the DAP board(s) associated with the handle.

```
BOOL __stdcall DapModuleUnload(
    HDAP hAccel,                      // unload handle
    const char *pszModName,           // module name
    unsigned long bmFlags,            // bit flags of unloading behavior
    void *pDapList                    // reserved
    );
```

Parameters

hAccel

Identifies the DAP board(s) from which to unload the module. If this is a DAP handle, unloads the module from the target DAP board only. If this is a server handle, unloads the module from all DAP boards on the server.

pszModName

Name of the module to unload.

bmFlags

Bit flags telling the service how to proceed with the unloading. The following flags are available.

Values	Description
`Dmf_ForceLoad`	Instruct the service to force unloading the module from the target DAP board(s) even if it is being used. The default is not to force unloading.
`Dmf_RemoveDependents`	Instruct the service to remove dependents of the module during unload. The default is not to remove dependents.
`dmf_OsDAPL2000`	Instruct the service to carry out the operation only if the target is running DAPL 2000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.
`dmf_OsDAPL3000`	Instruct the service to carry out the operation only if the target is running DAPL 3000. If neither `dmf_OsDAPL2000` nor `dfm_OsDAPL3000` is present, both are assumed. This flag is supported in servers with version 6.00 and later.

pDapList

Reserved. Must be `NULL`.

Return Values

Returns true if the unloading is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

When `dmf_ForceLoad` is specified, the operation may be partially destructive to the target DAP board(s) in order to complete the request. The result of a partially destructive operation is equivalent to issuing a “`RESET`” DAPL command to those DAPs

See Also

`DapModuleInstall`, `DapModuleLoad`, `DapModuleUninstall`, `DapReinitialize`

DapOutputEmpty

The `DapOutputEmpty` function wipes out all data from both sides of an input pipe to the Data Acquisition Processor.

```
int __stdcall DapOutputEmpty(
    HDAP hAccel                          // Open handle of the target pipe
    );
```

Parameters

hAccel

Specifies the open handle to the target pipe. The target pipe must be an input pipe to the Data Acquisition Processor.

Return Values

If the function succeeds, the return value is TRUE. The target pipe is completely emptied.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

`DapOutputEmpty` empties an input pipe to the Data Acquisition Processor. When it returns, the pipe is completely empty on both sides, on the PC and on the DAP.

Example

The following code wipes out data from the $binin pipe on both sides.

```
HDAP hBinin = DapHandleOpen("\\\\.\\dap0\\$binin, DAPOPEN_WRITE);
…
DapOutputEmpty(hBinin);
…
```

See Also

`DapInputFlush`, `DapInputFlushEx`

DapOutputSpace

The `DapOutputSpace` function gets the number of byte spaces available in the target pipe for writing.

```
int __stdcall DapOutputSpace(
    HDAP hAccel                              // Open handle of the target pipe
    );
```

Parameters

hAccel

Specifies the open handle to the target pipe. The target pipe must be an input pipe to the Data Acquisition Processor.

Return Values

If the function succeeds, the return value is the number of byte spaces available in the target pipe.

If the function fails, the return value is -1. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Unlike its counterpart in DAPL, this function always returns a count in bytes regardless of the actual width of pipe elements.

An application is always safe to write that number of data bytes to the pipe without ever being blocked.

DapPipeDiskFeed

The `DapPipeDiskFeed` function initiates a background disk feed session that reads data from an existing data file and places it into a DAP com-pipe.

```
BOOL __stdcall DapPipeDiskFeed(
    HDAP hFeedHandle,                    // Handle to control feed operation
    TDapPipeDiskFeed *pFeedInfo,         // Pointer to TDapPipeDiskFeed
    TDapBufferPutEx *pPutInfo            // Pointer to TDapBufferPutEx
    );
```

Parameters

hFeedHandle

A disk I/O handle previously obtained using `DapHandleOpen` with the `DAPOPEN_DISKIO` attribute. Closing *hFeedHandle* using `DapHandleClose` terminates the session.

pFeedInfo

A pointer to the `TDapPipeDiskFeed` structure that contains the disk I/O information.

pPutInfo

A pointer to a `TDapBufferPutEx` structure that contains optional pipe I/O information. If this pointer is `NULL`, a default set of attributes is assumed.

Return Values

If the session creation was successful, the return value is TRUE.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

The `DapPipeDiskFeed` function initiates a disk feed session. The disk feed session continues until the number of bytes specified in the `TDapPipeDiskFeed` *i64MaxCount* has been read, or the end of the file is reached, or the handle passed to the command is closed using `DapHandleClose`.

If the optional *pPutInfo* pointer is `NULL`, the following attributes are assumed:

```
iBytesMultiple = 1, dwTimeWait = 1000, dwTimeOut =0
(no time-out) and iBytesMultiple = 1
```

The value of *iBytesPut* in the `TDapBufferPutEx` structure is ignored. The actual value used depends on the availability of data read from the data file.

The `dpdf_FlushBefore` flag in the `TDapPipeDiskFeed` structure is ignored if the target board is a non-PCI Data Acquisition Processor.

Use *hFeedHandle* with `DapHandleQuery` to determine the ongoing session status. See `DapHandleQuery` for more information.

Some of the disk feed parameters depend on the value of the `TDapBufferPutEx` *iBytesMultiple*:

- The `TDapPipeDiskFeed` *i64MaxCount* value must be a multiple of the `TDapBufferPutEx` *iBytesMultiple* value or an error will be generated.

- If the data file's size is not a multiple of the `TDapBufferPutEx` *iBytesMultiple* value, only the data up to the last *iBytesMultiple* boundary will be fed. For example, if *iBytesMultiple* is 5 and the file size is 37, only 35 bytes will be fed to the target pipe.

- If the `TDapPipeDiskFeed` `dpdf_ContinuousFeed` flag is set, and the data file's size is not a multiple of the `TDapBufferPutEx` *iBytesMultiple* value, feeding will continue by wrapping around to the beginning of the file once the end-of-file is reached. This may yield unexpected result as the end-of-file may break the data grouping integrity typically enforced by specifying the *iBytesMultiple* value.

Security

When `dpdf_ServerSide` is not set in the `TDapPipeDiskFeed` structure, no file access control is enforced. The file name specified in *pszFileName* of the structure is either an absolute file path on the local machine or a partial path relative to the local directory where the application is launched.

When `dpdf_ServerSide` is set, disk feeding may access a file on a remote machine. The "Disk I/O" tab in the DAP control panel program provides security measures for controlling file access to the server machine.

The "Disk I/O" tab has two control entries: Permission and Default Path. They can be selected or specified to provide the desired control.

The Permission entry provides three selectable file access permission levels.

- Not allowed - No access is allowed at all on the server machine.
- Restricted - Access is restricted only to the path(s) specified in the Default Path entry and their subdirectories.
- Normal - Access is allowed anywhere reachable by the DAPcell/DAPcell Local server.

The Default Path entry specifies one or more directories that data can be read from. If *pszFileName* is a partial file name and can be found in more than one of the paths specified in Default Path, the first occurrence is used.

See the help file of the DAP control panel program for more information.

Example

In the following example, data are read from the local file `c:\data\wave1.dat` and sent to `DAP2` on the remote PC with the name `TESTPC` through the DAP board's `$BinIn` pipe. The file is repeatedly fed to the DAP board to provide a continuous stream of data. The default `TDapBufferPutEx` values are used.

```
TDapPipeDiskFeed FeedInfo;

/* Initialize the FeedInfo structure */
DapStructPrepare(FeedInfo);
FeedInfo.bmFlags =dpdl_ContinuousFeed;
FeedInfo.pszFileName = “c:\\data\\wave1.dat”;

/* Open a disk I/O handle */
hFeed = DapHandleOpen( “\\\\TESTPC\\Dap2\\$BinIn”,
DAPOPEN_DISKIO);
/* Start a logging session */
if ( hFeed && DapPipeDiskFeed(hFeed, &FeedInfo, NULL) )
{
    /* Read the data into the Dap */
}
else
{

/* handle error */
}
/* Terminate the logging session */
if (hFeed) DapHandleClose(hFeed);
```

Version

This service is only available in DAPcell Server and DAPcell Local Server version 4.00 or later. It is not available in DAPcell Basic Server.

See Also

`TDapPipeDiskFeed`, `TDapBufferPutEx`, `DapHandleQuery`, `DapPipeDiskLog`

DapPipeDiskLog

The `DapPipeDiskLog` function initiates a disk logging session between a DAP com-pipe and a disk file.

```
BOOL __stdcall DapPipeDiskLog(
    HDAP hLog,                            // Disk I/O handle
    TDapPipeDiskLog *pLogInfo,            // Pointer to TDapPipeDiskLog
    TDapBufferGetEx *pGetInfo             // Pointer to TDapBufferGetEx
    );
```

Parameters

hLog

Handle previously obtained using `DapHandleOpen` with the `DAPOPEN_DISKIO` attribute. Closing *hLog* with `DapHandleClose` terminates the logging session.

pLogInfo

Pointer to a `TDapPipeDiskLog` structure containing disk I/O information.

pGetInfo

Pointer to a `TDapBufferGetEx` structure containing optional pipe I/O information. If this pointer is `NULL`, a default set of attributes is assumed.

Return Values

If the function succeeds, the return value is TRUE; the logging session was started successfully.

If the function fails, the return value is FALSE. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

The `DapPipeDiskLog` function initiates a disk logging session. Disk logging continues until the number of bytes specified in the `TDapPipeDiskLog` *i64MaxCount* has been logged or until the handle passed to the command is closed using `DapHandleClose`.

If the *pGetInfo* pointer is `NULL`, the following set of attributes is assumed:

```
iBytesGetMin = 8192, iBytesGetMax = 8192, dwTimeWait = 1000,
dwTimeOut = 0 (no time-out), iBytesMultiple = 1
```

Use *hLog* with `DapHandleQuery` to determine the ongoing session status. See `DapHandleQuery` for more information.

The `TDapPipeDiskLog` *i64MaxCount* value must be a multiple of the `TDapBufferGetEx` *iBytesMultiple* value or an error will be generated.

Additional Features

It is possible to mirror the logged data to a second file. To enable mirror logging, initialize the `TDapPipeDiskLog` structure by specifying `dpdl_MirrorLog` in the *`bmFlag`* field and appending a mirror file name to the string *`pszFileName`* points to.

When mirror logging is active, an error that causes logging to the primary file to stop will not terminate the entire logging session if the mirror logging can continue. Likewise, an error that causes the mirror logging to stop will not necessarily terminate the primary logging either.

Security

When `dpdl_ServerSide` is not set in the `TDapPipeDiskLog` structure, no file access control is enforced. The file names specified in *`pszFileName`* of the structure are either absolute file paths on the local machine or partial paths relative to the local directory where the application is launched.

When `dpdl_ServerSide` is set, disk logging may access files on a remote machine. The "Disk I/O" tab in the DAP control panel program provides security measures for controlling file access to the server machine.

The "Disk I/O" tab has two control entries, Permission and Default Path. They can be selected or specified to provide the desired control.

The Permission entry provides three selectable file access permission levels.

- Not allowed - No access is allowed at all on the server machine.
- Restricted - Access is restricted only to the path(s) specified in the Default Path entry and their subdirectories. The specified subdirectory path will be created if it does not already exist.
- Normal - Access is allowed anywhere reachable by the DAPcell/DAPcell Local server, except that new directories will not be created if they are not qualified by the paths specified in the Default Path entry.

The Default Path entry specifies one or more directories that data can be written to. If *`pszFileName`* is a partial file name and can be found in more than one of the paths specified in Default Path, the first occurrence is used.

When mirror logging is enabled, the presence of multiple default paths may have a different meaning if the mirror file name is a partial path name. In this case, only the default path in the same list order as the mirror file name in *`pszFileName`* is used to qualify it.

For example, if *`pszFileName`* is "`c:\log\logs\logfile.log;c:\log\mirrors\mirror.log`" and the default path value is "`c:\log;d:\backup`", the primary logging will go to `c:\log\logs\logfile.log` and the mirror logging will go to `c:\log\mirrors\mirror.log` since they are both qualified by the first default path "`c:\log`". If *`pszFileName`* is "`logfile.log;mirror.log`" the primary logging will go to `c:\log\logfile.log` and the mirror logging will go to "`d:\backup\mirror.log`" since the file names are relative and are qualified by default paths in the same list order, "`c:\log" for "logfiles.log" and "d:\backup" for "mirror.log".`

See the help file of the DAP control panel program for more information.

Example

In the following example, the target pipe is first flushed. Data from `\\PC92\Dap3\$BinOut` is then continuously logged to the file `c:\logfiles\logfile.log` on the server side, the same side where the DAP board is located. This log session creates or overwrites the target file. Default parameters of the `TDapBufferGetEx` structure are used to read data from the pipe. Note that if the permission level specified in the "Disk I/O" tab of the DAPcell

/DAPcell Local Service's control panel program does not allow access to this file, `DapPipeDiskLog` will return with an error.

```
TDapPipeDiskLog LogInfo;

DapStructPrepare(LogInfo);
LogInfo.bmFlags = dpdl_ServerSide | dpdl_FlushBefore;
LogInfo.pszFileName = “c:\\logfiles\\logfile.log”;
LogInfo.dwOpenFlags = DAPIO_CREATE_ALWAYS;

/* Open a log handle */
hLog = DapHandleOpen( “\\\\PC92\\Dap3\\$BinOut”,
  DAPOPEN_DISKIO);
/* Start a logging session */
if (hLog && DapPipeDiskLog(hLog, &LogInfo, NULL))
{
  /* Log some data */
  /* Check logging status periodically until it is done. */
}
else
{
  /* handle error */
}
/* Terminate the logging session */
if (hLog) DapHandleClose(hLog);
```

Version

This service is only available in DAPcell Server and DAPcell Local Server version 4.00 or later. It is not available in DAPcell Basic Server.

See Also

`TDapPipeDiskLog`, `TDapBufferGetEx`, `DapHandleQuery`

DapReinitialize

The `DapReinitialize` function reloads a fresh copy of DAPL to the target DAP board(s) along with all modules installed for the target DAP board(s). This is a destructive operation. All data on the target DAP board(s) will be lost.

```
BOOL __stdcall DapReinitialize(
   HDAP hAccel                           // Reinitialization handle
   );
```

Parameters

hAccel

Handle to the DAP board(s) to reinitialize. Requires the handle opened with write access. If this is a DAP handle, reinitialize the target DAP board only. If this is a server handle, reinitialize all the DAP boards on the server.

Return Values

Returns true if the DAP boards were successfully reinitialized, returns false if an error occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

The `DapReinitialize` function performs a hardware reset of the specified DAP board(s) and reloads the DAPL operating system along with all installed modules. This is the same as the initialization performed at system boot.

See Also

`DapModuleInstall`, `DapModuleLoad`, `DapModuleUninstall`, `DapModuleUnload`, `DapReset`

DapReset

The `DapReset` function performs an interlocked DAPL "`RESET`" on the target DAP board(s). If the operation is successful, the "`RESET`" is guaranteed complete before this function returns.

```
BOOL __stdcall DapReset(
    HDAP hAccel                              // Reset handle
    );
```

Parameters

hAccel

Handle to the DAP board(s) to DAPL `RESET`. If this is a DAP handle, perform the DAPL `RESET` operation on the target DAP board only. If this is a server handle, perform the DAPL `RESET` operation on all the DAP boards on the server.

Return Values

Returns true if the operation is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

This function requires read or write access to the target DAP's `$SysIn` and `$SysOut`. The function fails if either pipe has been opened with read or write access.

Use `DapReset` as the preferred method to `RESET` the DAPL interpreter. Since the DAPL interpreter runs on the processors on the DAP board, and not on the processor in the PC, a command issued from the PC through $SysIn to the DAP board usually will not complete before the service used to issue the command returns. `DapReset` guarantees that the `RESET` completes on the DAP board(s) before returning control to the calling thread

The effect on DAPL of this function is identical to that of issuing a `RESET` through `$SysIn`. However, it is not the same as using `DapLinePut`, or other pipe service, to send a `RESET` command to `$SysIn`. When you send a DAPL command to `$SysIn`, the command is placed in the com-pipe on the PC. The transfer to the DAP board may then begin immediately, if there is no other traffic blocking the transfer and the DAP board has space to accept the new command. Or the transfer may be delayed until other transfers complete and/or the DAPL interpreter makes room on the DAP board side of the com-pipe by processing the commands already present in `$SysIn` on the DAP board. In either case, the commands in `$SysIn` on the DAP board will not be processed until the DAPL interpreter is scheduled to run by the DAPL scheduler.

See the DAPL Reference Manual for more information about the `RESET` command.

See Also

`DapReinitialize`

DapServerControl

The `DapServerControl` function operates on the target server in a way that is specified by the control key string parameter.

```
BOOL __stdcall DapServerControl(
    const char * pszServer,              // UNC target server address
    const char * pszKey                  // Control key string
    );
```

Parameters

pszServer

Specifies the target server address in UNC format. When a NULL pointer is specified, the local server is used.

pszKey

A control key string, identifying the operation to perform on the target server. See the section Control Keys for the supported control keys.

Control Keys

The parameter *pszKey* supports the following control keys:

"ServiceRestart"

Stop and restart the DAPcell service on the target server. A successful return of the function guarantees that the target service has been restarted. An option *[timeout=dddd]* can be specified in the key string where "*dddd*" is an integer value in milliseconds. This value forces the function to return in the specified amount of time even though the operation has not completed. If the option is not present, a default value of 60000 is used. Note that the time required to restart the service may vary with the server configuration. Specifying a small timeout value may cause the function to return with a time-out error while the service restart is still in progress and may eventually succeed.

"ServerShutdown"

Initiate a request to shut down the target server machine to a point where it is safe to turn off the power. If the target server supports the power-off feature, it also turns off the power. An option *[force=true]* can be specified in the key string to force processes to terminate; as a result, processes may lose data. This operation is asynchronous. A successful return of the function does not imply the completion of the operation.

"ServerReboot"

Initiate a request to shut down and restart the target server machine. An option *[force=true]* can be specified in the key string to force processes to terminate; as a result, processes may lose data. This operation is asynchronous. A successful return of the function does not imply the completion of the operation.

Return Values

Returns true if the operation is successful. Returns false if an error has occurred. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

All three `DapServerControl` functions invalidate client handles previously opened to the target server. It is a good practice to close all handles whenever possible before calling `DapServerControl`.

When the target is a remote server, control operations require a special security privilege to be enabled by the remote server. See the section Security for more information.

Control operations generally need the help of a running DAP service on the target server. However, if both the client and the server machines are running Windows NT or newer, the *"`ServerShutdown`"* or *"`ServerReboot`"* operation may succeed without a running DAP service.

Security

When a DAP service is running, operations may be restricted by the DAP service security control. A local server enforces no restrictions. A remote server, by default, allows *"`ServiceRestart`"* operation only. To allow other operations or to disable all operations from a remote client, use the DAP control panel program on the remote server to adjust the privilege level.

When no DAP service is running on the target server, `DapServerControl` function may use Windows services to perform the *"`ServerShutdown`"* or *"`ServerReboot`"* operation. This is only possible if both the client and the server machines are running Windows NT or newer. In this case, Windows security control overrides the DAP service security control. These two operations succeed on a local server only if the caller has the Windows NT "Shut down the system" right enabled and on a remote server only if the caller has the Windows NT "Force shutdown from a remote system" right enabled on the remote server.

See Also

`DapReinitialize`, `DapReset`

DapStringFormat

The `DapStringFormat` function writes a formatted string of characters to a DAP com-pipe.

```
BOOL __cdecl DapStringFormat(
    HDAP hAccel,                        // Open handle of the target pipe
    const char *pszFormat,              // Pointer to string
    ...
    );
```

Parameters

hAccel
Handle to DAP com-pipe to receive characters.

pszFormat
Pointer to format string which is compatible with *wsprintf*.

...
List of parameters for use with *pszFormat*.

Return Values

If the function succeeds, the return value is TRUE; the string is written.

If the function fails, the return value is FALSE; there is something wrong with the handle or the formatted string exceeds 1024 characters. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Writes a string of characters to the DAP com-pipe after formatting the string with the parameters specified in *pszFormat*. *pszFormat* is a format string which supports the format specifications shown in the documentation for the Windows API function *wsprintf*.

The resulting string after formatting must not exceed 1024 characters.

DapStringGet

The `DapStringGet` function gets a string of characters from the DAP com-pipe.

```
BOOL __stdcall DapStringGet(
    HDAP hAccel,                    // Handle to target pipe
    int iLength,                    // Length of pszBuffer
    char *pszBuffer                 // Pointer to buffer
    );
```

Parameters

hAccel
: Handle to DAP com-pipe from which to read a null-terminated string.

iLength
: Length of *pszBuffer* including space for the null character.

pszBuffer
: Pointer to buffer to store null-terminated string.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE; there is something wrong with the handle or *pszBuffer* is `NULL`. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Reads characters from the DAP com-pipe and places them in *pszBuffer* up to the first carriage-return character.

If more than *iLength - 1* characters are encountered before a carriage-return is found, it returns *iLength - 1* characters, and indicates success.

Line-feed characters are ignored and are never placed in the string.

See Also

`DapCharGet`, `DapLineGet`

DapStringPut

The `DapStringPut` function writes a string of characters to a DAP com-pipe.

```
BOOL __stdcall DapStringPut(
    HDAP hAccel,                    // Handle to target pipe
    const char *psz                 // String to write to DAP
    );
```

Parameters

hAccel

Handle to DAP com-pipe.

psz

Null-terminated string to write to DAP com-pipe. If *psz* is `NULL` does nothing and returns success.

Return Values

If the function succeeds, the return value is TRUE; the string is written.

If the function fails, the return value is FALSE; there is something is wrong with the handle. Call `DapLastErrorTextGet` to retrieve additional information about the error.

Remarks

Writes a string of characters to a DAP com-pipe. No terminating character is appended to the string so to send a line of text to the DAPL interpreter the string must include an explicit carriage-return `("\r")`.

See Also

`DapCharPut`, `DapLinePut`

DapStructPrepare

The `DapStructPrepare` function prepares a DAPIO32 structure for initial use.

```
void __stdcall DapStructPrepare(
    void *pStruct,                    // Pointer to structure
    int size                          // Size of structure to initialize
    );
```

Members

pStruct

Pointer to the structure to initialize. The structure can be any valid DAPIO32 structure whose first field is *iInfoSize*.

size

Size in bytes of the structure to initialize.

Remarks

The `DapStructPrepare` function prepares a DAPIO32 structure for initial use. All fields are initialized to zero. The *iInfoSize* field is set to the size of the structure expected by the DAPIO32 function.

A template is available for use in C++ applications. It takes a reference to a DAPIO32 structure and does not require the *size* parameter. See the example below for usage.

Examples

Declare and initialize a `TDapBufferGetEx` structure . Any non-zero fields must be initialized separately.

C example:

```
TDapBufferGetEx GetInfo;
DapStructPrepare (&GetInfo, sizeof(GetInfo));
GetInfo.iBytesGetMax = 1024;
```

...

C++ example using the template:

```
TDapBufferGetEx GetInfo;
DapStructPrepare (GetInfo);
GetInfo.iBytesGetMax = 1024;
....
```

5. Version Information

This chapter describes the DAPIO32 interface version 2.13.

DAPIO32 Version 2.13

Changes to the DAPIO32 interface since the 2.12 release

The 2.13 release of DAPIO32 continues to support all interfaces declared in the 2.12 release.

Changes

- The type of name or key string pointer field is changed from "`char *`" to "`const char *`" in structures `TDapCommandDownload`, `TDapHandleQuery`, `TDapPipeDiskFeed`, and `TDapPipeDiskLog`.

New DapHandleQuery keys

- `"DaplEventLog"`
- `"DaplInputAnalogGains"`
- `"DaplInputAnalogVRanges"`
- `"DaplInputMasterDivideMax"`
- `"DaplInputScanTimeMin"`
- `"DaplInputScanTimeMax"`
- `"DaplInputScanTimeIncrement"`
- `"DaplOutputScanTimeMin"`
- `"DaplOutputScanTimeMax"`
- `"DaplOutputScanTimeIncrement"`
- `"DaplMonitorData"`
- `"DaplName"`
- `"ServerOsSystemType"`
- `"ServerSystemType"`

Updated DapHandleQuery keys

- `"DaplInputCount"`
- `"DaplOutputCount"`
  `"DaplOverflowCount"`
  `"DaplSampleTimeIncrement"`
- `"DaplSampleTimeMaxAnalog"`
- `"DaplSampleTimeMaxDigital"`
- `"DaplSampleTimeMinAnalog"`
- `"DaplSampleTimeMinDigital"`
- `"DaplSupports"`
- `"DaplSymbol"`
- `"DaplUnderflowCount"`
  `"DaplUpdateTimeIncrement"`

- `"DaplUpdateTimeMaxAnalog"`
- `"DaplUpdateTimeMaxDigital"`
- `"DaplUpdateTimeMinAnalog"`
- `"DaplUpdateTimeMinDigital"`
- `"DapModuleInstall"`
- `"DapModuleLoad"`
- `"DapModuleUninstall"`
- `"DapModuleUnload"`

DAPIO32 Version 2.12

Changes to the DAPIO32 interface since the 2.11 release

The 2.12 release of DAPIO32 continues to support all interfaces declared in the 2.11 release.

New functions

- `DapBufferPeek`
- `DapServerControl`

New DapHandleQuery keys

- `"BindTransport"`
- `"DaplSymbol"`
- `"DaplWarnMsg"`
- `"DaplWarnNum"`

Updated DapHandleQuery keys

- `"DaplInputChannelGroupSize"`
- `"DaplSampleTimeMinAnalog"`
- `"DaplSampleTimeMinDigital"`

DAPIO32 Version 2.11

Changes to the DAPIO32 interface since the 2.10 release

No functional changes since the 2.10 release.

Minor fixes on the propagation of time-out warnings in functions `DapBufferGet` and `DapBufferPut`.

DAPIO32 Version 2.10

Changes to the DAPIO32 interface since the 2.00 release

The primary enhancement in the 2.10 release of the DAPIO32 interface is the addition of support for installation and loading of DAPL 2000 custom modules.

The 2.10 release of DAPIO32 continues to support all interfaces declared in the 2.00 release DAPIO32.H. You can use the new DAPIO32.DLL with a program compiled with the old DAPIO32.H.

New functions

- `DapModuleInstall`
- `DapModuleLoad`
- `DapModuleUninstall`
- `DapModuleUnload`
- `DapReinitialize`
- `DapReset`

New DapHandleQuery keys

- `"DapName"`
- `"DiskFeedEnabled"`
- `"DiskLogEnabled"`
- `"IsRemote"`
- `"ModuleInstallEnabled"`

Changes

- `DapHandleQuery` now supports the `DAPIO_VARIANT` return data type.
- The result type field of the `TDapHandleQuery` structure is no longer a union, but simply an integer field with the name of "eResultType".

DAPIO32 Version 2.00

Changes to the DAPIO32 interface since the 1.12 release

The 2.00 release of DAPIO32 continues to support all interfaces declared in the 1.12 release DAPIO32.H. To continue using the 1.12 interfaces use either the 1.12 DAPIO32.H or the DAPIO32.H shipped with this version of DAPIO32.DLL. You can use the new DAPIO32.DLL with a program compiled with the old DAPIO32.H.

Changes

- `DapComPipeCreate` accepts a new syntax for the pipe attribute option list in its pipe information string argument. The old syntax is still supported for compatibility, but mixed syntax will be rejected.

  With the new syntax, the pipe attribute `"type=xxx"` replaces the old attributes `"width=x"` and `"binary"` or `"text"`. The new syntax also allows applications to specify pipe attributes for both the PC side and the Data Acquisition Processor side in one option list.

- `TDapBufferGetEx` structure now includes a new field *`iBytesMultiple`*. With this new field, `DapBufferGetEx` always returns an amount of data that is an integral multiple of the value of this field.

- `TDapBufferPutEx` structure also includes the new field *`iBytesMultiple`*. `DapBufferPutEx` now always puts an amount of data that is an integral multiple of the value of this field.

Additions

- `TDapIoInt64`
- `TDapPipeDiskFeed`
- `TDapPipeDiskLog`
- `DapHandleQueryInt32`
- `DapHandleQueryInt64`
- `DapPipeDiskFeed`
- `DapPipeDiskLog`
- `DapStructPrepare`

DAPIO32 Version 1.12

Changes to the DAPIO32 interface since the 1.11 release

The 1.12 release of DAPIO32 continues to support all interfaces declared in the 1.11 release DAPIO32.H. To continue using the 1.11 interfaces use either the 1.11 DAPIO32.H or the DAPIO32.H shipped with this version of DAPIO32.DLL. You can use the new DAPIO32.DLL with a program compiled with the old DAPIO32.H.

Additions

- `DapOutputEmpty`

DAPIO32 Version 1.10

Changes to the DAPIO32 interface since the 1.00 release

The 1.10 release of DAPIO32 continues to support all interfaces declared in the 1.00 release DAPIO32.H. To continue using the 1.00 interfaces use the 1.00 DAPIO32.H; do not switch to the DAPIO32.H shipped with any newer version of DAPIO32.DLL. You can use the new DAPIO32.DLL with a program compiled with the old DAPIO32.H.

Changes

- The `TDapBufferGetEx` structure has been changed to include time-out intervals and several field names have been changed to force compilation failure if old code which uses the `DapBufferGetEx` service is compiled without modification. The `DapBufferGetEx` service in the DAPIO32.DLL continues to support the old interface for compatibility with old binaries.
- The `DapHandleQuery` service has many new keys.
- All simple Get/Put operations have a built-in 20 second time-out.
- `DapComPipeCreate`/`DapComPipeDelete` now manage the com-pipe on the DAP board as well as the com-pipe on the PC. This change means that it is no longer necessary to create the com-pipe on the DAP board manually. Both of these operations are `DESTRUCTIVE`. They remove all user-defined application definitions on the DAP board including custom commands.
- `DapCommandDownload`/`DapConfig` now looks for files first in the current directory and then in the executable directory. Previously these commands only looked for files in the current directory.

Additions

- `DapBufferPutEx`

- `DapLineGet`
- `TDapBufferPutEx`

Index