DAPtools for Python Manual

Version 1.10

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

Contents

Table of Contents

1. Introduction

The *DAPtools™ for Python* interface (informally "*DAPpython*" in this manual) is a set of "wrapper functions" that enable applications written in the *Python*™ programming language to configure and control operation of *Data Acquisition Processor*™ (DAP™) boards from Microstar Laboratories™, allowing high-performance data acquisition applications controlled from a normal Python application environment.

This version of *DAPtools for Python* supports both Windows™ and GNU/Linux™ operating systems. Without implying anything about roles of the Linux kernel, GNU Project components, and other software subsystems, we will defer to widespread practice and refer to GNU / Linux distributions informally as *Linux* systems.

Using Python for purposes of time-critical data acquisition might seem crazy at first. Such applications are typically laden with stringent *real-time constraints*. Python applications do what they do, when the appropriate functions happen to run, with execution time undefined. Processing can be delayed and rescheduled at any time because of various background activities such as inter-process communication or free memory reclamation ("garbage collection"). This would be poison in situations when being a millisecond late to respond to a hardware message means the difference between a successful application and a major data-corrupting failure.

With Data Acquisition Processor boards, however, *no direct interaction with devices or device drivers is required*. (Or even possible.) All of the complicated real-time interactions with hardware devices are delegated to the DAPL™ operating system running in an embedded environment on the DAP boards. Automatic buffering and data transfer management are provided by the DAPL system. On the host application side, the DAPIO programming interface provides access functions for configuration, process management, and data storage. The result is that you do not need special real-time operating system configurations or customized Python implementations. The obligations of the Python application are:

- Configure the DAP board to do exactly what needs to be done.
- Start the DAP processing.
- Accept all of the data requested, at a sufficient average rate to keep pace with the net data transfers and avoid a backlog of unprocessed data.

The DAPIO interface is implemented differently depending on the operating system environment.

- For the Windows environment, the *dapio32.dll* is provided by all editions of the *DAPtools Software*.
- For the Linux environment, the *dapio32.so* library file is provided by the *Accel32 for Linux* or *Accel64 for Linux* software distributions, available for download from the Microstar Laboratories Web site.

There are no additional software packages that you will need to license and maintain. In concept, you don't even need the *DAPpython* code – you could write your own wrapper functions for the DAPIO interface. But who would want to do that?

Dynamic library modules operate on a fundamental concept of "typed storage bound to symbolic addresses" in the manner of C and C++ languages. This works efficiently for data acquisition processing that transfers large volumes of raw binary data. It is not consistent with the "objects bound to symbolic names" concept of Python, which provides powerful manipulations of diverse objects, but with relatively unquantified internal workings.

To access functions from DLLs and dynamic libraries, Python provides the `ctypes` library – which is used extensively by the *DAPpython* interface. The access works at two levels.

1. The lower `ctypes` level. This consists of minimal Python "wrapper" functions to access the DAPIO functions, using the `ctypes` library and its special data types. While this brings the DAPIO functionality into the Python environment directly, the unfamiliar data types are awkward to use.

2. The higher `utilities` level. Some important functions of the `ctypes` level are "wrapped" in more convenient forms that use familiar Python objects and provide exception handling of the sort familiar to Python programmers.

Applications programmers can use either level. But in practice, you will probably find the `utilities` functions easier to use. For special needs, you always have the option to use the functions from the `ctypes` level directly – everything is there and everything is compatible.

Interactions with the DAP board are required to be single-threaded. This doesn't mean that the application must be single-threaded. A multi-threaded application should designate a single thread that manages all of the DAP interactions.

Manuals you will need to know about

This document is not intended as complete coverage for the DAPL system or the DAPIO programming interface. For full information about these important topics, you will need to have the following manuals available (from your copy of the *DAPtools Software*, or downloaded from the *Docs* section of the Microstar Laboratories site on the Web).

- For PCI-connected DAP boards such as the DAP 840 or the DAP 5200a, find information about the DAPL system in the *DAPL 2000 Manual* (`DAPL2000.PDF`).

- For USB-connected DAP boards such as the xDAP 7410, find information about the DAPL system in the *DAPL 3000 Manual* (`DAPL3000.PDF`) and the "supplement" manual *DAPL 3000 Extensions for xDAP Family* (`SUPPXDAP.PDF`).

- Find information about the DAPIO programming API in the DAPIO32 Reference Manual (`DAPIO32.PDF`).

2. Installation

DAPtools for Python is intended for Python versions 3.0 and later. There have been reports of successful applications using the lower-level `dapio.py` interface module with the Python 2.7 versions, though this is not confirmed or supported. The higher-level utility modules and examples are *not* expected to work under Python 2.7. It is presumed that you have your Python 3.X system installed and operating.

It is still common for the obsolescent Python version 2.7 to be the default version installed by many Linux distributions. For these distributions, you will need to take extra care to install one of the Python 3.X releases as well. To make sure that you run the correct version, you will then need to use a command line something like the following to start your Python application.

```
Python3.4  "application script file"
```

DAPtools for Python does not yet have a formal Python-style installer, so you will need to make some system adjustments manually.

Installing under Windows

When you run the *DAPtools for Python* installer from the main menu of the *DAPtools Software* CD program `SETUP.EXE`, it will place a copy of all of the Python code and test scripts on your main disk drive. The default location is the folder

"C:\Program Files (x86)\Microstar Laboratories\DAPtools\Python"

You will need to modify the the system Environment Variables so that Python software knows about this location. To do this:

1. Right-click on *Computer* and pick *Properties* from the context menu that pops up. (You can also get to this same place via the Control Panel.)

2. Select the *Advanced System Properties* link from the dialog.

3. Click the *Environment Variables* button.

4. If the display of existing environment variables does not show a `PYTHONPATH` item, click the New button to create one. See the comments below about whether this should be a system or user variable.

5. Modify the value of the variable to include a new path to the *DAPtools for Python* folder. New path specifications are separated from the previous items using a semicolon character.

The `PYTHONPATH` environment variable can be placed in the *System* section, making the DAPIO interface available to all users, or in the *User* section for each individual user who needs the access. There is little difference on a one-user workstation. But be careful, a `PYTHONPATH` item in a *User* area will override one that is already present in the *System* area.

Installing Under Linux

The *DAPtools for Python* distribution is a gzipped-tar archive file. You can find it in the DAPtools Software CD files under the `/linux` directory. Extract the files anywhere, but the suggested location is a `DTpython` subdirectory under the `MicrostarLaboratories` software install directory. The suggested location is `/usr/local/MicrostarLaboratories/DTpython`.

The Linux system must know some extra things to access the *DAPpython* code:

- the location of the `dapio32.so` library files
- the location of the Python `dapio.py` interface file and its related utilities.

How you provide this information depends on whether you (as the administrator) intend to make the DAPtools for Python files accessible to individual users or to all users.

For all users:

- To make the dynamic library accessible, look up the `ldconfig` man page for information about adding the `dapio32.so` directory into the system's shared library list. The typical location for this library file is:

  `/usr/local/MicrostarLaboratories/lib`

- Next, establish the Python system supplemental search path for Python libraries. Typical Linux systems will have a script file `/etc/profile` or a directory of startup script files `/etc/profile.d/` that run at the beginning of every shell script session. Locate or create the appropriate script file and add to it the following command lines.

```
# Establish Python access to the dapio programming interface library
PYTHONPATH="/usr/local/MicrostarLaboratories/DTpython"
```

Be careful! If your system has already set up a default PYTHONPATH variable, you should add the new path to the end of the previous list, separating the new item by a colon character, rather than entirely replacing the previous value.

For individual users:

Command lines like the following need to be put in the shell startup script in the individual user's home directory area (depending on the actual install location of the files).

```
export LD_LIBRARY_PATH=/usr/local/MicrostarLaboratories/lib
export PYTHONPATH=/usr/local/MicrostarLaboratories/DTpython
```

Before making these changes it is a good idea to run a command shell and check whether these variables are already defined.

```
>>  echo  $LD_LIBRARY_PATH
>>  echo  $PYTHONPATH
```

If defined values already exist, you should add the new paths to the end of the values that were set up previously, separating the new paths from the previous items using a colon character.

A point of confusion here is that various distributions are not consistent about which shell configuration files are provided – or not. The files are “hidden files” typically named `.bashrc` or `.bash_profile` . If you have one of these, you can modify it. If you do not have one of these, you can create one with your favorite text editor.

3. Low-level DAPIO access

The DAPIO interface provides a language-independent interface for accessing the DAPcell Services that manage Data Acquisition Processor boards. The implementation of the API interface on the system side is provided in the *dapio32.dll* interface module. Access to external interfaces of this sort are achieved using the Python `ctypes` library. In the Python environment, the DAPIO functions are represented as methods of an `HDap` ("DAP Handle") object as define in the `dapio.py` library file. The `HDap` method names resemble the raw function names implemented by the DAPIO interface. For example, the DAPIO function

```
DapHandleOpenA
```

has a corresponding `HDap` object method

```
HDap.Open( … )
```

Access to DAPIO functions is very much like access to a network communication channel. You must

- instantiate a DAP handle object
- use the DAP handle to open a connection
- perform any required activities: configure signal properties, log data, etc.
- close the DAP handle channel
- release the DAP handle object

Functions at the DAPIO level make extensive use of elements from the `ctypes` library. To make application programming a little easier, this package provides supplemental `Utilities` level functions that combine features of the `ctypes` library, DAPIO services, and familiar Python elements. Presuming that the `Utilities` level functions are preferable when provided, descriptions in this manual of lower-level access methods are terse – or omitted. If you need to use DAPIO-level methods directly, you can see how the `Utilities` level functions use them, and do similar things in your application code.

For additional information about what DAPIO services do, find this in the *DAPIO Reference Manual*.

Summary of the 'ctypes' low-level wrapper functions

`HDap.BufferGet(self, length, bufobj)`	Receive a data block of a stated size from the DAP, putting it into a provided storage area, using a simplified strategy.
`HDap.BufferGetEx(self, info, bufobj)`	Receive a data block from the DAP in a configurable manner specified by a special `TDapBufGetEx` information block.
`HDap.BufferPeek(self, ctrl, bufobj)`	Non-destructively observe a data stream, in a manner specified by a special `TDapBufPeek` information block.
`HDap.BufferPut(self, length, bufobj)`	Send a block of data of specified size from application storage to a DAP, using a simplified strategy.
`HDap.BufferPutEx(self, info, bufobj)`	Send a data block to the DAP in a manner configured by a special `TDapBufPutEx` information block, taking the data from a specified storage area.
`HDap.CharGet(self)`	Receive a single 8-bit character value from the DAP through the communication channel associated with this handle.
`HDap.CharPut(self, charval)`	Send a single 8-bit character value from the host to the DAP through the communication channel associated with this handle.
`HDap.ComPipeCreate(self,options)`	(ADVANCED) Create a nonvolatile communication pipe channel between the host and the DAP board associated with this handle, on-the-fly.
`HDap.ComPipeDeleteA(self,options)`	(ADVANCED) Remove a nonvolatile communication pipe channel between the host and the DAP board associated with this handle, on-the-fly.
`HDap.Config(self,dapfile)`	Send a configuration script from the file path specified to the DAP associated with this text communication pipe handle.
`HDap.Open(self,pipename,openflag)`	Open a connection to an existing DAP or DAP communication pipe, specified by name, using the option specified.
`HDap.Close(self)`	Close a connection a previously opened DAP or communication pipe handle.

`HDap.QueryTxt(self,keystr)`	Send an information query to the channel associated with the DAP handle, of the query type indicated by the keyword string, for a query variation that returns a text message.
`HDap.QueryInt32(self,keystr)`	Send an information query to the channel associated with the DAP handle, of the query type indicated by the keyword string, for a query variation that returns a 32-bit integer value.
`HDap.QueryInt64(self,keystr)`	Send an information query to the channel associated with the DAP handle, of the query type indicated by the keyword string, for a query variation that returns a 64-bit integer value.
`HDap.InputAvail(self)`	Send a query to a data communication pipe, text or binary data from DAP to host system, to obtain the most-recent estimate of the number of bytes of data available to be taken by the host.
`HDap.InputFlush(self)`	Command a data communication pipe, text or binary data from DAP to host system, to eliminate any previously produced but unprocessed data currently buffered in the data channel, using timing defaults.
`HDap.InputFlushEx(self, TimeOut, Wait)`	Command a data communication pipe, text or binary data from DAP to host system, to eliminate any previously produced but unprocessed data currently buffered in the data channel, using configurable timing.
`HDap.Int16Get(self)`	Receive a single 16-bit integer (sample) value from the DAP through the communication channel associated with this handle.
`HDap.Int32Get(self)`	Receive a single 32-bit integer (sample) value from the DAP through the communication channel associated with this handle.
`HDap.Int16Put(self, shortval)`	Send a single 16-bit integer (sample) value from the host system to the DAP through the communication channel associated with this handle.
`HDap.Int32Put(self, longval)`	Send a single 32-bit integer value from the host system to the DAP through the communication channel associated with this handle.
`HDap.LastHostErrorGet(self)`	Report the error text of the last failed operation on the host system. (Does not see errors that occurred on the DAP board.)

`HDap.LineGet(self, timeout)`	Retrieve the text of a message sent by the DAPL system through the associated text communication channel, with timeout for the case that no such message was produced, reporting text and line length.
`HDap.LinePut(self, strobj)`	Construct a terminated text line from the string object, and send it to the DAP through the text communication channel associated with this handle.
`HDap.ModuleLoad(self,binpath,flags)`	(ADVANCED) Load a command module defining embedded “custom” DAP processing, in a nonvolatile manner, from the file specified, downloading it to the DAP board associated with this handle.
`HDap.ModuleUnload(self,modname,flags)`	(ADVANCED) Removes a command module previously downloaded to a DAP board, so that it no longer available and frees on-board resources.
`HDap.OutputEmpty(self)`	Removes any data, binary or text, previously sent by the host to the DAP board through the associated data communication channel, but held in buffer memory and not yet processed by the DAP board.
`HDap.OutputSpace(self)`	Reports the size of unused buffer memory space, in bytes, available for accepting new data or text to be delivered to the DAP board through the associated data communication channel.
`HDap.PipeDiskFeed(self, feeder, buffer)`	(WINDOWS) Initiate a direct streaming data transfer from the host system to a DAP from a disk file, using transfer information from a special TDapPipeDiskFeed information object, and using buffer storage provided by a TDapBufferPutEx buffer configuration object.
`HDap.PipeDiskLog(self, feeder, buffer)`	(WINDOWS) Initiate a direct streaming data transfer from the DAP to a disk file on the host system, using transfer information from a special TDapPipeDiskLog object, and using buffer storage provided by a TDapBufferGetEx buffer configuration object.
`HDap.Reset(self)`	Force a reset operation to be performed on a DAP board associated with this handle, regardless of the state of normal command processing and DAP board operation.
`HDap.StringGet(self, maxcount)`	(ADVANCED) Retrieve text lines delimited by LF characters from the associated communication channel, reporting the number of characters returned along with the text.

`HDap.StringPut(self, strobj)`	(ADVANCED) Send text characters from the host to the DAP, in the form provided, without injecting any line delimiter control characters.
`HDap.DoubleGet(self,timeout)`	Receive a single 64-bit double-precision floating point binary value from the DAP through the communication channel associated with this handle.
`HDap.FloatGet(self,timeout)`	Receive a single 32-bit double-precision floating point binary value from the DAP through the communication channel associated with this handle, reporting the value as 64-bit floating point.

4. DAPL configuration files

This topic is too broad to cover in one short section. But at least the discussion should underscore the fact that DAP boards, in order to operate independently of the host, need to be told what to do by means of a DAPL configuration script.

The DAPL configuration script is an ordinary ASCII text file, containing commands to the *command interpreter* task in the DAPL system. The things that DAPL configuration scripts typically need to do for each run:

1. Clear any stale operations or configurations.
```
// Clear prior operations and clear memory
RESET
```

2. Define any special on-board elements needed for processing.
```
// Declare user-defined pipes for data movement between tasks
PIPES  pAveraged WORD
PIPES  pSignal  WORD
```

3. Define the properties of input sampling. For this example, there are four channels, each sampled 25000 times per second.
```
// 4 channels x 5 microseconds = 20 microseconds
//    --> 50000 samples per second in each sampled channel
IDEFINE  Sample4
  CHANNELS  4
  SET Ipipe0 S0
  SET Ipipe1 S1
  SET Ipipe2 S2
  SET Ipipe3 S3
  TIME  5
END
```

4. Define the properties of output signal updating. For this example, there is one output signal that will update a D-to-A converter 10000 times per second.
```
// 1 channels x 100 microseconds
//    --> 100 microsecond intervals to update D-to-A
ODEFINE  Update1   1
  SET Opipe0 A0
  TIME  100
END
```

5. Define the processing required. This could be as simple as copying sample values to the predefined `$BinOut` communication pipe. As a more elaborate example, a waveform is generated to drive an output signal, while averaging in groups of 10 is applied to four input signals to reduce the effects of random noise.
```
// Processing
PDEFINE  GenSend
 // Calculate sine wave data for analog output
 SINEWAVE(10000,250,Opipe0)
 // Average to reduce from 50000 to 5000 samples per second
 BAVERAGE(Ipipes(0..3),4,10,pAveraged)
 COPY(pAveraged,$BinOut)
END
```

6. Start the processing. This doesn't necessarily need to be done in the same script – you can configure what to do first, then start things later. But if there is nothing else that needs to happen, starting the configuration immediately at the end of the script is the best way to get things moving.
```
// Start DAP activity immediately!
START
```

Collect these configuration commands together in a text file, and save the file in an application file folder where you can find it easily. You will need the configuration file when you establish communication connections, as discussed in the next section.

As long as the commands arrive, in the correct sequence, it doesn't really matter whether they come from one file or multiple files, or even individual command lines sent one at a time. Once a START command arrives, however, the entire configuration is started all at once, very quickly. You need to be ready receive any data you asked for.

There are many things you can do in the DAPL configuration:

- Select data based on features observed in the sampled data stream
- Apply digital filtering
- Select occasional data blocks rather than delivering everything continuously
- Apply mathematical corrections such as offset adjustments or unit conversions
- Apply frequency transforms
- Look for correlations between data streams
- Collect processing statistics

Unfortunately, to learn about this, you will need to study the tutorial material at the www.mstarlabs.com Web site, or read more about DAPL system configurations in the *DAPL Manual*.

5. Opening and closing connections

This section discusses the utility functions that you typically will use to establish connections to a DAP board and its communication channels. The `daphandle.py` module provides Utility functions that help to make this easy.

The DAP board provides default communication channels that will serve the needs for most applications:

- `$SysOut` - Message texts and error diagnostics from DAP to host
- `$SysIn` - Command text from host to DAP
- `$BinOut` - Binary data blocks from DAP to host
- `$BinIn` - Binary data blocks from host to DAP

You can establish connections to all of these in one operation. This does no harm if you don't need some of the communication channels – if you obtain access to any channel, you basically "claim possession" of the DAP board until the connections are released. (Just make sure that you do not deliver data for data transfers and then disregard the data sent – the unused data can clog the channel and becomes a roadblock to other processing.)

Accessing the `daphandle.py` library file

First, import the `daphandle.py` module.

```
# Utility functions for DAP connections
from daphandle import *
```

Preparing for typical single-DAP operation

Next, call the `Open1Dap` function. This function assumes that you have one board in your local system, which means that it will have the name `Dap0`. This function also assumes that you have previously established the processing configuration that you want, in the form of a DAPL configuration file on your disk drive. (Unfortunately, the DAPIO operation doesn't know about Python disk-like objects in memory.)

```
# Obtain handles to the predfined communication pipes
DH = Open1Dap("c:\\\\DAQapp\\config.dap")
```

The results are provided in the form of a list of `HDap` handle objects, for the four communication channels in the order previously presented. You can index them by name if you wish:

```
# Select the handle for the channel that the DAPL system uses for error messages
errhandle = DH[ SysOut ]
```

Preparing for typical two-DAP operation

Some systems use multiple, coordinated DAP boards, for example, two DAP 5400a boards with a master/slave hardware connection that results in 16 channels sampled simultaneously at high rates. Systems with a 2-DAP configuration can use the `Open2Daps` utility function instead of the `Open1Dap` function.

The boards are assumed to have names `Dap0` and `Dap1` and be located together in the local host system. A DAPL configuration script must be provided for each board, and available on the host's local file system.

```
# Obtain handles to the predfined communication pipes fortwo DAP boards
DHL = Open2Daps("c:\\\\DAQapp\\config0.dap",c:\\\\DAQ\\config1.dap" )
```

Similar to the `Open1Dap` function, you will receive `HDap` handle lists, but in this case, there is a list for each of the two DAP boards. To access individual boards, you will need to separate the handle sets.

```
# Isolate the DAP handle lists for the two DAP boards
DH0 = DHL[0]
DH1 = DHL[1]
```

Cleaning up at the end of the application

The DAP board wouldn't care if you just terminated your application and left a configuration running forever. But this could leave you in a jam. Suppose that you come back at a later time. The DAP board will see that it has communication channels open, which it interprets as meaning that some other process has locked access. The DAP will have no way to know that "*some other process*" stopped running a long time ago. You remain locked out until you shut down and then restart your DAP services in the Data Acquisition Processor control panel applet. Not convenient.

To avoid these situations, make sure that you shut down processing and release the DAP communication pipes. Pass the operation the list of handles that was received at the time handles were opened.

```
# Release the communication pipe handles, for one-DAP case
Close1Dap(DH)

# Release the communication pipe handles, for two-DAP case
Close2Daps(DHL)
```

Advanced topic: More access

There could be times when you need to open connections using different options than the ones assumed by the utility functions. You have other options to open handles individually, using the options that you specify, with the DAP boards that you specify, for only the channels you need to use. Some example situations:

- Opening DAP boards other than `"Dap0"` and `"Dap1"`
- Opening with special access modes
- Opening pipes to DAP boards on remote servers
- Opening communication pipes that are not in the predefined set

For purposes such as these, you can use the `OpenDAPHandle` function. For this function, you specify the complete path to the resource that you wish to access. For example, to open a query mode pipe to board `"Dap1"` provided by a separate machine `"DapHost55"` (you need the DAPcell Networking Server software from DAPtools Professional for this) open the connection as follows.

```
# Open a selected DAP board for query access on the remote system
uncpath = "\\\\DapHost55\\Dap1"
qhandle = OpenDAPHandle(uncpath,DAPOPEN_QUERY)
```

When you are done, you can close the handles individually.

```
# Close the query handle
CloseDAPHandle(qhandle)
```

Advanced topic: Opening query channels

Information about activity on a DAP board can be obtained without interfering with any of the data channels, using a query channel. Unlike the generic handle opening functions, this does not assume which DAP board you intend to access, and this doesn't necessarily associate with any communication pipe. That is what makes this an *advanced topic*. For example suppose you want to access the `"DaplMemFree"` information on the board "Dap0". (This can tell you whether the buffer memory is being overrun by unprocessed data.) To do this, you will need to open a handle for the board, specifying the board name only. For example: `"Dap0"` .

```
# Open a selected DAP board for query access
QH = OpenQueryHandle("\\\\.\\Dap0")
```

The queries themselves are more complicated. See the DAPIO32 Reference Manual for details about about all of the query options and the handle specifications for accessing them.

When you have finished with query processing, you should close the connection as you would any connection to data communication pipes.

```
# Close the connection for query access
CloseQueryHandle( QH )
```

function: OpenDAPHandle

Open a connection to a DAP board and associate with an `HDap` handle object.

hd = OpenDAPHandle(uncpath,option)

Parameters

<uncpath>
A string specifying a path to the desired resource (server, DAP, communication pipe) .

<option>
An enumerator specifying the open mode.

Returns

<hd>
Opened `HDap` handle object.

Library

`daphandle.py`

Exceptions

`DapException`

Description

An `OpenDapHandle` utility function facilitates construction of an `HDap` handle object, and opens that object to establish a connection to a DAP board. This function can be useful in situations where a more generic operation that opens the predefined communication handles does not provide sufficient configuration control.

The resource to be opened is specified by the *uncpath* parameter. This string specifies a path to the resource using an UNC notation. The string can take one of the following forms:

- `""` The local DAPcell service
- `"\\\\."` The local host server for a DAP board
- `"\\\\host"` A remote host server for a DAP board
- `"\\\\.\\dapname"` A DAP board on the local host
- `"\\\\host\\dapname"` A DAP board on a remote host
- `"\\\\.\\dapname\\pipe"` A communication pipe on the local host
- `"\\\\host\\dapname\\pipe"` A communication pipe on a remote host

Within the UNC string, the fields "host", "dapname", and "pipe" are replaced with the text for the actual host machine name, DAP board, and communication channel to be accessed. The host name is established when the host operating system is installed. The dapname field specifies one of the DAP board names assigned automatically by the operating system, in an undefined sequential order, at boot time. Valid DAP board names have the form

```
Dap0
Dap1
Dap2
```

etc.

The pipe field typically specifies one of the predefined communication pipe names:

```
$SysOut     $SysIn     $BinOut     $BinIn
```

However, it might also specify the name of a supplemental communication pipe added to the host system using the *Data Acquisition Processor* applet in the Control Panel:

```
$Cp2In      $Cp2Out     $Cp3In      $Cp3Out     etc.
```

The operating mode of the connection is specified by the *option* parameter. This is an enumerator code that can be one of the following:

DAPOPEN_READ	Open to receive data from the DAP, either binary or text
DAPOPEN_WRITE	Open to write data to the DAP, either binary or text
DAPOPEN_QUERY	Open to query about the state of the DAP

Raises a DapException if the open operation is not successful.

Otherwise, when successful, it returns an HDap handle object that has a nonzero .Value() property.

Examples

```
DH = DapOpenHandle("\\\\.\\Dap0\\Cp2Out",DAPOPEN_QUERY)
```

Open a handle to a specially-configured Cp2Out communication pipe for board Dap0, located in the local host. The Cp2Out communication pipe must be previously added to the DAP's system configuration for this purpose. Binds the returned handle object to identifier DH.

function: CloseDAPHandle

Close a previously-opened connection to a DAP board.

CloseDAPHandle(hd)

Parameters

<hd>
The name of an HDap handle object previously opened.

Returns

None

Library

daphandle.py

Exceptions

None

Description

A CloseDAPHandle utility function terminates the connection established when the HDap handle object *hd* was constructed and opened using some other function such as HDap.Open or OpenDAPHandle.

It is very important for every application to perform a close operation when it is done using any DAP communication handle, either using the CloseDAPHandle utility or some other function. Python memory management will clean up an HDap object when it is no longer required, but Python cannot know what is going on in the embedded environment of a DAP board to perform cleanup operations there. Failing to close the connection can leave the DAP board "locked" to a communication connection that no longer exists, or even to a host task that no longer exists, thus blocking future access to the DAP board.

Examples

```
DapCloseHandle( DH0 )
```

Close the connection to a communication pipe previously opened and associated with DAP handle object DH0.

function: Open1Dap

Open connections to all predefined communication pipes for a single DAP board on the local host.

Hlist = Open1Dap(script)

Parameters

<script>
A string specifying a path to a file containing DAPL configuration commands.

Returns

<hlist>
List of four `HDap` objects, opened and ready to use.

Library

```
daphandle.py
```

Exceptions

`DapException`

Description

An `Open1Dap` utility function opens communication pipe handles for all of the predefined communication pipes for board `'DAP0'` on the local host. This is a simplified alternative to using the **OpenDAPHandle** function explicitly for each individual communication channel.

In addition to setting up handles for communication, this command

- terminates any ongoing DAP board activity
- flushes any stale configurations or data from past history of operation
- loads a new configuration script to tell the DAP board what to do

The *script* parameter specifies a path and file name for the file that contains the DAPL configuration commands. See **section 3** of this manual for an introduction to the content of this text file.

The returned value is a list of `HDap` objects that have been created, initialized, and opened. To access an individual channel, it is necessary to extract the appropriate handle from the returned list. The array can be indexed using the following index notations to access the individual `HDap` objects.

`SysOut`	(index 0), for `$SysOut` communication pipe, opend for reading text messages
`SysIn`	(index 1), for `$SysIn` communication pipe, opened for writing text commands
`BinOut`	(index 2), for `$BinOut` communication pipe, opened for reading binary data
`BinIn`	(index 3), for `$BinIn` communication pipe, opened for writing binary data

The pipes are opened in the appropriate mode for reading or writing data. Observe that the naming conventions are from the point of view of the DAP board. Thus, for example, the `BinIn` pipe is for reading on the DAP side, but it is for writing data on the PC host side.

Examples

```
DHlist = Open1Dap( "C:\\\\AppPath\\DataAq\\script0.dap" )
HBinOut = DHlist[BinOut]
```

Open all of the predefined handles to communication pipes for DAP board `Dap0` on the local workstation. Receive a list of 4 opened handles. Initialize the DAP board configuration using the commands in the `script0.dap` file. The `$BinOut` communication pipe will be used extensively for transfers of large quantities of data, so the corresponding communication pipe handle is extracted from the returned list for more efficient access.

function: Close1Dap

Close1Dap(dhlist)

Close all previously opened connections to predefined communication pipes to a single DAP on the local host.

Parameters

<dhlist>
The list of opened DAP handles as previously set up by an **Open1Dap** function.

Returns

None

Library

`daphandle.py`

Exceptions

None

Description

A `Close1Dap` utility function closes all of the communication pipe connections previously opened by an `Open1Dap` function. The *dhlist* parameter is the list of opened communication pipe hands as it was initially built. Any activity currently running on the board is terminated, and stale configurations, if any exist, are cleared from memory.

It is very important for every application to perform a close operation on each connection before terminating the application. The `CloseDAPHandle` utility could be called for each individual `HDap` object in the handle list, but the `Close1DAP` function closes the set of handles in a single operation. Failing to close the connection to any communication pipe can leave the DAP board "locked" to a communication connection that no longer exists, or even to a host task that no longer exists, thus blocking future access to the DAP board. Python memory management will clean up `HDap` objects when no longer required, but Python cannot know what is going on in the embedded environment of a DAP board, so it can't perform its clean-up operations there.

Examples

```
Close1Dap( DHL )
```

Terminate all activity and close the communication pipes in the list `DHL` previously produced by the `Open1Dap` function for accessing DAP board `Dap0` on the local workstation.

function: Open2Daps

hlist2 = Open2Daps(script0, script1)

Open up connections to all predefined communication pipes for two DAP boards on the local host.

Parameters

<script0>
A string specifying a path to a file containing DAPL configuration commands for `Dap0`.

<script1>
A string specifying a path to a file containing DAPL configuration commands for `Dap1`.

Returns

<hlist2>
List containing two elements, each of which is a list of four `HDap` objects, opened and ready to use.

Library

`daphandle.py`

Exceptions

`DapException`

Description

An `Open2Dap` utility function is like a combination of two `Open1Dap` utility functions, one for preparing `Dap0`, the other for preparing `Dap1`, for a system that has two DAP boards, `Dap0` and `Dap1.` Because there are two DAP boards, the returned list has two elements. Because there four communication handles predefined for each DAP board, each of the returned elements is a list containing four opened handles. The details of this operation are otherwise the same as for the `Open1Dap` utility function.

This function can be helpful to avoid some of the complications of operating two DAP boards in a master/slave configuration. When DAP boards are linked together with a special synchronization cable, and configured in the scripts for master/slave operation, there are some restrictions about the sequence of events that must occur to initialize them. Specifically:

1. The slave DAP, `Dap1`, must be initialized and started before the master DAP, `Dap0`.
2. Then, the master DAP,.`Dap0`, must be initialized and started.

If this sequence is not observed, it is possible for the `Dap0` master board to start sending hardware timing signals to the `Dap1` slave board before the slave board's hardware is expecting them, with undefined results. Using the `Open2Dap` utility will assist in getting this sequence right every time. However, for an extra measure of safety, add the following line to the end of the `Dap0` **master** board configuration script before the `START` command there:

```
PAUSE 100
```

This allows the slave board an extra 1/10 second of time to get its configuration ready before the master board starts operation.

Examples

```
DHL = Open2Daps( "C:\\\\DataAq\\script0.dap", \
  C:\\\\DataAq\\script1.dap )
HB0 = (DHL[0])[BinOut]
HB1 = (DHL[1])[BinOut]
```

Open all of the predefined handles to operate two DAP boards, `Dap0` and `Dap1` on the local workstation. Receive a list with two sets of 4 opened handles, the first for `Dap0` and the second for `Dap1.` With the two boards operating simultaneously, data will arrive at a high rate rate over the two `$BinOut` communication channels, so the handles for the `$BinOut` communication pipes from each board are selected.

function: Close2Daps

Close2Daps(dhlist2)

Close the connections to all predefined communication pipes for two DAP boards on the local host.

Parameters

<dhlist2>
The list with two sets of opened DAP handles previously set up by the `Open2Daps` function.

Returns

None

Library

`daphandle.py`

Exceptions

None

Description

A **Close2Dap** utility function closes all of the communication pipe connections previously opened by an **Open2Daps** function. The *dhlist2* parameter is the list containing two sub-lists of opened communication pipe handles, as originally built by the **Open2Daps** function. Any activity currently running on the boards is terminated, and configuration information, if any, is cleared from memory.

It is very important for every application to perform a close operation on each connection before terminating the application. The `CloseDAPHandle` utility could be called for each individual `HDap` object in the handle lists, but the `Close2DAP` function closes both set of handles in a single operation. Failing to close the connection to any communication pipe can leave the DAP board "locked" to a communication connection that no longer exists, or even to a host task that no longer exists, thus blocking future access to the DAP board. Python memory management will clean up `HDap` objects when no longer required, but Python cannot know what is going on in the embedded environment of a DAP board, so it can't perform its clean-up operations there.

Examples

```
Close2Daps( DHL )
```

Terminate all activity for DAP boards `Dap0` and `Dap1` on the local host, clear all unused data, remove all configuration information, and close all communication pipe connections in the two lists contained in `DHL,` which was previously produced by the **Open2Daps** function.

6. Simple data transfers

After you have successfully opened a `HDap` handle object for reading data through a communication pipe, and configured your DAP board to send data, you will want to receive the data into your application. Similarly, after you have successfully opened a `HDap` handle object for writing data, and configured your DAP board with processing that expects to receive it, you will want to send the data to the board from the host.

Simple single-value transfers are covered in this section.

Caveat: Efficiency for processing data one value at at time is poor. This is something that you should do only when efficiency is not an issue.

Suppose, for example, that an activity is entirely managed by a DAP board. How can your application know whether the DAP continues to operate normally, or has encountered a problem and prematurely shut down?

You might try a "watchdog" strategy. Suppose that the DAP is processing 100000 samples per second. In parallel with other processing, reduce this data to a steady one sample every ½ second. Transfer these through the predefined `$BinOut` communication pipe. Now arrange for your Python application to request all of the available data from the communication pipe at approximately one-second intervals. Depending on small variations in the timing, you might receive one, two, or three samples. You don't care the exact number, or what the values are, as long as something is present. However, if you receive no samples at all, that means serious problems, and some kind of exception condition should be raised.

With so few samples involved, efficiency makes little difference. In fact, trying to use an elaborate and highly optimized data transfer configuration might make things worse.

Single-value transfers are so simple that there is no need for special utility functions to perform them. You can call the raw DAPIO access functions directly. In all cases, it is assumed that the appropriate DAP handle object has been set up and opened for the kind of transfer you intend to make. Let the name `DH` represent this handle in examples below.

Is there anything to see?

This can be a very tricky question. Suppose that you ask the DAP to send you a number, but no number available. What is the DAP supposed to do? Should it reports back immediately that it has no number? Just at the instant that it does this, the number could appear from its other processing tasks, rendering the information false. But if the DAP waits for the requested value to arrive, the wait time could be indeterminate. There is no simple solution to this problem.

One thing that you can do to defend against this kind of situation is to not ask for data that do not yet exist. You can use the `InputAvail` method of the DAP handle object to test whether there is anything available to fetch.

```
test = DH.InputAvail( )
```

If the returned test variable is nonzero, there is something to read. If the returned test variable is zero, there is nothing available. You can do other things and try again later, report an "empty value," or treat the situation as an error .

Some methods provide a timeout parameter. This usually makes things easier, because you don't need to poll in advance to see if there are any data available. Just allow reasonable time, and if the result code indicates that nothing was transferred, take appropriate corrective action.

What is the data type of a transfer?

In the Python environment, you don't have to think about this much. Numbers are numbers. But it makes a difference to the DAP board – if it expects a number in an IEEE 32-bit floating point representation, nothing else will do. Select the appropriate DAPIO methods or utility functions that convert the data to the data type that the DAP expects.

Use a communication pipe that is configured for correct type of data and the correct transfer direction – DAP to host, or host to DAP. The `HDap` handle object to be used for the transfer is called `DH` in the examples below.

Transferring single characters

- Send a character to the DAP board. This expects an 8-bit encoding, so do this as follows.
```
CharVal='A'
DH.CharPut(CharVal.encode())
```

- Receive a character value
```
if  DH.InputAvailal( ) > 0 :
  InChar = DH.CharGet( )
else :
  InChar = 0
```

Transferring 16-bit integer sample values

- Send a 16-bit integer value to the DAP board.
```
IntVal=32700
DH.Int16Put(IntVal)
```

- Receive a 16-bit integer value from the DAP board
```
if  DH.InputAvailal( ) > 0 :
 InInt = DH.Int16Get( )
else :
 InInt = 0
```

Transferring 32-bit integer values

- Send a 32-bit integer value to the DAP board.
```
IntVal=3270000
DH.Int32Put(IntVal)
```

- Receive a 32-bit integer value from the DAP board
```
if  DH.InputAvailal( ) > 0:
 InInt = DH.Int32Get( )
else :
 InInt = 0
```

Transferring floating point values

In addition to transferring a floating point number value, the transfer method converts the representation to or from the DAP board's floating point representation, either 32-bit or 64-bit IEEE floating point, to the number representation used in the Python environment.

Transfers from the DAP provide a timeout parameter, so in most cases it is not necessary to check for available data before attempting to receive it.

- Send a 32-bit floating point value to the DAP board.
```
FloatVal=32700.00
DH.FloatPut(IntVal)
```

- Receive a 32-bit floating point value as a Python number, waiting no more than 1/10 second
```
(result,Val) = DH.FloatGet(100)
if result = 0 :
  raise DapException("No float value received")
```

- Send a 64-bit floating point value to the DAP board.
```
DblVal=32700.00
DH.DoublePut(DblVal)
```

- Receive a 64-bit floating point value as a Python number, waiting no more than 1/10 second
```
(result,value) = DH.DoubleGet(100)
if result > 0 :
  raise DapException("No double value received")
```

method: HDap.Get

Receive a single transferred value into the host from a communication pipe.

Result = HD.CharGet( )

Result = HD.Int16Get( )

Result = HD.Int32Get( )

Result = HD.FloatGet(timeout)

Result = HD.DoubleGet(timeout)

Parameters

<HD>
A previously opened communication handle for a pipe sending data to the host.

<timeout>
When available, a timout interval for returning a value, non-negative, in units of milliseconds.

Returns

<Result>
A list with a result code and data value.

Library

`dapio.py`

Exceptions

None

Description

These are closely-related communication methods for an `HDap` object. These methods are a lightweight means to receive values one at a time, at a slow rate, from the DAP.

The DAP handle object *<HD>* must be a handle for a communication stream previously opened for transferring the appropriate data type from the DAP to the host application. If opened with a generic function like `Open1Dap`, the handle must be extracted from the list of open communication handles.

The method returns a list of two values. The first is an error code. If it is greater than 0, the call was successful and the second value is the one received from the DAP board. If the code is 0 or negative, the call was unsuccessful, and the second value has no meaning.

The function variant to choose depends on the data type that the DAP is configured to send. The returned value is the sort of native value that Python normally uses.

`Int16`	returns a Python number
`Int32`	returns a Python number
`Float`	returns a Python number
`Double`	returns a Python number
`Char`	returns a one-character Python unicode string

For the methods that support a timeout function, the timeout interval <*timeout*> must be a non-negative integer number in units of milliseconds. If this much time elapses, and no value becomes available, the call returns and gives a 0 result code. (The unit of time resolution, milliseconds, is different from the one-second resolution units used by the Python `time` library.)

Examples

```
result = HD.Int16Get( DHL[2] )
if  result[0] == 0
  raise DapException("No DAP value received")
val = result[1]
```

Request the next 16-bit processed sample value sent from the DAP through the `$BinOut` communication pipe associated with the opened `HD` handle object. If the value is received successfully, extract it from the returned list.

method: HDap.Put

Send a single transferred value from the host to the DAP through the associated communication pipe.

Result = HD.CharPut(val)

Result = HD.Int16Put(val)

Result = HD.Int32Put(val)

Result = HD.FloatPut(val)

Result = HD.DoublePut(val)

Parameters

<*HD*>
A previously opened communication handle for a pipe sending data to the host.

<*timeout*>
When available, a timeout interval for return of value, non-negative, in units of milliseconds.

Returns

<*Result*>
Result code

Library

`dapio.py`

Exceptions

None

Description

This is a closely-related set of communication methods for an `HDap` object. These methods are a lightweight means to send individual values one at a time at a slow rate from the DAP.

The DAP handle object <*HD*> must be a handle for a communication pipe previously opened for the appropriate data type and transfer direction. If opened with a generic function like Open1Dap, the handle must be extracted from the list of open communication handles.

The methods expect that the DAP processing is operating normally and taking all data transferred through the pipe in a timely manner. If there is any possibility that this processing could be delayed, check for available buffer storage first before using this method.

The function variant to choose depends on the data type that the DAP expects to receive. The value is converted from the value that Python natively uses.

`Int16`	converted from a Python number
`Int32`	converted from a Python number
`Float`	converted from a Python number
`Double`	converted from a Python number
`Char`	converted to an ASCII character from one Python Unicode character

The returned result code will be greater than 0 if the operation was successful, or 0 if it failed. Success only means that the data went into the pipe; the exact time at which the DAPL system tasks will be scheduled to receive and act on that data is not predictable.

Examples

```
result = HD.FloatPut( val )
if  result == 0
  raise DapException("Value transfer to DAP failed")
```

Convert the number val to a 32-bit floating point number and send it to the DAP through the opened communication pipe associated with handle `HD`. Raise an exception if the pipe cannot accept the value.

7. Bulk data transfers

For transferring large amounts of data between the host application program and the DAP board, it is imperative to send the data in relatively large blocks. A typical range is *250 to 4000 values* per block. The cost of a transfer operation is relatively high, but the cost of processing one sample within that transfer is almost negligible. For example: if you can transfer 1000 values at a time, your data transfers approach 1000 times more efficient. The only down-side to this is that there are longer "latency" delays for assembling the larger blocks data blocks.

There are two kinds of block data transfers.

- **Simple transfers.** These are easy to apply, but they give you limited configuration options, and they have some limitations. For example, if you request more values than the DAP is configured to send, your application can *hang* for a relatively uncomfortable length of time before the impasse is detected.

- **Configurable ("extended") transfers.** These take more thought and effort to set up, but if the application will be used a lot, the incremental performance improvements are worth the effort.

Transfers can be managed at a higher or lower level.

- **Utility level.** These higher-level functions are easier for accessing data immediately, because results are converted to familiar Python objects.

- **ctypes level. transfers.** Lower-level access functions of the HDap object involve less overhead, and might be useful for optimizing operations where data post-processing is deferred to later.

Setting up data buffer storage

Allocating bulk storage is a somewhat foreign idea in Python. Typically, you to construct a data object and then bind a name to it. You don't worry about memory allocation or the internal details about how the storage is organized. For purposes of DAP access, data types and layouts must be as the DAP expects.

The recommended way to provide bulk storage is to use the `array` library and the `dapblock` library.

```
from array  import *
from dapblock  import *
```

Next, you need to set up the data you wish to transfer as a Python array. Python arrays are restricted to contents of a single data type, but this allows a compact memory representation. When you construct the array, you specify a one-character code to select the data type. The supported type codes are

`'h'`	short integer (sample value)
`'i'`	long intger
`'f'`	32-bit float
`'d'`	64-bit float

If you are receiving data, the transfer operation provides the data in the appropriate format. If you are sending data, you can use the `append` method of the array to insert the values into storage, or you can import the data from some other more general Python structure like a list.

```
block1 = array('h')
block2 = array('h',[1, 2, 3, 4])
block1.append( 5 )
```

Note: if you choose to use the DAPIO methods directly, you will need to use C buffer objects that store the following data types:

`c_short`	storage for 16-bit integers
`c_long`	storage for 32-bit integers
`c_float`	storage for 32-bit floating point
`c_double`	storage for 64-bit floating point

Simple data block transfer from DAP to host

You can use the `SendDAPBlock` utility function to do a simple block transfer from the DAP to the host application. First, configure your DAP processing to send the output data stream. Typically, you will do this through the `$BinOut` pipe. Suppose that the `HDap` object `HRecv` was set up for this pipe.

Next, call the **GetDAPBlock** utility function and give it the previously opened handle. Specify the number of values to be received, and the data type code for the kind of data to expect from the DAP board.

```
print('Receive a block of 100 sample values from 2 channels')
result = GetDAPBLock( dhlist[BinOut], 200, 'h' )
```

In return, you will receive a list of two items. The first item is the number of values actually returned. In the event that the operation times out (this could take a *very* long time) the count could be different from the number you specified. The second item in the list is the array object containing the data. (You can verify size, data type, and so forth from the properties of that object if you wish.)

```
nvals = result[0]
dat200 = result[1]
print("The array containing ",nvals," results: ", dat200)
```

If you choose to use the DAPIO `GetBuf` method directly, you must set up a `ctypes` buffer object for the appropriate data type. The DAPIO transfers are always performed using data units of bytes, so the counts of the number of terms received must be converted according to the size of the items.

Simple data block transfer to host to DAP

Binary data transfers are typically used to send raw signal data for the DAP board to generate an analog signal output, but they are also used to send configuration and control data for special processing. Typically, data are sent through the `$BinIn` predefined communication pipe. Select a transfer pipe that is configured for the type of data you want the DAP to receive, and make sure the connection is opened.

Set up a Python array object, specifying the data type code for the type of data the DAP is configured to expect. The size of the transfer block will be the size of this array that you build. For example, you can declare an array for 16-bit integer sample values.

```
ivalues = array('h')
for i in range(nvals) :
  ivalues.append( sigval[i] )
```

To perform the transfer, call the **SendDAPBlock** utility function. Specify as arguments the handle to the communication pipe to receive the data, and specify the source array. The size of the transfer will be deduced automatically from the array object.

```
nbytes = SendDAPBlock(dhlist[BinIn], ivalues)
```

The returned value is the number of bytes of data sent during the transfer operation. Usually, anything other than zero means success, but if the channel is full, a partial transfer is possible.

Configurable data block transfer from DAP to host

To set up a configurable block transfer to the host, you need to define an additional transfer configuration object. Look in the DAPIO32 Manual for complete information about this object.

```
incnf = TDapBufGetX()
```

Now you must decide on the configuration, and initialize the fields in this control structure accordingly. For example, suppose that you will accept any number of available 16-bit sample values – as few as one, as many as 1000. The `BytesMultiple` property specifies the number of bytes in each item, which for 16-bit integer sample data is 2. Other items are also expressed in bytes, so each item includes a factor of 2.

```
incnf.BytesMultiple = 2
incnf.BytesGetMin   = 1*2
incnf.BytesGetMax   = 1000*2
```

Adjust the timing constraint fields. The simplest way is to set the maximum inactive time and the maximum time to complete the transfer operation the same, expressed in milliseconds. This is at least a reasonable starting configuration. In this example, the data stream is fast, so if a data block fails to arrive sooner than 1/10 second (100 milliseconds), something is seriously wrong.

```
incnf.GetWait = 100
incnf.Timeout = 100
```

The remaining fields are filled in automatically when the `TDapBufGetX` element is constructed. Keep it handy, since you will use it for each block transfer.

Now to receive the data, use the `GetDAPBlockEx` utility function. Provide the handle previously opened for accessing the pipe that transfers the data, typically the `$BinOut` predefined communication pipe. Also provide access to the special `TDapBufGetX` configuration object.

```
result = GetDAPBlockEx(HD, incnf)
nReceived = result[0]
indata = result[1]
```

Configurable data block transfer from host to DAP

To set up a configurable block transfer into the DAP, you need to define an additional transfer configuration object. This is similar to the case of a block transfer from the DAP, but the `TDapBufPutX` object used has slightly different properties.

```
outcnf = TDapBufPutX()
```

Adjust the configuration fields. For example, suppose that you are sending a block of 1000 32-bit floating point values. The `BytesMultiple` property specifies the number of bytes in each item, which for 32-bit floating point values is 4. Other items are also expressed in bytes, so each item includes a factor of 4.

```
outcnf.BytesMultiple = 4
outcnf.BytesPut   = 1000*4
```

Adjust the timing constraint fields. The simplest way is to set the maximum inactive time and the maximum total time to complete the transfer operation the same, expressed in milliseconds. For most applications, the data transfers are relatively small and infrequent, so any delay longer than 1/10 second (100 milliseconds) for the pipe to accept the data means something is seriously wrong.

```
outcnf.SendWait = 100
outcnf.Timeout   = 100
```

The `TDapBufPutX` object is ready. Keep it handy, since you will need it for each block transfer.

To send the data, use the `SendDAPBlockEx` utility function. Provide the handle previously opened for accessing the pipe that transfers the data, typically the `$BinIn` predefined communication pipe. Also provide access to the special `TDapBufPutX` configuration object.

```
nsent = SendDAPBlockEx(HD,outcnf)
```

The returned value is the number of bytes sent to the pipe during the transfer.

function: GetDAPBlock

Receive a block of binary data from the DAP to the host application through the specified communication pipe.

(num,values) = GetDAPBlock(HD, length, typecode)

(num,values) = GetDAPBlockEx(HD, config, typecode)

Parameters

<*HD*>
A previously opened communication handle for a pipe sending data out from the DAP.

<*length*>
The maximum number of values to receive from the DAP.

<*config*>
A `TDapBufGetEx` object specifying configuration information for the data transfer.

<*typecode*>
A one-character data type code for a Python `array` object, specifying the returned data type.

Returns

<*num*>
Number of items delivered in the returned data array storage.

<*values*>
Returned Python `array` object containing the data values.

Library

`dapio.py`

`dapblock.py`

Exceptions

`DapException`

Description

Receive a block of data sent by the DAP board to the application on the host system. There are two forms. For the simple transfer form, the number of items desired in the transfer block is specified. For the configurable transfer form, pass a special configuration object of type `TDapBufGetEx` that specifies the transfer characteristics.

For both forms, the `DapHandle` object <*HD*> must be a handle for a communication stream previously opened for transfers from the host system to the DAP board. If opened with a generic function like `Open1Dap`, the handle must be extracted from the list of open communication handles. For both forms, a one-character codes specifies the data type that is expected from the DAP. These codes are:

`'h'`	short integer (sample value)
`'i'`	long intger
`'f'`	32-bit float
`'d'`	64-bit float

The returned value is a list containing two elements. The first element *<num>* is the integer number of values returned in the memory of the Python `array` object. The second element *<values>* is the `array` object. Properties of the data elements and the array can be polled from the `array` object.

If the DAP communication pipe is unable to deliver the specified full number of values, the returned value is the number of values actually delivered into the pipe. Typically, the returned count will be greater than 0 if the operation is successful, or 0 if no data could be transferred. However, beware that if the transfers are delayed for some reason, the full block of data might not arrive.

Examples

```
result = GetDAPBlock( Hbinout, 100, 'h' )
count = result[0]
values = result[1]
if  count == 0
  raise DapException("DAP sent no data")
```

Perform a simple block transfer from the DAP to the host application. The desired block size is 100 terms of 16-bit integer values. The Dap Handle object `Hbinout` is open and ready for transfers from the DAP to the host. The number of terms in the received array, and the `array` object containing the data, are returned. If nothing is received, raise an exception.

```
cfg = TDapPutBufEx
...
result = GetDAPBlockEx( Hbinout, cfg, 'f')
values = result[1]
```

Perform a configurable block transfer from the DAP to the host application. The desired block size and timing parameters are sent using the `cfg` object. The data to be received are 32-bit floating point values. The Dap Handle object `Hbinout` is open and ready for transfers from the DAP to the host. The `array` object `values` returns the data.

function: SendDAPBlock

Send a block of binary data from the host system to the DAP through the specified communication pipe.

num = SendDAPBlock(HD, values)

num = SendDAPBlockEx(HD, config, values)

Parameters

<*HD*>
A previously opened communication handle for a pipe sending data to the DAP.

<*values*>
A Python `array` object that represents the data to transfer in the appropriate data type.

<*config*>
A `TDapBufPutEx` object specifying configuration information for the data transfer.

Returns

<*num*>
Number of bytes actually delivered to the transfer pipe.

Library

`dapio.py`

`dapblock.py`

Exceptions

`DapException`

Description

Send a block of data sent to the DAP board from the host system. There are two forms. For both forms, the `DapHandle` object <*HD*> must be a handle for a communication stream previously opened for transfers from the host system to the DAP board. If opened with a generic function like `Open1Dap`, the handle must be extracted from the list of open communication handles. For both forms, the data source <*values*> is a Python array object containing data of the appropriate data type. For the simple transfer form, the number of bytes to transfer is determined from the size of the provided data array. For the configurable transfer form, provide an additional configuration object <*config*> of type `TDapBufPutEx` .

If the DAP communication pipe is unable to accept the specified number of values, the returned value is the number of bytes actually delivered into the pipe. Typically, the returned count will be greater than 0 if the operation is successful, or 0 if data could not be transferred. However, beware that if transfers are delayed for some reason, the transfer can be partial. A later attempt might be needed to deliver the rest of the data.

Examples

```
WDarray = array('h')
...
result = SendDAPBlock( Hbinin, WDarray )
if  result == 0
  raise DapException("DAP sent no data")
```

Perform a simple block transfer to the DAP. The data are set up in a Python array `WDarray` of 16-bit integer data. The DAP Handle object `Hbinin` is open and ready for transfers from the host to the DAP. The number of terms to send is determined from the number of terms in the array. Raise an exception if the pipe is backlogged with data and cannot accept more.

```
cfg = TDapPutBufEx
WDarray = array('h')
...
result = SendDAPBlockEx( Hbinin, cfg, WDarray )
if  result == 0
  raise DapException("DAP sent no data")
WDarray = result[1]
```

Perform a configurable block transfer from the DAP. The data are set up in a Python array `WDarray` of 16-bit integer data. The Dap Handle object `Hbinin` is open and ready for transfers from the host to the DAP. The `TDapBufPutEx` configuration object `cfg` is set up and configured with the desired transfer properties. Raise an exception if the pipe is backlogged with data and cannot accept more.

method: HDap.BufferGet

Receive a block of data sent from the DAP to the host system through the associated communication pipe.

Num = HD.BufferGet(max, storage)

Num = HD.BufferGetEx(config, storage)

Parameters

<HD>
A previously opened communication handle for a pipe sending data to the host.

<max>
Maximum number of bytes that can be received (storage capacity).

<config>
A `TDapGetEx` object specifying configuration information for the data transfer.

<storage>
Previously configure `ctypes` buffer area for returning the data.

Returns

<Num>
Number of bytes actually received and placed into storage.

Library

`dapio.py`

Exceptions

None

Description

This is a lower-level method to receive a block of data sent by the DAP to the host system. This has somewhat less Python overhead, so it might be useful to optimize some critical data transfer operations; however, it is harder to use.

There are two forms. For both forms, construct a `ctypes` storage area where the transferred values are placed. For the simple transfer form, you specify the number of bytes *<max>* that should be transferred from the DAP into the reserved storage. For the configurable transfer form, set up a special configuration object of type `TDapBufGetEx` and pass this instead of a fixed byte count.

The `DapHandle` object <*HD*> must be a handle for a communication stream previously opened. If opened with a generic function like `Open1Dap`, the handle must be extracted from the list of open communication handles.

If the DAP is unable to provide the specified number of values, the returned value is the number of bytes actually returned and placed into storage. Typically, the returned count will be greater than 0 if the operation is successful, or 0 if no data could be transferred. However, beware that if the transfers are delayed for some reason, the full block of data might not arrive.

Examples

```
from ctypes import *
storage = (c_short * 1024)()
result = HD.BufferGet( 1024*2,storage )
if  result == 0
  raise DapException("DAP sent no data")
```

Perform a simple block transfer from the DAP. Expect exactly 1024 16-bit integer sample values, each 2 bytes in length. Raise an exception if no data are found. (This could delay a long time.)

method: HDap.BufferPut

Send a block of data from the host system to the DAP processing through the associated communication pipe.

sent = HD.BufferPut(num, storage)

sent = HD.BufferPutEx(config, storage)

Parameters

<*HD*>
A previously opened communication handle for a pipe sending data to the DAP.

<*num*>
The number of bytes to be sent (storage capacity containing data).

<*config*>
A `TDapBufPutEx` object specifying configuration information for the data transfer.

<*storage*>
Previously configure `ctypes` buffer area providing the data to send.

Returns

<*sent*>
Number of bytes actually delivered for transfer.

Library

`dapio.py`

Exceptions

None

Description

This is a lower-level method to send a block of data from the host system to a DAP. This has somewhat less Python overhead, so it might be useful to optimize some critical data transfer operations; however, it is harder to use.

Send a block of data to the DAP from the host system. The `DapHandle` object <*HD*> must be a handle for a communication stream previously opened for sending data into the DAP. If opened with a generic function like Open1Dap, the handle must be extracted from the list of open communication handles.

There are two forms. For the simple transfer form, you specify the exact number of bytes that should be transferred from the host into the DAP processing from the reserved storage. For the configurable transfer form, set up a special configuration object of type `TDapBufPutEx` and pass this instead of a fixed byte count.

The returned value is the number of bytes actually delivered to the pipe buffers. Typically, the returned count will equal the number specified. However, if the communication pipe cannot accept all of the provided data for delivery, for whatever reason, beware that the returned value might be lower. If the pipe is completely full and no data can be taken, the return value will be 0.

Examples

```
from ctypes import *
storage = (c_short * 1000)()
...
result = HD.BufferPut( 500*2,storage )
if  result == 0
  raise DapException("DAP accepting no data")
```

Perform a simple block transfer from the DAP. Only the first 500 of the available 1000 locations of buffer storage contain data for 16-bit integer sample values, each 2 bytes in length. Raise an exception if the DAP cannot accept tany data. (This simplified form could delay a long time.)

8. Text and command transfers

In Python culture, text is stored in string objects with Unicode, while for DAPIO, text is primitive ASCII character sequences with control character terminations. Translating back and forth is an ongoing process, but utility functions in the `daptext.py` module can help.

To access the `daptext.py` functions:

```
from daptext  import *
```

When using DAPIO methods directly through the DAP Handle object, you will make frequent use of the Python `encode()` and `decode()` methods for string objects. The `encode` method will convert the string content to a compatible ASCII character form. The `decode` method will convert to a Python internal representation from various other kinds.

```
strtext = "ABCD"
asciitext = strtext.encode()

strtext = asciitext.decode()
```

Sending configuration and command texts

Rather than embedding DAP configuration details into application code, you can place the configuration commands (see section 3) in a separate text file and move them to the DAP board all at once. Most applications will want to do this.

If you use the `Open1Dap` utility function to establish your communication pipe connections, this functions expects you to provide a configuration script file. You simply specify the path to this script at the same time that you open connections, for example:

```
result = Open1Dap( "C:\\\\AppFolder\\ThisApp\\config.dap" )
```

For applications that need more configuration control, you can send the configuration file separately, using the `DapSendConfig` utility function. This function applies the `Config` method provided by the DAP handle, but also takes care of the text conversions automatically. For example:

```
# Location of the DAP configuration script file
filepath = "C:\\\\AppFolder\\ThisApp\\config.dap"
# Send the configuration file
DHsysin = DapHandleOpen( "\\\\.\\Dap0\\$SysIn", DAPOPEN_WRITE )
result = DapSendConfig(DHsysin,filepath)
```

An alternative is to use the DAPIO `Config` method directly, instead of the utility function. When you do this, you must convert the text to ASCII form yourself.

```
# Get a handle for downloading text to DAP
result = DHsysin.Config( path.encode() )
```

Some applications require configurations that are constructed dynamically. (This strategy is harder than it first appears, and can interfere with application validation and debugging, but you do what you must.) For these cases:

- One strategy is to build the script dynamically as lines in a memory-resident "*file-like object*", and move this to actual file storage temporarily for DAPIO functions to load. (Unfortunately, DAPIO does not know about "*file-like objects*" so it can't load them from Python memory directly.)

- Another strategy is to deliver the script interactively, line by line, under software control.

It does not matter how the lines arrive, singly or in groups, via configuration files or a line at a time. You can mix the two strategies. The only thing that matters is that all of the necessary lines are presented in the proper sequence .

A very common example of a mixed stratregy is to send all of the configuration lines first from a file, but omit the `START` command that would start DAP operation. After the rest of the application is set up, at some later time the DAP processing can be started quickly with one additional "`START`" command line.

Here is a complete DAPL configuration for streaming two channels of sampled data through the `$BinOut` communication pipe, but with a configurable sample time interval. In this example, all commands are sent line-by-line, using a previously-opened DAP handle object `HDsysin`. The format method of the Python string object supplies the substituted sampling time interval value.

```
timespec = 100
SendDAPCommand(HDsysin,"IDEFINE SAMP2")
SendDAPCommand(HDsysin,"  CHANNELS 2")
SendDAPCommand(HDsysin,"  SET  IPipe0  D0")
SendDAPCommand(HDsysin,"  SET  IPipe1  D1")
SendDAPCommand(HDsysin,"  TIME  {1}".format(timespec)")
SendDAPCommand(HDsysin,"END")
SendDAPCommand(HDsysin,"PDEFINE SEND2")
SendDAPCommand(HDsysin,"  COPY(IPipes(0,1),$BinOut)")
SendDAPCommand(HDsysin,"END")
```

This configuration doesn't start until one additional "`START`" command line is sent later.

```
DapSendCommand(HDsysin,"START")
```

An alternative to using the `SendDAPCommand` utility function is the `LinePut` method of the DAP Handle object.

```
cmdline = "START"
HDsysin.LinePut(cmdline.encode())
```

Receiving text messages from DAP

The DAP sends messages as ASCII text lines that are terminated with "`CR`" or "`CR LF`" control characters. Both ways are unnatural to Python. The DAPIO methods and utility functions attempt to make this a little easier by detecting the termination characters and converting all of the DAPIO text content to Python-friendly strings.

The command interpreter task of the DAPL system sends messages through the `$SysOut` predfined text communication pipe. DAPL configuration options (the `OPTION` command) configure this behavior.

- Diagnostics in response to configuration errors
- Responses requested by certain commands
- Text displays formatted at runtime

Your main tool for retrieving text lines is the DAPIO method `LineGet()`. For example:

```
iavail = DHsysout.InputAvail( )
print("Currently ", iavail, "text characters are available" )
if  iavail > 0 :
  result = DHsysout.LineGet( 100 )
  print( result )
```

In this example it is determined that there is a message present in the DAPIO buffers, arrived from the `$SysOut` communication pipe, and the `LineGet` method should retrieve it, allowing no more than 1/10 second delay. The `LineGet` method takes care of the translations in to Python-style Unicode.

Error checking

The DAPIO operations return an indication of whether they were successful. Unfortunately, they can't do this in a completely uniform way (like 1 for success, 0 for failure, because those values might have a meaning as data). To get further information about host system failures when a DAPIO operation indicates a problem, you can use the `LastHostErrorGet` method. It returns a result code and a messge text. You can use any opened DAP communication pipe handle for this.

```
result = DHsysout.LastHostErrorGet( )
if  result[0] > 0 :
  print("DAPIO error detected:  ",result[1])
```

However, this is not the complete story. This only tells you about problems on the host system side. For example, the download of a DAPL configuration file might go without a hitch as far as the DAPIO communications are concerned, but problems happen later when the DAPL system tries to process the commands and finds a command syntax error. One useful thing that you can do – and should – is to verify successful processing of the DAPL configuration file. You can do this by asking the DAPL system to report the first error condition that it detected, as follows:

```
SendDAPCommand(HDsysin,"DISPLAY EMSG")
result = GetDAPText( 100 )
if  result[0] > 0 :
  print("DAP configuration error:  ",result[1])
```

If the error message text is empty, there was no DAPL processing error.

Advanced topics

The DAPIO interface has a long-established behavior. As it receives text from the DAP board, it counts all characters received as it stores them, *including* all of the control characters used to block the text into messages. As it later isolates and delivers messages into user-provided buffer storage, it sends on only the text content characters. Sooner or later, it purges all of the control characters from the original transport buffers. You will never see these in

the count of characters the application receives, nor in the text the application receives, so for the most part, no harm done.

However, an inconsistency arises when using the Dap Handle object `InputAvail` method. `InputAvail` looks ahead into the hidden data transport buffers, and sees the count of all characters available there, including control characters. The number of available characters reported will not be equal to the number of text characters the application will eventually receive. Because of this discrepancy, the application might incorrectly think that it has an additional message text pending. This could cause an application to waste processing time seeking a message text that does not exist.

Some ways to avoid problems of this sort:

1. Avoid transferring "empty message" texts.

2. Use the `GetDAPText` utility function rather than low-level methods to retrieve the text. This is slightly less efficient, but that should make little difference, since text is already inefficient. The utility does some extra checking to purge any additional control characters from the input buffer storage if they happen to be present. It is then safe to interpret any non-zero returned value from the `InputAvail` method to mean: "*You have another line of text available.*"

3. When using the Dap Buffer object `LineGet` method directly, interpret a return value of 1 from the `InputAvail` method to mean: "*The next text line has no useful data.*" You don't need to take any action.

function: SendDAPConfig

Send a batch of command text lines to the DAPL command interpreter from a text file.

result = SendDAPConfig(HD , path)

Parameters

<HD>
A previously opened communication handle for sending command text to the DAP.

<path>
A Python string specifying a path to a text file containing the DAPL configuration commands.

Returns

<result>
1 if the operation was successful, 0 if the operation failed.

Library

`daptext.py`

Exceptions

DapException

Description

Send a collected batch of command lines from an ASCII text file specified in string object *<path>* from the host system to the DAP, via the communication pipe associated with the *<HD>* parameter. The communication pipe must be previously opened and configured for text transfers into the DAP.

The download could fail for a host-related or DAP-related reasons.

- If the return value is 0, the processing on the host side failed – for example, the file does not exist. You can use the `LastHostErrorGet` method of the DAP handle object to get additional diagnostics from the host system.
- If the return value is 1, the transfer was successful, but processing of the configuration commands could have encountered errors. Send the "`DISPLAY EMSG`" command to the DAPL system to ask for the text of any configuration error that might have occurred.

Examples

```
from daptext import *
...
SendDAPConfig( DHstdin, "C:\\\\AppFolder\\App1\\config.dap" )
```

The `DHstdin` pipe is previously opened as a connection to the `$SysIn` command pipe to the DAPL system. Send the DAPL system a message to load the contents of the `config.dap` text file as configuration commands.

runction: GetDAPText

Get a line of message text sent from a DAP to the host system.

(num, text) = GetDAPText(HD)

Parameters

<*HD*>
A previously opened communication handle for reading text from the DAP.

Returns

<*num*>
Number of bytes in the buffered message text.

<*text*>
The contents of the message text.

Library

`daptext.py`

Exceptions

DapException

Description

Receive a line of text sent from the DAPL system to the application on the host system via the pipe specified by the <*HD*> parameter. The communication pipe must be previously opened and configured for transfer of text from the DAP to the host system. The received message text is converted to a Python string object.

Text from the DAPL system consists of ASCII characters delimited by carriage-return control characters or sometimes carriage-return and linefeed control character pairs. Each call to `GetDAPText` selects the characters from the next text line received by the DAPIO system, up to but not including the carriage-return delimiter character, copying the characters to the returned string. If any linefeed control characters are encountered, they are removed and ignored, neither counted nor placed in the returned text. The return value is a list with two fields, a <*num*> field reporting how many characters are in the returned line, and a <***text***> field reporting the contents of the message.

It is possible for the DAP to send message texts with no content. You could get this, for example, if you asked the DAPL system to `DISPLAY PIPES`, but the current DAPL configuration did not define any pipe elements. For cases like this, the <*num*> field will report a character count 0, and the <***text***> field will report a zero-length string.

Text message transfers are relatively inefficient. To avoid unnecessary attempts to received text data when there is nothing to receive, the DAP handle object's `InputAvail` method can be used to determine in advance whether any text is available to receive.

> *Note:* In addition to retrieving the message text, the `GetDAPText` function also performs some automatic clean-ups to remove residual control characters from the input pipe. If this is of no to concern to your application, an alternative is to invoke the DAP handle's `LineGet` method directly.

Examples

```
From time import sleep
from daptext import *
...
SendDAPCommand( DHstdin, "HELLO" )
sleep(0.1)
result = GetDAPText( DHstdout )
if  result[0] == 0 :
  print("DAP is not responsing!")
  raise DapException
else :
  textline = result[1]
```

Verify that the DAPL system is available and responsive by sending it a "`HELLO`" command text line and then using the `GetDAPText` function to look for the returned response message.

Function: SendDAPText

Send a line of command text from the application to the DAPL system command interpreter.

result = SendDAPCommand(HD , command)

result = SendDAPText(HD , text)

Parameters

<HD>
A previously opened communication handle for reading text from the DAP.

<text>
The contents of the message text to send as a Python string.

<command>
The contents of the message text to send as a DAPL system command.

Returns

<result>
1 if the operation was successful, 0 if the operation failed.

Library

`daptext.py`

Exceptions

DapException

Description

These two functions do essentially the same thing, send a line of text data from the host to the DAP system, via the communication pipe specified by the *<HD>* parameter. The communication pipe must be previously opened and configured for transfer of text from the host system to the DAP. The text or command to send is specified by the the *<text>* parameter or *<command>* parameter. The message text is converted from the internal Python string object form to an ASCII text form for transfer. The only difference between the two function variations is the intent: the *SendDAPCommand* form is reserved for command messages to be processed by the DAPL command interpreter, while the *SendDAPText* form is for various other text transfer purposes.

The functions provide processing to translate the Python string objects into ASCII character sequences and terminate them with appropriate delimiter characters. In general, avoid the NUL, CR, and LF special control characters, because these can cause the processing on the DAPL system side to interpret the messages as multiple or prematurely terminated messages.

Examples

```
from daptext import *
...
SendDAPCommand( DHstdin, "OPTIONS  ECHO=OFF" )
```

The `DHstdin` handle is previously opened as a connection to the `$SysIn` command pipe to the DAPL system. Send the DAPL system a message to disable the automatic return echo of all command line text content back through the `$SysOut` pipe. The echoing feature is very useful for interactive console display applications, but interferes with receiving error messages in typical GUI driven applications.

9. System configuration

Previous sections covered the basic things that you need to do to run an application: establish connections, send and run an embedded configuration, transfer data, close connections. But there are a few more operations at a systems level that you might need to know about.

Clearing a DAP board

The normal way of clearing a DAP board is to send a `RESET` command to the DAPL command interpreter through the `$SysIn` predefined communication pipe. This transfer operation "finishes" immediately once the command goes into the communication pipe. But that doesn't mean that the DAPL system has responded! How long it takes for the DAPL system command interpreter task to be scheduled and to actually complete the operation is indeterminate, dependent on activity occurring on the DAP board, particularly if there are other activities that delay processing of new command text.

Another way to force a reset operation is at the DAPIO level, bypassing the ordinary command text interpreter. Use a handle to the DAP board (*not* a handle to one of its communication pipes, in the normal manner).

```
HD = Hdap()
HD.Open("\\\\.\\Dap0", DAPOPEN_QUERY)
HD.Reset( )
HD.Close( )
```

When you do this there is no doubt about whether or not the board has reset. The `Reset` method does not return until the operation is complete.

Managing downloadable modules

Custom command modules containing specialized or proprietary processing can be downloaded to the DAPL system using the Data Acquisition Processor applet in the Windows Control Panel. Modules can be special application modules obtained from Microstar Laboratories, or they can be custom-developed modules that you designed and built yourself.

The default options place a copy of the module in the DAP system data area to be reloaded automatically at each boot. But what happens if this is not a behavior you want? It is awkward to force users to manually mount and unmount the modules in the Windows Control Panel for each application run. An alternative is to make the application software load these modules dynamically, and then unload them when no longer needed. This can be done using DAPIO module management methods. To load a module:

```
HD = HDap()
ModPath = "C:\\\\AppFolder\\"
HD.Open("\\\\.\\Dap0\\$BinIn", DAPOPEN_WRITE)
HD.ModuleLoad(modpath+"modname.dlm",0)
```

When you have stopped the application, cleared out the DAPL system configuration with a `RESET` command, and no longer need to use the special module, you can unload it.

```
HD.ModuleUnload("modname.dlm",0)
HD.Close( )
```

If you used a utility function like `Open1Dap`, and already have an opened handle to the `$SysIn` communication pipe, you can use that handle. You do not need to open and close a different connection to use the `ModuleLoad` and `ModuleUnload` methods.

10. Direct-disk operations

This is an advanced topic. These features are only available on Windows systems, because they require an additional resident *DAPcell Services* software layer.

Python is not famous for fast and deterministic real-time behavior. If you intend to capture data streams at rates in the millions of bytes per second – even low-end DAP models can do this – Python needs to reliably perform data block transfer operations at a rate of several thousand operations per second. If Python fails to keep pace, the DAP hardware can run out of physical buffer capacity, and data will be lost.

The bottleneck occurs when Python attempts to take the data from the DAP (which does not know anything about Windows files) and move it into the file system (which does not know anything about DAP processing). There is no way Python can do this fast enough. The DAPIO system gets around this problem, very literally, by moving all of the bulk data transfer processing to a separate system process, leaving the Python application to configure and monitor the activity at its own pace.

There are two kinds of direct-disk operations:

1. **Server-side disk logging**. The data transfers are shuttled directly from a DAP communication pipe channel to a specified file for storage – a *"data logging"* operation.

2. **Server-side disk feeding**. The data transfers are shuttled directly from a specified file to a DAP communication pipe channel – a *"data feeding"* operation.

There are four things you need to set up for automatic direct-disk operations:

1. Some special configuration changes for DAPcell Services

2. A special connection to the DAP board. (The DAPIO32 Manual calls this a *Disk I/O Handle*.)

3. A special configuration object defining the characteristics of the operation.

4. A buffer object providing intermediate storage that the operation can use during the transfer.

DAPcell configuration

The disk storage options available to you for the feeding or logging operations will depend on the version of the ***DAPtools Software*** that you use with your DAP board. If you use the *DAPcell Networked Server* from the *DAPtools Professional Edition*, you can log to file server shares (such as remote DAP hosts) on networked machines. If you use the *DAPcell Local Server* from the *DAPtools Standard Edition*, you can only use disk drives (or equivalent storage devices) physically in your local host machine. If you use the *DAPcell Basic Server* from the *DAPtools Basic Edition*, you do not have access to direct-disk features. The *DAPIO32 Manual* provides some additional information about configuring file access permissions using the *DAPCell Service.*

- Establish a Windows *Share* folder on your disk drive. It might not be strictly necessary to do this, but some obscure problems regarding "could not create file" might occur otherwise under some options. Your mileage might vary, go ahead and try other options to see if they work for you.

- Run the Data Acquisition Processors applet in the Windows Control Panel. One of the tabs at the top of the window will be *Disk I/O*. Click this tab.

- There are two *Edit* buttons. Click one of them. In the pop-up dialog, provide the full path to the *Share* folder that you set up, then click *OK* to complete the dialog. Now click the other *Edit* button, and enter the same path there. Again click *OK*.

- You should be back at the *Server Disk I/O Options* tab display. There are two *Permission* panels in this display, with radio-button options; click the *Normal* options in each. Click the *Save* button at the lower left to record your new settings.

These suggested "default settings" should be sufficient to get you going. Whether they are the proper or best settings is anybody's guess. The DAPcell Manual available from the *Microstar Laboratories* Web page might provide some insights; or you can try navigating online technical information provided by Microsoft. Be careful to record your configuration settings, when you find the ones that work, for the next time that you install or update your DAPcell software.

Server-side disk logging

For setting up a disk logging session, your application must open a handle to the DAP pipe sending the data. Most commonly, the predefined `$BinOut` communication pipe is used for this, because it is already configured to send data sample values to the host. (Another option is to set up a separate pipe such as `Cp2In` or `Cp2Out` in the *Data Acquisition Processors* applet in the Windows Control Panel.)

```
DH = OpenDAPHandle("\\\\.\\Dap0\\$BinOut",DAPOPEN_DISKIO)
```

Notice the special option code, `DAPOPEN_DISKIO`, not ordinary reading or writing.

The next step is to construct a `TdapPipeDiskLog` object to store the configuration information.

```
logcnf = TDapPipeDiskLog()
```

The disk logging options must be configured in `TdapPipeDiskLog` object fields.

Specify the path to the file that receives the data. If you have access problems, you can try making the folder a *Share*. The Python string must be translated into an ASCII character string using the encode() method.

```
path = "C:\\\\AppFolder\\Logs\\LOGFILE01.DAT"
logcnf.FilePath = path.encode()
```

An oddity of the DAPIO logging operation is that it forces the file names to be stored as upper-case characters. To prevent any subsequent confusion about this, is it recommended that you specify a file name using upper-case characters.

Specify the file opening mode.

```
logcnf.OpenMode = DAPIO_CREATE_NEW
```

The following file opening modes are available. Specify only one.

`DAPIO_CREATE_NEW`	Open the logged output data file as a new file that did not previously exist.
`DAPIO_CREATE_ALWAYS`	Open the logged output data file as a new file even if this overwrites a file that previously existed.
`DAPIO_OPEN_EXISTING`	Require that a file already exists, and opens the output file to overwrite the file with new data.
`DAPIO_OPEN_ALWAYS`	Open an output file to overwrite an existing file if that file already exists, or create a new file if no previous file exists.

Specify the minimum size of each data block to transfer. You can use this to force a best match between your data transfers and your disk sector geometry. If you want to let the system select a transfer size for you, specify the value 0.

```
logcnf.BlockSize = 0
```

Specify a maximum bound on the number of bytes to transfer. If you specify zero, this means the transfer size is indeterminate, and the logging will stop when the handle to the DAP communication pipe is closed. This value should be a multiple of the `BytesMultiple` field specified for the buffer object.

```
logcnf.MaxByteCount = 0
```

Specify server mode options. If you wish to apply none of the options, specify the value 0.

```
logcnf.LogOptions = dpdl_FlushBefore
```

The following server mode options are available. They can be used in combination, using a "+" operator to combine them.

`dpdl_ServerSide`	Locate the logged data file in the same system hosting the DAP board. (Normally, the file is stored on the system running the application.)
`dpdl_FlushBefore`	Flush the input data pipe before beginning the logging session. Be careful when using this option. If your DAPL configuration starts running before the disk logging operation does, you will lose data! (Normally omitted.)

dpdl_FlushAfter	Flush stale unprocessed data from the data pipe upon termination of logging. To be effective, the DAPL configuration must be terminated with a STOP or RESET command first.
dpdl_MirrorLog	Specify this option to enable logging to an additional file copy. Normally disallowed.
dpdl_AppendData	Append data to an existing file. Must be used in combination with one of the file opening modes DAPIO_OPEN_ALWAYS or DAPIO_OPEN_EXISTING.
dpdl_BlockTransfer	Use an ultra-large block transfer mode, highly dependent on disk hardware, and unsuitable for most ordinary data transfers. See the *DAPIO32 Manual*.

Specify a file sharing option.

```
logcnf.ShareFlags = DAPIO_FILE_SHARE_NONE
```

The following sharing flags are available. They can be used in combination, using a "+" operator to combine them.

DAPIO_FILE_SHARE_NONE	The file is accessible only by the Python application process that is managing the logging.
DAPIO_FILE_SHARE_READ	Other processes are allowed to read data from the file.
DAPIO_FILE_SHARE_WRITE	Other processes are allowed to write data to the file.

If you are going to use a DAPIO *Peek* operation to watch the progress the disk logging, specify the READ option, because otherwise there could be an access fault when the application attempts to *Peek* at the data stream while the logging (in a different process) is writing the data stream. If you aren't sure, select the DAPIO_FILE_SHARE_NONE option.

Specify file processing options. These can be used in combination, using a "+" operator to combine them.

```
logcnf.FileFlags = DAPIO_FILE_FLAG_NORMAL
```

The following file flag options are available.

DAPIO_FILE_FLAG_NORMAL	Normal sequential processing of the file. Use this if no other options apply.
DAPIO_FILE_FLAG_WRITE_THROUGH	Advanced option, use only if necessary. Attempt to optimize the data transfers by bypassing intermediate buffer storage.
DAPIO_FILE_FLAG_ENCRYPTED	Advanced option. Logged data are encrypted. See Windows documentation regarding this.
DAPIO_FILE_FLAG_SEQUENTIAL_SCAN	Advanced option. Most applications should not use this. Attempt to optimize transfers of exceptionally large data blocks.

The *NORMAL* mode is recommended unless you have some special requirements.

The next step is to create an intermediate buffer object, a `TDapBufferGetEx` buffering object (for receiving data from the DAP). You have already seen how to do this in Section 6, where bulk data transfers are covered. Typical field values for the configuration are:

```
BytesMultiple                   1
BytesGetMin                     8192
BytesGetMax                     8192
GetWait                         1000
Timeout                         0
```

The most likely adjustments to these settings are:

- Make the `BytesMultiple` field match the data type you are logging (typically 2)
- Make the `BytesGetMin` and `BytesGetMax` fields match the block size you specified in your disk logging configuration.

Once the `TDapPipeDiskLog` object and the `TdapBufferGetEx` data object are ready, the application can begin. Send the DAPL configuration to generate the data stream, in the usual manner, and start it running. Beging logging activity using the the `PipeDiskLog` method of the `HDap` handle.

```
HDap.PipeDiskLog(logcnf, incnf)
```

If you specified start-up data flushing, do these two things in reverse order: start the disk logging, and then (before the `GetWait` timing interval expires!) start the disk logging.

End the data logging session by closing the connection to the `HDap` handle.

```
HDap.Close( )
```

The *DAPIO32 Manual* provides some additional information about using the `HDap` handle while logging is underway, to query status information about the logging activity.

Server-side disk feeding

This is basically the same application as server-side logging, except that the data flow goes in the opposite direction, from disk storage to the DAP. Applications using this typically would be doing some kind of waveform generation or playback using the DAP board's digital-to-analog converters.

The setup is very similar to server-side disk logging, except that a different kind of configuration object is used, and the handle opened for DISKIO operation is associated with a communications pipe for data flowing into the DAPL system.

For setting up a disk feed session, your application must open a handle to the DAP pipe receiving the data. Most commonly, the predefined `$BinIn` communication pipe is used for this, but it must be configured to operate in the special DISKIO mode (not the familiar `DAPOPEN_DISKIO` mode).

```
DH = OpenDAPHandle("\\\\.\\Dap0\\$BinIn",DAPOPEN_DISKIO)
```

The next step is to construct a `TdapPipeDiskFeed` object to store the configuration information.

```
feedcnf = TdapPipeDiskLog()
```

The disk feed options must be configured in `TdapPipeDiskFeed` object fields.

Specify the path to locate the file that supplies the data. The Python string must be translated into an ASCII character string using the string `encode()` method. If you have access problems, you can try making the folder a *Share* folder.

```
path = "C:\\\\AppFolder\\Logs\\sigfile01.dat"
feedcnf.FilePath = path.encode()
```

Specify the minimum size of each data block to transfer. You can use this to force a best match between your data transfers and your disk sector geometry. If you want to let the system adjust transfer block sizes for you, specify the value 0.

```
feedcnf.BlockSize = 0
```

Specify a maximum bound on the number of bytes to transfer. If you specify zero, this means the duration of the logging activity is not predetermined, and the logging will be stopped by closing the handle to the DAP communication pipe or by exhausting the data from the source file. This value should be a multiple of the `BytesMultiple` field specified for the buffer object.

```
feedcnf.MaxByteCount = 0
```

Specify server mode options. If you wish to apply none of the options, specify the value 0.

```
feedcnf.FeedOptions = dpdl_FlushBefore
```

The following server mode options are available. A "+" operator can be used to combine them.

`dpdl_ServerSide`	Take source data file from a file in the same system hosting the DAP board. (Normally, the file is stored on the system running the application.)
`dpdl_FlushBefore`	Flush the data pipe from the host before beginning the feed session. (Normally omitted.)
`dpdl_ContinuousFeed`	Do not stop at the end of the file; instead, restart back at the beginning of the file to repeat the feed cycle.
`dpdl_BlockTransfer`	Use an ultra-large block transfer mode, highly dependent on disk hardware, and unsuitable for most ordinary data transfers. See the *DAPIO32 Manual*.

Specify file sharing options. The file sharing flags options are the same ones available for server-side disk logging. Unless you have special requirements, the

```
feedcnf.ShareFlags = DAPIO_FILE_SHARE_READ
```

The file sharing flags options are the same ones available for server-side disk logging. Use a "+" operator to combine them. Unless you have special requirements, the `DAPIO_FILE_SHARE_NONE` option is recommended.

Specify file processing options. These can be used in combination, using a "+" operator to combine them.

```
feedcnf.FileFlags = DAPIO_FILE_FLAG_NORMAL
```

The available file flag options are the same as the ones for server-side disk logging. Unless you have special requirements, the `DAPIO_FILE_FLAG_NORMAL` option is generally recommended.

The next step is to create an intermediate buffer object, a `TDapBufferPutEx` object to temporarily receive data intended for the DAP board. You have already seen how to do this in Section 6, where bulk data transfers are covered. Typical field values for the configuration are:

```
BytesMultiple                   1
SendWait                        1000
Timeout                         0
BytesPut                        0   (not used)
```

The most likely adjustment to these settings is to make the `BytesMultiple` field match the data type you are logging (typically 2).

Once these two objects are configured, the application can begin. Send the DAPL configuration to receive and redistribute the data stream, in the usual manner, and start it. Begin disk feed activity using the the `PipeDiskFeed` method of the `HDap` handle (in DISKIO mode). Usually, the DAP is started first, because it can wait patiently as long as necessary for the data stream to begin.

```
HDap.PipeDiskFeed(feedcnf, putcnf)
```

End the data logging session by closing the connection to the `HDap` handle, or by waiting for the data from the source file to be exhausted.

```
HDap.close( )
```

method: HDap.PipeDiskFeed

Initiate direct transfers from a disk file to a DAP board without client application intervention.

result = HD.PipeDiskFeed(hostcfg, bufcfg)

Parameters

<HD>
A previously opened communication handle configured for DISKIO transfers into DAP.

<hostcfg>
A `TDapPipeDiskFeed` object configuring properties of the transfer.

<bufcfg>
A `TDapBufPutEx` object configuring intermediate transfer buffer storage.

Returns

<result>
Flag indicating success or failure in starting transfer process.

Library

```
dapio.py
daplog.py
```

Exceptions

None

Description

Direct disk-to-DAP data transfers are initiated, using information about the properties of the disk access provided in the *<hostcfg>* parameter object of type `TDapPipeDiskFeed`, and using information about timing, storage, and so forth from the *<bufcfg>* parameter object of type `TDapBufPutEx` .

If the configuration is accepted and activated successfully, the method returns immediately, sending a *result* value of 1. If there is something wrong with the configuration and it fails to start, the *result* value will be 0.

The hard part is the setup for the configuration objects. Configuring the data buffer objects of type `TDapBufPutX` is discussed in *Section 6* of this document, under the topic of *Bulk Data Transfers*. Configuring the file system and transfer options is discussed here in Section 9 of this document, but you might need to inspect the code in the *dapio.py* module, or refer to the *DAPIO32 Manual* for details. The host system configuration is highly dependent on the operating system environment, so you might need to refer to some of its development tool documentation for information about what the file access parameter mean.

After successful return, the disk-to-DAP transfers are managed under a separate host process. You might need to query the DAPL system or the host system to verify progress of the ongoing transfer activity.

Examples

```
Chfeed = HDap()
path = "\\\\.\\Dap1\\$BinIn"
...
Chfeed.Open(path.encode(),DAPOPEN_DISKIO)
result = Chfeed.PipeDiskFeed( hostcfg, bufcfg )
if  result == 0
  raise DapException("Disk feed configuration not accepted")
```

Initiate concurrent disk file to DAP data transfers controlled by a separate process, using the `hostcfg` and `bufcfg` configuration objects previously established. Verify that the process starts successfully.

method: HDap.PipeDiskLog

Initiate direct transfers from a DAP board to a disk file without client application intervention.

result = HD.PipeDiskLog(hostcfg, bufcfg)

Parameters

<HD>
A previously opened communication handle configured for DISKIO transfers from the DAP.

<hostcfg>
A `TDapPipeDiskLog` object configuring properties of the transfer.

<bufcfg>
A `TDapBufGetEx` object configuring intermediate transfer buffer storage.

Returns

<result>
Flag indicating success or failure in starting transfer process.

Library

```
dapio.py
daplog.py
```

Exceptions

None

Description

Direct DAP-to-file data transfers are initiated, using information about the properties of the transfer and the disk access provided in the ***hostcfg*** parameter object of type `TDapPipeDiskLog`, and using information about timing, storage, and so forth from the ***bufcfg*** parameter object of type `TDapBufGetEx` .

If the configuration is accepted and activated successfully, the method returns immediately, sending a ***result*** value of 1. If there is something wrong with the configuration and it fails to start, the a ***result*** value will be 0.

The hard part is the setup for the configuration objects. Configuring the data buffer objects of type `TDapBufGetX` is discussed in *Section 6* of this document, under the topic of *Bulk Data Transfers*. Configuring the file system access options is discussed here in Section 9 of this document, but you might need to inspect the code in the *dapio.py* module, or refer to the *DAPIO32 Manual* for details. The host system configuration is highly dependent on the operating system environment, so you might need to refer to some of its development tool documentation for information about what the file access parameter mean.

After successful return, the DAP-to-disk transfers are managed under a separate host process. You might need to query the DAPL system or the host system to verify progress of the ongoing transfer activity.

Examples

```
Chlog = Hdap()
path = "\\\\.\\Dap0\\$BinOut"
...
Chlog.Open(path.encode(),DAPOPEN_DISKIO)
result = Chlog.PipeDiskLog( hostcfg, bufcfg )
if  result == 0
  raise DapException("Disk logging configuration not accepted")
```

Initiate concurrent DAP channel to disk file transfers controlled by a separate process, using the `hostcfg` and `bufcfg` objects previously established. Verify that the process starts successfully.

11. Example application

This section presents a nontrivial application example, showing the interaction between the application written in Python (using the *DAPpython* interface) and the embedded processing of the DAP board.

Application description

This application is a material strength study, to determine how much cantilever force a brittle material can withstand before cracking failures occur. There are manufacturing variations in the material, so the test must be repeated many times, on a number of sample speciments, to get a reasonable estimate of the statistical distribution. This would be tedious and error-prone to do entirely by hand, so a certain amount of measurement automation provides a great benefit. On the other hand, formal development of a full-featured application program for this kind of one-time project would be too expensive to consider.

A piece of the material under test is clamped at one end, with the other end extending out horizontally. At a precise distance away from the clamp, a pneumatic piston presses a chisel-shaped head down against the material under test. Air pumped into the pneumatic piston, gradually increases the applied force. The force level is controlled by a voltage signal applied to the control valve servo-mechanism. The actual piston pressure tends to lag the control setting, because the time is required for air to move through the valve into the pressure cylinder. To accurately determine the actual applied force, a balanced-bridge load cell is mounted behind the chisel head, reporting the measured force in the form of a differential voltage. When the applied force reaches the material breaking point, the load cell will detect a very rapid drop in force. The failure threshold is the observed force that was applied just before the detection of the cracking event.

Control signals:

1. A pressure relief "safety" valve to remain opened when the pneumatic cylinder is not pressurized. It is solenoid activated, and driven by a TTL level digital logic signal line.

2. A pneumatic pressure level command signal, driven by a digital-to-analog signal line.

The output signals:

1. A differential strain voltage signal from the load cell, requiring a gain of 40 to boost the voltage and improve the measurement resolution.

Processing by the DAP board

There are two process groups. One group at a time is used.

- When the *idle* processing is active, a zero voltage is sent to the pressure control valve, and a 0 bit is sent to the digital output port to open the relief valve and vent cylinder pressure.

- When the *ramp* processing is active, digital port bit is set to 1, enabling a buildup of pressure in the air cylinder. The control voltage, hence applied force, is smoothly adjusted upwards from 0. The material breaking point is expected in less than 10 second, but each experiment can run up to 30 seconds if necessary. At approximately one-second intervals, a status report message is sent to the host system via

text. Readings of the actual force are observed from the load cell. These readings will tend to pick up stray vibrations from the floor and pneumatic pump, so reundant samples are collected and averaged to reduce the measurement noise. Meanwhile, the load cell readings are checked for any large and rapid force reductions indicating a breakage event. When this is detected, clean measurement values prior to and following the event are selected for post-processing analysis, and transmitted to the host application.

These processes are specified in a DAPL configuration script. It will implement *both* parts of the processing, the *idle* or the *ramp* mode, but the host system will decide when each group will run.

```
// CFORCE.DAP
// Configuration script for the material failure test
// Clear any prior configurations or accumulated data
RESET

// Pre-calculated control voltage ramp data
PIPES  pRamp  WORD
FILL   pRamp  0

// Bit values for controlling the pressure relief valve
VARIABLE   valveOPEN   WORD = 0
VARIABLE   valveCLOSED  WORD = 1
VARIABLE   ctrlZERO  WORD = 0

// The pipes for cleaned pressure measurement data
PIPES  pCleaned  WORD,  pSelected  WORD

// Software trigger for recording pressure droop events
TRIGGER  tBreak  mode=normal holdoff=500

// Input sampling, used during the ramp-up processing
//   -- Watch load cell voltage using gain 40 on differential chan 0
//   -- 100 samples per millisecond --> 10 microsecond intervals
//   -- Limit of 30 seconds for sampling activity
IDEFINE  forcewatch
  CHANNELS  1
  SET  IPipe0  D0   40
  TIME  10
  COUNT 3000000
END


// Idle processing mode. The bit on the digital output port
// is turned off, releasing pressure from air cylinder.
PDEFINE  idle
  DIGITALOUT(valveOPEN,0)
  DACOUT(ctrlZERO,0)
END


// Ramp-up processing mode. Multiple activities.
PDEFINE  ramp
  // Turn on the digital control bit, closing saftey valve
  DIGITALOUT(valveCLOSED,0)
```

```
  // Generate the ramped voltage output data sequence
  pRamp = pRamp + 1
  OPipe0 = pRamp
  // Collect and process load cell force measurements
  AVERAGE(IPipe0,100,pCleaned)
  // Detect any sudden drop in force
  DLIMIT(pCleaned,  INSIDE,-32768,-200,  tBreak, \
                    INSIDE,-32768,200 )
  // Report measurements, 20 before and 20 after event
  WAIT(pCleaned, tBreak,20,20,  $BinOut)
  // Send progress report once per second via text
  SKIP(pCleaned, 0,1,999, pSelected)
  FORMAT("Current force level:  ",pSelected)
END


// Deliver ramped output voltage updates at 1 millisecond intervals
ODEFINE  rampout  1
  SET  OPipe0  A0
  TIME  1000
END
```

One intentional omission from this script is the START command. The action is initiated later, under direct control of the Python application.

Processing by the Python application

```
'''
Measure cantilever breaking force for sintered rods
'''

# Import libraries for DAP access
from  dapio  import  *
from  daphandle  import  *
from  dapblock  import  *

# Import standard Python libraries
from  time  import  sleep
from  array  import  *

# Open typical one-DAP connections and send configuration file
dhlist = Open1Dap("C:\\\\TestData\\CForce\\CFORCE.DAP")
dhmsg = dhlist[0]
dhcmd = dhlist[1]
dhdat = dhlist[2]
```

```
# Run the "idle" processing one second to establish initial state
print("Establishing initial quiescent state")
dhcmd.SendDAPCommand("START  idle")
sleep(1.0)
dhcmd.SendDAPCommand("STOP")

# Start the output control and measurement processing
print("Starting the force-ramping sequence")
dhcmd.SendDAPCommand("START  forcewatch, rampout, ramp")

# Allow experiment to run up to 320 1/10 second intervals
elapsed = 0
while  elapsed < 320
  # Display status message from DAP if one arrives
  if  dhmsg.InputAvail( )
    result = GetDAPText( dhmsg )
    print  ( result[1] )
  # Did DAP board send some measurement results?
  if  dhdat.InputAvail( )
    # Receive the measurements as 40 "half-size integers"
    result = GetDAPBlock( 40,'h' )
    print("20 measurements before and after detected break event")
    if  result[0]<40
      print("Some of the expected measurements missing")
    # Display captured data
    print( result[1] )
    # Force exit from processing loop
    break
  # Otherwise, wait for something to happen
  sleep(0.1)
  elapsed = elapsed+1


# Stop DAP measurement processing, restore quiescent state
dhcmd.SendDAPCommand("STOP")
dhcmd.SendDAPCommand("START  idle")
sleep(0.1)

# Release DAP board
dhcmd.SendDAPCommand("RESET")
Close1Dap(dhlist)
```

Notice the economy of this solution: disregarding code comments, there are about 50 combined DAPL script and Python application code lines.

--- END OF DOCUMENT ---