DAPtools for MATLAB Manual

DAPIO32 Interface for MATLAB

Version 5.10

Microstar Laboratories, Inc.

Part Number DAPMATLABM510

Contents

Table of Contents

1 Introduction

DAPtools for MATLAB™ allows MATLAB applications to communicate with a Data Acquisition Processor™ (DAP) from the MATLAB programming environment. This integrates the processing power of a Data Acquisition Processor with the powerful technical computing environment of MATLAB. DAPtools for MATLAB allows the MATLAB environment to digitize signals on demand and apply this data directly. In addition, MATLAB applications can take advantage of real-time processing within the DAP system for numerical signal preprocessing, data selection, real-time signal playback, and other time-critical concurrent operations.

2 Getting Started

DAPtools for MATLAB requires a working knowledge of MATLAB scripting, and also a certain amount of knowledge about how to configure and operate a Data Acquisition Processor (DAP) using commands to the DAPL operating system. The DAPL operating system runs in the DAP board's embedded environment. You don't need to control hardware in real time – the DAPL system takes care of all of that for you. But you must configure the DAPL system to provide the digitized data at the rates you want, on the number of channels you want, with the data selection or processing you specify. See the DAPL Manual and online tutorial materials for full information about using features of the DAPL system. Basic examples are provided with this package, and in many cases new applications can be configured using slight variants of these examples.

System Requirements

This software is intended for use with versions of MATLAB R2009b or later. It has been tested with MATLAB 2013a. It will work with 32-bit operating systems Windows XP, Windows Vista, Windows 7, and Windows 8. It will work with 64-bit operating systems Windows 7 and Windows 8, either in native 64-bit mode (by installing the 64-bit version of MATLAB) or in 32-bit compatibility mode (by installing the 32-bit version of MATLAB). There have not been many fundamental differences in the support for pre-compiled commands using basic features of MATLAB, so you might find that this DAPtools for MATLAB version works fine with older or more recent versions of the MATLAB, and older or more recent versions of the Windows system. If you find that new MATLAB releases present problems under the supported versions of Windows, please report to Microstar Laboratories.

If you need your application to run on an older operating systems and older versions of MATLAB, contact Microstar Laboratories. Perhaps a previous version of the DAPtools for MATLAB package is available that will work well enough for you.

Installation

Follow the steps below to install the DAPtools for MATLAB.

1. Install the DAPtools Standard or DAPtools Professional software on your system. Run the SETUP.EXE program and install the DAPcell Server software. After this installation, shut down the system and install the DAP board. Reboot the system, and run the Data Acquisition Processor application in the Windows Control Panel. Verify on the Browser sheet that the Data Acquisition Processor is recognized by the system.

2. Run the `SETUP.EXE` program from your DAPtools Software again. Click on *DAPtools for MATLAB* and follow the instructions onscreen to complete the installation.

3. If you select the usual destination for your installation, the software is located at `C:\Program Files\Microstar Laboratories\DAPtools\MATLAB`. There are some additional example application script that are placed in the `\User\Documents\MATLAB` area under your user name.

Configuration

To tell the MATLAB system where you installed the DAPtools for MATLAB commands, you need to add their location into the MATLAB search path. The easiest way to do this is to get administrator privilege on your machine and set up a startup.m script file in the root install directory of your MATLAB software, for example

```
C:\Program Files\MATLAB\2013a
```

Use a text editor (or Notepad) to create a text file and enter the following lines.

```
%startup.m     user-configurable startup file
%  This script is executed when MATLAB starts, if it exists
%  anywhere on the path.
disp('Adding DAPtools for MATLAB to the MATLAB search path');
mslpath='C:\Program Files (x86)\Microstar Laboratories\DAPtools\MATLAB';
addpath(mslpath,path);
```

Save the file and exit the editor.

New Features

Be sure to check the README.TXT file that comes with the DAPtools for MATLAB package, for the latest news about installation, feature changes, and so forth.

3 DAPIO and MEX Functions

The functions that access DAP boards are provided by the programming API called *DAPIO*. The file that implements this interface is called `dapio32.dll.` These functions are provided by the DAPcell Server that runs in you host Windows system. The MATLAB package doesn't automatically know that this API exists – so this is where *DAPtools for MATLAB* comes in. This package provides pre-compiled MATLAB extension functions that also know about the DAPIO interface services, to give the developer access. The `dapio.32.dll` file is installed automatically with the *DAPcell Server* software, and registered with the Windows system so that it is available to applications.

The pre-compiled MATLAB functions – the “MATLAB Extension files” – are known as “`mex` files” and are identified by the file type extension: `.mexw32` when the functions are compiled for running in a 32-bit system (or in a 64-bit system running in 32-bit compatibility mode), or file extension `.mexw64` when the functions are compiled for running native 64-bit mode in a 64-bit system. These files are typically installed in the `Program Files (x32)\Microstar Laboratories` area of your file system by the DAPtools for Matlab installer. The MATLAB command search path makes these commands available to your MATLAB applications.

The DAPIO API features do not correspond one-for-one with the DAPtools for MATLAB `mex` functions, and parameters of the `mex` functions to not mesh perfectly with DAPIO service calls, because the Windows internal environment and MATLAB environment are very different. However, all of the most important API features are available, and the correspondence is pretty close. The names of the `mex` files differ from the names provided by the DAPIO32 API mostly for historical reasons – they were shortened and made more “MATLAB-like,” for better or worse, so we are stuck with the same old names today.

The following table compares the `mex` functions and `dapio32.dll` entry point names.

MEX Functions for MATLAB	**DAPIO32 DLL Functions**
dapclose	DapHandleClose
dapclrpa	DapConfigParamsClear
dapcnfig	DapConfig
dapcpc	DapComPipeCreate
dapcpd	DapComPipeDelete
dapfedpd	DapPipeDiskFeed
dapflshi	DapInputFlushEx
dapflsho	DapOutputEmpty
dapgavl	DapInputAvail
dapgetm	DapBufferGetEx
dapgstr	DapLineGet
dapmdin	DapModuleInstall
dapmdld	DapModuleLoad
dapmduin	DapModuleUninstall
dapmduld	DapModuleUnload
daplogpd	DapPipeDiskLog
dapopen	DapHandleOpen
dappavl	DapOutputSpace
dappstr	DapLinePut
dapputm	DapBufferPutEx
dapquery	DapHandleQuery
daprecnf	DapConfigRedirect
dapreset	DapReset
daprinit	DapReinitialize
dapsetpa	DapConfigParamSet

4 General Notes and Hints

Quick command help

You can obtain abbreviated help information in the MATLAB command window for any function in the DAPtools for MATLAB package. Type the command line `help` *`function`* (for example, `help dapclose`).

Function return values

Return values report whether the functions have executed successfully. Most of the returned values are messages (string) or error codes (number). The function descriptions and the DAPIO32 Reference Manual explain these codes. These can be deceptive sometimes, because the DAPIO interface will successfully complete its mission – and report success. But this mission might be to tell the DAP system to do something impossible, causing an error condition in the embedded DAP board environment that is harder to observe. The examples show how to poll the DAPL system for information, including error reports.

Boolean result codes

Some of the return values from functions are *boolean* error codes. A value of 1 (true) indicates that the processing wsa successful. A value of 0 (false) indicates that the processing failed. The `dapgerr` function can sometimes provide more information about the failure.

Function argument names

Names used for function arguments in the command descriptions are illustrative only. Those variables can have any sort of name acceptable to the MATLAB environment.

String arguments

String arguments are used to specify the UNC addresses for accessing DAP services, full or relative paths to files, and various option keywords. Follow the general MATLAB syntax, enclosing strings with single-quote characters.

Command lines

The DAPL system expects *terminated command text lines*, but the MATLAB system has no specific concept of *text line*. Use ordinary MATLAB strings to represent commands, without any special termination characters. When you send a command string, the DAPtools for MATLAB functions automatically apply any required termination characters.

UNC addresses

UNC addresses are strings that identify the DAP resource that you wish to access. The UNC addresses come in four flavors.

- A *null address*, represented as an empty MATLAB string `''`.
- An address something like `'//host'` points to the entire host machine. When used to access DAP services, it implicitly refers to the DAPcell server and all of the DAP boards on that host. The shorthand name `'//.'` refers to the local host machine running the MATLAB application.
- An address something like `'//host/Dap0'` points to one specific DAP board located in the indicated host. Most systems have one DAP board, identified as `Dap0`.
- An address something like `'//host/Dap0/Cp2In'` points to one specific virtual data channel, often called a "communication pipe", between the MATLAB application and the specified DAP board.

Predefined communication pipes

The DAPcell Server automatically provides the following communication pipes. To a certain extent, you can adjust their properties, but they will always be present. Unless you have a specialized application that transfers data using an advanced mix of access strategies and data rates, these four predefined communication channels are probably all that you will need.

1. `$SysIn` - command text going into the DAP board
2. `$SysOut` - response message coming out of the DAP board
3. `$BinIn` - downloaded binary data going into the DAP board
4. `$BinOut` - uploaded measurement data coming out of the DAP board

Handle variables

Handle variables are critical to accessing the DAP board. Handles identify the DAP board or its communication channel that you have opened for access. During your application processing, you must preserve the values of the *handle variables* – most of the DAPtools for MATLAB functions will need them to identify which DAP board and which data channel to access. After the communication connections are closed, the handle variables are no longer required and can be discarded.

DAP processing cycle

When using the DAPcell services through DAPIO functions, it is very much like communicating with a separate host over a network. A four-step process is used.

1. Establish a connection. Use the `dapopen` function, specifying the Windows system UNC resource name for the board you want to access. You receive back a *handle variable* that you will use to identify the connection.

2. Tell the DAP system what you want it to do. Either issue a sequence of direct commands using the `dappstr` function, or pass a collection of commands from a text file using the `dapcnfig` function.

3. Transfer the data. Use `dapgetm` or `dapputm` functions to do the transfers.

4. Release the DAP system when you are finished with it. Use the `dapclose` function.

Some of the `mex` functions take `filename` as input parameter. In general, you can identify files using a UNC path to a shared file, a full filesystem path with file name on the local host system, or a path to a file relative to the MATLAB environment current working folder.

Function return values

Some of the `mex` functions have optional return values. All names shown in the command reference pages and examples are illustrative, such as `handle, code, numBytes, dataM,` and `string.` These names can be chosen by the user.

Some of the return value arguments are optional. If no return value location is provide, results are unobservable except for their side effects. In some cases, when the command has no way to report problems through a result code, the command will raise a MATLAB error that terminates application script execution.

Time control

Some functions have optional `TimeWait` and `TimeOut` parameters, specifying time intervals. Substitute the desired interval lengths in units of milliseconds for these. Be careful, because the Matlab conventions typically represent time in units of seconds – a hint at the difference in time scales at which the DAPL system and MATLAB system operate.

`TimeWait` specifies the longest time in milliseconds that the operation will sleep while waiting for data. If no data show up in that amount of time, the function returns. `TimeOut` specifies a time limit in milliseconds during which the operation should complete. If it fails to complete in this amount of time, the function returns. Whichever condition arises first will force the return. Specifying a value of zero for a timing parameters typically means to apply a default behavior, which can be to ignore the specification. Using no timeouts can be dangerous, because it could easily leave the application permanently deadlocked waiting for data that will never arrive. Using default timeouts is safe, but could take many seconds to return control. Timing limits are imprecise, so there might be additional time delays before the Windows system returns control to your application. You will need to check the return values to determine whether the operation was completed normally or terminated early. More information about timing control is available in the *DAPIO32 Reference Manual*.

Options lists

Some specialized function parameters allow selection of certain processing options. These options are specified as MATLAB strings containing certain reserved keywords separated by blank characters. Two consecutive single-quote characters represent an empty option string – equivalent to a request “please apply the usual defaults.” Spelling is critical in the keywords.

DAPL system configuration scripts

Just as MATLAB needs script files to tell its computing engine what to do, the DAPL system needs command scripts to tell its embedded processor what to do. DAPL scripts are provided for the example applications described in the next chapter. The DAPL scripts are typically organized into the following sections.

1. **Preparation.** Comments identifying the purposes of the script. A reset command terminates any previous DAP board activity that may have been left by accident.

2. **Declarations.** Define some additional processing elements when needed, such as configurable constants (for example, offset and gain adjustments for calibrated sensor devices), shared variables, and pipes for streaming data between tasks.

3. **Input sampling.** Most applications will use this, since *acquisition* is central to *data acquisition*. This section specifies the number of channels, their signal range properties, the timing cycle for automatic capture of samples, and in some cases some special properties for how sampling starts and stops.

4. **Processing.** Many applications use just one processing command – to tell the DAPL system which data streams to copy to the host system. For example,

   ```
   copy(IPipes(0..7), $BinOut)
   ```

   This example tells the DAPL system to take the set of 8 channels configured in the input sampling section, and transfer them to the host through the pre-configured `$BinOut` data transfer communication pipe.

   More advanced applications might be able to take advantage of other processing features, such as "skipping" to reduce the size of data transfers, digital filtering for avoiding aliasing and noise, taking data selectively when special events are detected, or reorganizing samples into contiguous groups.

5. **Output conversions.** Some specialized applications use this feature to configure automatic digital-to-analog hardware, so that sampled signals are precisely reconstructed as analog signals.

6. **Operation.** While it is always possible to tell a DAP board configuration to run using a separate `start` command, sometimes the application wants the board to begin as soon as possible, and it is convenient to place the `start` command into the script for immediate execution.

Refer to the DAPL Manual for complete information about the standard features available with the DAPL system.

5 Application Examples

The DAPtools for MATLAB package includes several example script files showing how to use the DAPtools For MATLAB MEX functions. This following chapter briefly describes each application or technique, and why it might be relevant to you.

Copies of the scripts described in these examples are placed into your User\Documents area when the DAPtools for MATLAB package is installed.

Establishing Connections

Applications will establish connections to the DAP boards and their communication channels, use those connections for a while, then release them. This section covers the first step.

There is no harm in setting up the connections very early: it reserves access to the DAP board, without actually changing anything until you start to send commands.

The commands discussed here are provided in the example script file **dapinit_simple.m**.

- Your application will need to open a channel so that you can send commands to the DAP board. Commands are sent through the predefined `$SysIn` communication pipe. The DAP boards on your system are assigned names of the form Dap0, Dap1, etc. so that the host system can identify them using UNC networking and resource names.

```
texthandle = dapopen('\\.\Dap0\$SysIn', 'write')
if texthandle == 0
   error('Error opening DAP input command text handle')
end
```

- Almost every application does data acquisition in the classic sense – capturing digitized data – and will need to open a channel for delivering this data to the host system. This uses the predefined `$BinOut` communication pipe.

```
binaryhandle = dapopen('\\.\Dap0\$BinOut', 'read')
if binaryhandle == 0
   error('Error opening DAP binary output handle')
end
```

- At the beginning of your application run, you probably do not expect to have something already running on the DAP board, producing and consuming data. In case this happened by accident, send a command to stop this activity.

```
dappstr(texthandle, 'reset');
```

- Any prior activity could immediately deliver meaningless data into your data transfer channel the moment you connect to it. Now that the source of the data has been stopped, you can remove the undesired data.

```
dapflshi(binaryhandle);
```

Closing Connections

All well-structured applications that initiate activity on DAP boards and that terminate normally should attempt to "clean up" and clear out data and applications that are no longer needed. Connections to the board should be released before the application terminates, so that other applications – or another run of this application – will have access.

The commands discussed here are provided in the example script file **dapclose_simple.m**.

- First, stop any processing activity that might still be running, by sending a STOP command to the DAPL system running on the DAP board.

```
dappstr(texthandle,'stop');
```

- If you will no longer be using the measurement configuration that you set up, clear it from DAP memory by sending a RESET command to the DAPL system.

```
dappstr(texthandle,'reset');
```

- Now use the dapclose function to release the connections to the communication channels that were opened at the start of the application.

```
dapclose(texthandle)
dapclose(binaryhandle)
```

Robust Management of Connections

Rather than using the “simple” channel management examples shown in the previous two sections, we suggest using more robust MATLAB scripts that package all of the opening and closing operations into simple and easy-to-use one line operations.

Code discussed here is available in the **dapallopen.m** and **dapallclose.m** scripts.

```
% Open connections to DAP board 0
dapallopen;

% Use the DAP connections...

% Close connections to DAP board 0
dapallclose;
```

You already know everything about how this works. The `dapallopen` and `dapallclose` scripts just cover more “housekeeping” considerations.

Conventional communication handles

There are actually four predefined communication pipes that you should know about. These scripts try to open or close a handle to each channel automatically, making the channels available if you need them, doing no harm if you don't need them. If you use these scripts, you will soon come to expect that these handles are always available and you won't think about them much. All of the other examples use these.

Handle variable	Associated communication pipe	Purpose
`hTextToDap`	`$SysIn`	Commands going to DAPL system on DAP
`hTextFromDap`	`$SysOut`	Messages returned to host system from DAP
`hBinToDap`	`$BinIn`	Precomputed signals downloaded to DAP
`hBinFromDap`	`$BinOut`	Sample values delivered to host from DAP

Non-robust handle horrors

If you use simple command sequences to initialize and close your connections, these will typically work just fine... until something goes wrong.

Suppose for example that your script opens a handle variable to the `$SysIn` pipe, allowing it to send commands to the DAP board. You now collect data for 100 samples and process them... oops! You intended to collect 128 samples, so your index is out of range for the 100 sample matrix. Your application terminated, but no matter. That is why you have MATLAB, it is simple to patch the script and run it again.

When your script starts up again the next time, the first thing it tries to do is establish connections to the DAP communication channel for sending commands – and this fails. Some application out there has already reserved the DAP board to write to that channel. Well yes, it was really this same application, just a different run, but the DAPcell Server doesn't know that. To clear the problem, you can manually call the `dapclose` function, and this clears the channel so your script can open it again. Only, you can't do that either. When you tried to open the handle the second time, and this failed, the `dapopen` function returned a 0 value to flag the error. The returned 0 value wrote over the top of the handle value you established in the previous run. You no longer know the handle value to close the connection, and until you close it you can't connect to it or use it.

To recover from this kind of disaster, you must exit the MATLAB application to disassociate it from the DAP boards, and then you must run the Data Acquisition Processor applet from the Windows Control Panel. Click on "Stop DAPcell Service" to clear all of the existing connections to the DAP board. Click on "Start DAPcell Service" to restore normal operation.

This isn't something you want to do very often.

More robust handle management

If you use the `dapallopen` and `dapallclose` scripts, and your application gets one of the dreaded "communication pipe is already reserved" errors, manually type `dapallclose` into the command window and then run your application again. Most of the time, this repairs the problem.

The reason it can do this: the two scripts work together, and know which handles to expect. The `dapallclose` script won't try to close a handle that for some reason is not valid, but it will close all handles that appear to have been opened. The `dapallopen` script will not change the values of any handle variables until it is sure that operations were successful, and if handles appear to be already opened, they are not opened again.

You can study the scripts for more details. For more advanced applications, you might need to generalize the scripts to work with multiple DAP boards or additional communication pipes.

Configuring for Data Sampling

After you establish connections to the DAP communication channels, it is time to do something useful with them. Since data acquisition is primarily about acquiring data, some additional script lines will do this.

Actually, acquiring data consists of two closely related parts: the operations that run on the DAP board to collect and deliver the data, and the operations in your application that receive the data. We will look at what happens in the application script first.

The code discussed here is provided in the **data.m** example file.

The application script

- The DAP board must be told how you want your samples captured. This information is collected in a text script file (DAPL operating system script, not an m-file script), and sent to the DAP board through the command pipe. If the DAPL system script embeds a `start` command, as quickly as you send this, the data can start to arrive.

```
cnfg = dapcnfig(hTextToDap, 'data.dap');
   if cnfg < 1
      % Print error message
      error('Error configuring DAP')
   end
```

- Request the data. You will specify the data type in which the data are transferred, but there is an automatic conversion to the floating point internal format used by MATLAB. In this example, data arrive from two channels in the 'natural' order that they are taken by the sampler, alternating samples for each channel. The results are returned in a two-row matrix with 1000 elements.

```
channels = dapgetm(hBinFromDap, [2, 500], ‘int16’)
```

The DAPL configuration script

The DAPL configuration determines what data the application receives, and how that data is organized. The `data.dap` script file that the `dapcnfig` function sends has the following sections.

- The first section specifies that there are two single-ended analog-to-digital conversion channels, and the capture cycle for these channels in combination is 10 microseconds.

```
idefine  fastcapture
  channels  2
  set ipipe0  s0    ; analog 'stimulus' channel, single ended
  set ipipe1  s1    ; analog 'response' channel, single ended
  time  5.0
end
```

- The next section tells the DAPL system what processing to perform. In this case, it only needs to transfer all of the sampled data to the host application, in the natural sequence, without reordering or grouping. There is no point in waiting, so the entire configuration is started immediately.

```
pdef  sendall
  copy(IPipe(0,1), $BinOut)
  end
start
```

Extensions

The `data.m` script shows how high-resolution data are expected at 1 second intervals. If you examine the `data.dap` script file, you can see how the processing definition section includes some data-selection processing between the sampling and the data transfers.

With time intervals of 10 microsecond to cover each pass through the channels, the example configuration captures 100 matrix data blocks every second. If you have previous experience with real-time data management in Windows systems, you will know that trying to do anything regularly at 10 millisecond intervals is a very uncertain business. The work is scheduled erratically over time. Some kinds of processing, such as “live” graphics must be updated at much lower rates. But the sampling can't be reduced to operate at that same rate, otherwise the required time resolution is lost and the data are not meaningful.

The solution shown in the script is that data are taken selectively – returning some data blocks and discarding others to reduce the volume of data traffic.

Logging Data to Disk

Saving sampled data to a disk data set for later analysis is perhaps the most common application of data acquisition. Data logging using a Data Acquisition Processor is a particularly good fit for the MATLAB environment, because the DAP can take care of the fast real-time considerations, while the MATLAB environment covers the chore of getting all of that data moved to the disk concurrently. The **logfile.m** script shows an example.

Though the data are produced at a regular rate, when things begin to go fast, data appear to arrive irregularly by the time they reach the MATLAB environment. By providing a sufficient buffer area for the transfer pipes, irregular transfers will be collected and the transfers to the disk can be done in more orderly large blocks.

The MATLAB `fopen` function is used to prepare the file and receive a file handle to access it. Those details, and the details of opening and closing the DAP communications are skipped.

What typically remains in the logging application is a large run-time loop that alternately looks for data and moves it. For this application, it is known that the samples will be processed in groups of 1024 values, and this is deemed an acceptably large block size for the data transfers.

- Various strategies can be used for determining when this loop is done. For this example, the termination condition for the loop is that a predetermined number of samples are collected.

```
total_terms_written=0;
while   total_terms_written < 102400

  % Are new samples available?
   ...

  % Move them.
   ...

end
```

- The `dapgavl` function is used to determine whether enough data are available in the transfer buffers.

```
  Bavail = dapgavl(hBinFromDap);
  Navail = floor(Bavail / 2);
  if  Navail < 1024
     continue   % Try again later
  end
```

- Knowing that a full block is available, it can be moved from the DAP transfer buffers to the disk without any further delays. One transfer operation moves one block.

```
    channel0 = dapgetm(hBinFromDap,[1024, 1], 'int16');
    fwrite(fid, channel0, 'short');
    total_terms_written = total_terms_written + 1024
```

Exit the application loop when the number of terms reaches the limit. After the exit, close the files and the DAP connections.

Extensions

The idea of moving data in convenient moderate-size groups to match the way the data will be analyzed can be an expensive luxury when data are arriving very fast. For such cases, there is never an issue of not having data, but amounts of data available can vary a lot because of timing variability. It is probably a better strategy to omit looking for data with `dapgavl` in these cases. Simply ask for data immediately using a `dapgetm` function with a moderately large requested matrix size and a relatively short timeout option. You are very unlikely to wait for very long. Any other activity besides the transfer loop is likely to cause delays that will degrade the net logging rate.

For extremely demanding applications, using the Local Server or Network Server editions of *DAPcell Server*, available with the *DAPtools Software Standard* and *Professional* editions, will allow you to use the `daplogpd` function to log data to disk without using a control loop in the MATLAB environment. The data can also move data across network connections if necessary.

Commands and Error Messages

Error messages and codes that you can receive from the *DAPtools for Matlab* functions can be deceptive. They might report “completed successfully,” which means that the request was transmitted to the DAP board successfully. But that doesn't mean the the DAP board was able to process the request.

Checking for errors requires message passing between your application and the DAPL system. This is done using the `dappstr` and `dapgstr` functions. You can see a lot more information about using these function in the `dstring.m` example script file.

In this example, we want to see if we have somehow produced a crash condition that leaves the DAP board unable to respond. This is uncommon, but far from impossible. Sending a HELLO command to the DAP is roughly the equivalent of running a *“Hello, world”* application in other environments... it demonstrates that something is able to run.

- First, the HELLO command is sent to the DAPL system as a command text line using a `dappstr` command.

```
dappstr(hTextToDap, 'hello');
```

- The HELLO command will provoke a one line response from the DAPL system if the system is running well enough to respond. The response is almost immediate, so allow only a very short waiting interval

```
message=dapgstr(hTextFromDap,250)
```

If you fail to receive anything more than an empty string, the DAPL system is in trouble and the board needs to be reset.

Extensions

Receiving error messages works much the same way, except that you have options whether to check for them as they happen, or to poll for them at specific times of your choosing. The example script shows examples of both strategies.

Two Way Data: Stimulus and Response

Stimulus and response testing is tricky with MATLAB. While MATLAB can compute extremely advanced data signal sequences, it has little control over the Windows environment to determine when the stimulus goes out, and relative to that, when the responses come back.

Using a DAPL system, a MATLAB script can download a predetermined stimulus signal and buffer it in DAP memory. Then, a single START command initiates the stimulus and response processing. The responsibility for accurate timing, both for regenerating the analog stimulus signal and for observations, then falls entirely on the DAP board. Stimulus and response signals are both monitored, in parallel, so there is very little timing uncertainty once the processing starts.

Details about the two directions of data transfer can be examined in the **biway.m** script.

- The first thing to do is prepare the stimulus data set and download it to the DAP through the `hBinToDap` communication connection. The matrix stimulus contains the samples of the stimulus signal to generate.

```
dapputm(hBinToDap, stimulus, 'int16');
```

- The stimulus data is stored, but not used immediately. After everything is downloaded and the test is ready, one START command will initiate all activity.

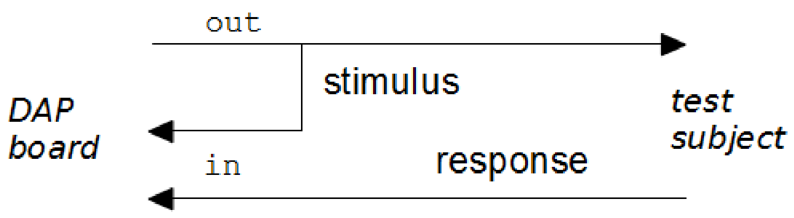
```
dappstr(hTextToDap,'start');
```

- All that is necessary now is to collect an appropriate number of data samples to match the duration of the stimulus and expected response signals.

```
response = dapgetm(hBinFromDap, [2,300], 'int16');
```

Extensions

The **biway.dap** configuration is a very questionable sort of simulation, for purposes of demonstration only. In an actual stimulus-response experiment, you will need a DAPL configuration to support the following connections between the DAP board output, the test subject, and DAP board signal inputs.



This loopback configuration, monitoring the output signal sent to the subject under test, and simultaneously monitoring the response, establishes a very tight time-association between the stimulus and response signals. There can be some delay before the hardware updating cycles begin signal output generation, but this doesn't matter, because you can see all of the delays as they occur in the paired data streams.

The DAPL system configuration to support this has the following requirements:

1. Processing must copy the predetermined stimulus sequence from the `$BinIn` communication pipe to an output channel pipe.
2. The output channel pipe is configured in an output procedure that specifies the digital-to-analog conversion port address and the output timing.
3. An input sampling procedure is configured to accept the data from the two input signal channels, the stimulus-loopback and the response.
4. Processing route the samples captured by the two signal channels in the input procedure back to the host through the `$BinOut` communication pipe.

If you apply suitable scaling for your measured stimulus-response data sample sets, they are very well suited for direct display in MATLAB graphics.

Waveform Generation and Playback

The stimulus-response application ideas suggested one way to generate an output waveform: have MATLAB calculate the waveform data, convert it into the appropriate range for the output converters, and download the data for a DAP output procedure to generate.

Here are two other possible strategies that can be useful in special applications.

Direct on-board signal synthesis

The DAPL system provides some simple waveform generator functions that might be easier to use – because there is no data stream to precompute and download. The data are produced efficiently on-the-fly.

Examine the **waves1.m** script and the **waves1.dap** configuration to see how waveform data can be generated. This particular example doesn't do anything very useful, it just sends some data to the MATLAB system for plotting, but this at least demonstrates what is possible to produce on-board. The MATLAB system could do all of that an more, but only by pre-computing it, not real time. The story can be very different if the waveforms must be issued continuously and accurately in real time. See the Frequency Response section to see an advanced development of this idea.

This example has one extra feature: there is a **waves.m** script showing how a stimulus-response application might be controlled manually using a click-button GUI widget. These displays are updated fast enough that they would be easy to mistake for “real time live action.” But don't be fooled... the displays are “real time” only on a human (and Windows) time scale.

Direct playback of a prerecorded real signal

The normal features of MATLAB file I/O can be used to read a file of binary data previously logged. The data, they are typically loaded into a matrix within the MATLAB workspace, in a manner similar to the following:

```
Nchan = 4;
fid = fopen('databin.log', 'rb');
bindata = fread (fid, [nchan,inf], 'int16')
fclose(fid);
```

After the data are loaded, one of the options available is to feed the data set back to the DAP board using the `hBinToDap` communication pipe handle.

As an alternative to logging signal data in raw binary form, MATLAB supports saving and reloading data in a .WAV file format (uncompressed PCM). This data file format can also be loaded for processing by quite a few applications that might otherwise find raw logged data confusing.

The `playwav.dap` and `playwav.m` script files show details of how data are sent to a DAP board for real-time playback. A `playwav.wav` file is also provided, with a few seconds of signal data compatible with the scripts.

Time Sequence Analysis

Everybody knows that when you are using MATLAB, you have arguably the world's most powerful mathematical engine and a world class graphical data display engine – but without a great time-sense. Your Data Acquisition Processor system can provide data to the MATLAB system with rigorous timing. This should be a perfect combination.

If you run the `fourier.m` example script, it will set up a graphical display window. Each time you click the button, this script invokes the **fourier1.m** script, which in turn uses the **fourier1.dap** configuration on a DAP board to deliver new data for a 3-axis time-frequency plot. This kind of analysis is business-as-usual for MATLAB, to such a degree that it hardly worth mentioning. Except...

The unique thing about this example is that MATLAB is not doing any of the time-frequency analysis. The plot is showing the raw spectrum data as delivered directly from the DAP board.

As a practical matter, you are probably not going to use MATLAB this way. But still, this hints at the real-time processing resources available on the DAP board to run in parallel with MATLAB processing. Some of the more meaningful things that your on-board DAP processing might do:

- *Noise reduction.* Sample at a much higher rate than strictly necessary, then smooth the data by averaging or digital filtering with decimation to achieve the intended data rate.

- *Frequency-selective digital filtering,* using zero-phase-distortion FIR filter DSP operations, to help isolate the information relevant to your MATLAB analysis.

- *Selective data capture.* Observations of *events* in a data stream can often determine efficiently whether captured data blocks contain anything relevant for applying a full MATLAB analysis. If there is nothing to see, the DAP board does not need to send anything.

- *Parallel data buffering*. For example, suppose that MATLAB performs a complex analysis with multiple graphical data displays. While doing these things, it is NOT collecting new data – processing is serialized. Ordinarily, before the next analysis cycle can start, MATLAB must request that data, and then wait for it to be collected and delivered, and this can take a while. Using DAPL system processing in parallel, MATLAB might not need to wait at all – when the analysis is done, the next data set is already collected and waiting. And you might not need to do anything for this, just provide enough buffer space.

- *Critical timing.* For events that are easily detectable with a simple analysis of the data stream, MATLAB might not get a chance to see the data until a response is too late. By embedding time critical processing in the DAPL system, these response latencies are reduced by orders of magnitude.

Frequency Transfer Response Measurements

For some reason, university students always seem to get into trouble with this kind of application – trying to get real-time response out of the Windows system when it isn't there.

What is the obvious way to calculate a system frequency response? (Often, this is the critical step for building a system model using MATLAB tools.)

1. Use MATLAB to calculate a stimulus signal sequence with known frequency.

2. Download the data set to drive the digital-to-analog converters, generating a timed analog signal to excite the system.

3. Measure the response of the system.

4. Analyze the magnitude and phase angle of the response signal and compare to the magnitude and phase of the excitation signal at the known frequency.

5. So far, this yields the system frequency response at one frequency. Repeat N times. If you don't mishandle the data sets, eventually you can collect the response data as a reasonably clean transfer response curve.

While this is tedious, trying to automate this process so that MATLAB properly coordinates the excitation signal and response analysis in slow real time is no great joy either.

The **freqresp.m** example shows how easy it is to delegate the signal generation and response collection to the DAPL system, as a special case of stimulus-response testing. A continuously-adjusted, swept-frequency signal is generated on-board using a `sinewave` command. This provides roughly equal excitation energy at each frequency from roughly 0 to ¼ of the Nyquist limit of the sampling. Exactly equal excitation energy doesn't matter, but providing a relatively uniform energy distribution and avoiding excessive noise variance can make a huge difference. The DAPL configuration coordinates the signal generation, and collects measured data as one large block of input/output pairs. The input and output frequency characteristics are relatively trivial to analyze using FFT methods. With a few repetitions, and a little data smoothing, you will get excellent data for model fitting.

6 Alphabetical Command Reference

This section provides details about each function in the DAPtools for MATLAB package, listed by name.

dapclose

Close a target handle to a DAP, communication pipe, or server.

```
<boolean> = dapclose(handle)
```

Parameters

`handle`

Handle to a previously opened handle to a DAP, communication pipe, or server.
`double.`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function closes the specified handle which was returned by an earlier call to the `dapopen` function. The return argument boolean is optional. If no return argument is given, error checking is handled internally, and an error message will occur if the processing fails.

Examples

```
dapclose(hTextFrpmDap)
```

Close the previously opened handle specifies by variable `hTextFromDap`.

```
ret = dapclose(hBinToDap)
```

Close the handle specifies by `hBinToDap`, and retain the return value in variable `ret`.

See Also

`dapopen`

dapclrpa

Clear all program-defined substitution macros and clear all pre-set defaults.

```
<boolean> = dapclrpa( )
```

Parameters

```
(None)
```

Return Values

```
boolean
```

Optional return code, always succeeds.
`double`

Description

This function clears all program-defined substitution macros and any default values for them previously provided to the application by the DAPIO system and prior calls to the `dapsetpa` function. The current implementation has no way to detect failure and always reports 1 (true). Future versions may have more information and might return 0 (false), so there is no harm to check the result.

Example

```
dapclrpa
```

Clear all program-defined and default parameters in memory without any result code value.

See Also

`dapsetpa`

dapcnfig

Send a DAPL command file through a DAP communication pipe, with parameter substitution.

```
<boolean> = dapcnfig(handle, filename)
```

Parameters

`handle`

Handle to a communication text pipe to a DAP, previously opened with write access.
`double`

`filename`

UNC or local host path and filename of the DAPL command file to send.
`string`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function sends a DAPL command file to a the DAPL system on the DAP board associated with the `handle` variable. For more information, please see `DapConfig` in the *DAPIO32 Reference Manual.*

Example

```
dapcnfig(hTextToDap, ‘c:\test.dap’);
```

Sends the text content of the DAPL command file `test.dap` from the current working directory to the DAP com pipe specified by `hDapSysin`.

See Also

`daprecnf, dapsetpa, dapclrpa`

dapcpc

Create a logical communication pipe channel between the PC and a DAP.

```
<boolean> = dapcpc( ...
    'pInfo <[type=dT maxsize=v1 | maxsize=v2 blocking=bsz]>')
```

Parameters

`pInfo`

UNC path specifying the host, DAP board, and pipe name to create.
`UNC field in string.`

`type = dT`

Optional host-side attribute text specifying the data type of elements to transfer through the pipe. The *dT* text field must be *byte*, *word*, *long*, *float*, *double* or *text*.
`Keyword field in string.`

`maxsize = v1`

Optional host-side attribute specifying the transfer buffer size limit in units of elements.
The specification `v1` should represent an integer value of at least 1024.
(If omitted, the DAPcell software supplies a generally good default value.)
`Integer field in string.`

`maxsize = v2`

Optional DAP side attribute specifying the maximum pipe buffer size in unit of elements.
The specification `v2` should represent an integer value of at least 1024.
(If omitted, the DAPcell software supplies a value matching the one on the PC side.)
`Integer field in string.`

`blocking = bsz`

Optional DAP side attribute specifying a transfer group size. Data are collected until this number of data elements are available, and then they are transferred as a group. The default is 1 element. In certain cases of slow data, increasing the blocking size can avoid rapid small transfers that significantly load the host CPU.
`Integer field in string.`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function tells the DAPcell server to create a create a communication pipe channel between the PC and a DAP. This function has one mandatory string parameter. At a minimum, this string must include a UNC field `pInfo` that specifies the host, DAP identity, and new pipe name to create. These pipe names have restricted forms. For most applications, the DAPcell Server will provide default values for all other parameters, and all of the options text can be omitted.

If specified, the remaining text in the parameter string must be separated from the UNC path by a blank space character and then enclosed within square bracket characters as shown. The remaining option parameters consist of two groups, the first specifying properties on the PC side, and the second specifying properties on the DAP side. The two sides must be separated by a vertical bar character. If parameters for one side or the other are all omitted, the vertical bar character is also be omitted. Keywords parameters for these properties must either be omitted or specified in the order shown and with the exact spelling shown, with keyword and equal sign separator. The placeholders shown on the right of the equal sign separator must be replaced by numeric text characters specifying the desired number values. For more information about the communication pipe configuration options, please see the `DapComPipeCreate` function in the *DAPIO32 Reference Manual.*

The effects of this persist over multiple system reboots, potentially affecting all applications that use the specified DAP board in the specified host, so it should be used with caution.

Examples

```
dapcpc('\\.\Dap0\Cp2Out')
```

Create a com pipe `Cp2Out` with default characteristics provided by the DAPcell Server.

```
dapcpc('\\Station2\Dap1\Cp4Out [type=word maxsize=2048]')
```

Create a com pipe `Cp4Out` for `Dap1` in remote host `Station2` with default DAP side attributes and a 2048 element limit for the host side transfer buffer capacity.

```
ret = dapcpc('\\.\Dap0\Cp4In [type=long | maxsize=65536]')
```

Create a com pipe `Cp4In` for transferring 32-bit integer data, reserving buffer storage for up to 64K values on the DAP side. Retain the return value for checking success.

See Also

`dapcpd`

dapcpd

Remove a communication pipe connection from the specified host and DAP board.

```
<boolean> = dapcpd(pInfo)
```

Parameters

`pInfo`

UNC path naming the host and communication pipe to be deleted.
`string`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function tells the DAPcell Server in the host specified by the UNC string argument `pInfo` to delete the communication pipe channel also specified in the UNC string argument. If the target communication pipe channel does not exist, this function returns success without doing anything. An application can use the same `pInfo` string used to create the pipe with the `dapcpc` function, including additional attribute information – any attribute information will be ignored. For more information, please see function `DapComPipeDelete` in the *DAPIO32 Reference Manual*.

This function has effects that persist over multiple system reboots, potentially affecting all applications that use the specified DAP board in the specified host, so it should be used with caution.

Example

```
dapcpd('\\.\Dap0\Cp4In [type = word, maxsize=2048]’)
```

Deletes the com pipe `Cp4In` from `Dap0` on the local host machine, ignoring the pipe attributes field.

```
ret = dapcpd('\\Remote1\Dap0\Cp4Out’)
```

Deletes the com pipe connection `Cp4Out` to DAP board `Dap0` located at station `Remote1`, and save the result of the operation in return variable `ret`.

See Also

`dapcpc`

dapfedpd

Initiate a background disk feed session that transfers data from a data file to a DAP board.

```
<boolean> = dapfedpd(handle, flags, filename, fileShareMode,
      fileFlagAttr, maxCount, blockSize
      <,[bytesMult, timewait, timeout]> )
```

Parameters

`handle`

Handle to a DAP communication pipe previously opened for `diskio` access.
`double`

`flags`

A list of zero or more keyword options specifying the behavior of the disk feed operation.
Allowed options: `ServerSide`, `FlushBefore`, `ContinuousFeed`, or `BlockTransfer`.
`string`

`filename`

Windows system UNC or local file system path to a data file providing the data.
`string`

`fileShareMode`

A text specifying the file sharing property of the data file.
Allowed options: *`'read'`*, or *`'write'`*.
`string`

`fileFlagAttr`

A list of option keywords specifying additional file access properties.
Allowed options: *`'normal', 'readonly', 'encrypted'`*, or *`'sequentialscan'`*.
`string`

`maxCount`

Maximum number of bytes to feed before automatic feed termination (see `bytesMult`).
If zero is specified, there is no predetermined maximum.
`double`

`blocksize`

Minimum amount of data (in bytes) to read from the data file if available.
If 0 is specified, use the default value.
`double`

`bytesMult`

Number of bytes written must be restricted to multiples of this value.
If 0 is specified, use the default value.
`double`

`timewait`

Longest time the feed operation can wait for space to become available on DAP.
If 0 is specified, use the default value.
`double`

`timeout`

Longest time for the feed operation to complete.
If 0 is specified, use the default value.
`double`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function initiates a disk feed session. During a disk feed session, data will be read from the file specified by `filename` and transferred in blocks to DAP through the communication pipe associated with the open `handle` variable. The communication pipe must be previously opened using the mode `diskio`.

There are many configurable parameters.

- The `flags` argument controls the behavior of the disk-feed operation. The `flags` options for are covered under *`bmFlags`* in the `TDapPipeDiskFeed` section of the *DAPIO32 Reference Manual*. They are a "bitmask" in the actual DAPIO control structure, but specified by space-separated keywords in the `flags` string.

- The argument `fileShareMode` defines how the data file is read. The file sharing options for are discussed under *`dwFileShareMode`* in the `TDapPipeDiskFeed` section of the *DAPIO32 Reference Manual*.

- The argument `fileFlagAttr` specifies the file access mode for the data file. The file processing options are discussed under *`dwFileFlagsAttributes`* in the `TDapPipeDiskFeed` section of the *DAPIO32 Reference Manual*.

- The parameter `maxCount` specifies a limit on the how many bytes should be fed through. If this value is not zero and the count of data bytes transferred reaches this limit, the transfer stops automatically.

- The parameter `blockSize` specifies how many bytes to move in each block of data transfer, and sometimes adjusting this size can optimize the throughput efficiency of the transfer (which is machine-dependent).

After it is started, the data feeding operation continues in the background. The `dapfedpd` function call returns immediately. The MATLAB application can continue doing other processing.

This function does not use a file handle, since that is specific to MATLAB I/O functions. Instead, the DAPcell service opens the file separately and defines an association between it and the specified handle used by the DAP for the feed operation. When the `dapclose` function is used to close the connection between the application and the communication pipe, the DAPcell service will then close the file and disassociate it with the application.

If it is necessary to monitor the process of the disk feed process, it can be done using the `dapquery` function to check whether the disk feeding operation has finished.

The optional parameters `BytesMult`, `timewait`, `timeout` are used as a group. These parameters are sometimes helpful for optimizing the transfers.If you want to use defaults for all of these, they can be omitted from the list. To use one of these parameter, all three must provided. However, by specifying a value of zero, you can still apply the default for an individual parameter.

- You can use `BytesMult` to specify a data transfer grouping. If you happen to know that data are always used in groups, such as 1K blocks from FFT transforms, aligning the transfers to this block size can sometimes avoid inefficient tearing and reassembly of data blocks.

- You can use `timewait` to specify how long the DAPcell Server should wait for the DAP to be ready to accept new data. Because the DAP board has plenty of buffer memory, it is almost always ready. If processing stalls for some reason, `timewait` can detect this.

- You can use `timeout` to specify a limit on the total time required for the playback. This allows early termination of processing that is moving at a very slow and erratic pace.

Default values are supplied for parameters when the function call specifies values of 0. For these cases, the DAPcell Service provides generally suitable defaults, as follows:

- `maxcount` The default behavior is that there is no maximum limit. The feed connection must be terminated by closing the communication handle or exhausting the source data file.

- `BytesMult` The default value is 1. There is no restriction on how many bytes are taken from the file in any given read operation. The `maxcount` and `blocksize` values must be multiples of this.

- `blocksize` The default is to let the Windows system decide how much data to take from the file in each disk read operation.

- `TimeWait` The default is 1000 milliseconds (1 second) before the DAPcell Service decides that the DAP has stopped accepting any more data.
- `Timeout` The default behavior is that no maximum time limit is applied to the disk feed. It will stop when the disk file is exhausted or the handle for the feed channel is closed.

There are too many advanced options available in the various option strings to cover here. For more information, please refer to the `DapPipeDiskFeed`, `TDapPipeDiskFeed` and `TDapBufferPutEx` sections in the *DAPIO32 Reference Manual*.

Examples

```
dapfedpd (hDap, 'ContinuousFeed', 'c:\inData.dat', '', '', 0, 16384)
```

Feed the data from the file `inData.dat` in the local `c:` drive to the DAP specified by the handle `hDap`. The file is repeatedly fed to the DAP board to provide a continuous stream of data, transferred in blocks of 16384 bytes.

```
ret=dapfedpd(hDap, 'ServerSide', '\\PC5\Sigdat\inData.dat', ...
    '', '', 60000,  0, 0, 100, 20000)
```

Feed 60000 bytes of data from the file `inData.dat` in share area `Sigdat` on host `PC5`, and stream this data to the DAP binary data pipe specified by the handle `hDap`. The DAP board associated with this handle is also located at host `PC5`, which was established when the handle was opened. The `'ServerSide'` option indicates that the data file for the feed is at the location of the system that hosts the DAP board, not the local system that hosts the MATLAB application. Since the local host must address files on the `PC5` machine using a *network share*, the file address is in the form of a network UNC address rather than a local disk\file path. The data will be transferred using default options. This operation will wait at most 100 milliseconds for available space in the DAP communication buffer. If the entire feed operation does not complete in less than 20 seconds, terminate the operation early. The `ret` return value indicates whether the feed process started normally.

See Also

`dapquery`

dapflshi

Flush a DAP output communication pipe for reading.

```
<boolean> = dapflshi(handle <, timeout, timewait>)
```

Parameters

`handle`

A previously opened handle to a communication pipe for reading data from a DAP.
`double`

`timeout`

Optional limit on total time for the operation to complete, in units of milliseconds.
`double`

`timewait`

Optional quiet time interval for assuming that all data are removed, in units of milliseconds.
`double`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function flushes all data in the target pipe specified by `handle`. The data can be text or binary. The `handle` must refer to an opened communication pipe for transferring data from a DAP to the host application. Since the DAPcell software does not know why data are in the pipe, it attempts to clear everything it finds there by repeatedly reading and discarding the data, until no more are received. Clearly, this is hopeless if the DAPL system is actively generating new data as fast as DAPcell can discard it, so always be sure to first `reset` all processing on the DAP board.

The optional `timeout` parameters limits the amount of time allowed to complete the operation. If you omit the `timeout` argument, a value of 20000 (milliseconds) is used by default.

If the third `timewait` interval has passed without any new data arriving, it is presumed that all data have been flushed from the pipe, and the `dapflshi` function returns, reporting success. If the `timewait` argument is omitted, a default value of 100 (milliseconds) is assumed. If data continue to arrive after the `timeout` interval has passed, the `dapflshi` function reports failure.

If you omit the optional return value, you cannot tell whether the flush operation succeeded or failed.

Example

```
  ret = dapflshi(hTextFromDap, 3000, 1000);
```

Allow up to 3 seconds to flush all of the stale message text from the communication pipe to receive message from the DAP, with handle `hTextFromDap`. After removing all of the old message text, the pipe must remain empty for at least 1 full second before reporting success in the return location `ret`.

See Also

`dapflsho`

dapflsho

Empty any undelivered data previously written by the application to a DAP communication pipe.

```
<boolean> = dapflsho(handle)
```

Parameters

`handle`

Handle to a communication pipe previously opened in a '*write'* mode.
`double`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function completely empties buffers for the communication pipe previously opened for writing data to the DAP, as identified by the specified `handle` variable. When it returns, the pipe is completely empty on both sides, on the host and on the DAP.

Unlike the `dapflshi` function, the host application has control of data generation, and this clean-up operation always finishes quickly, so no timing control parameters are needed.

Examples

```
ret = dapflsho(hBinToDap);
```

Completely empties the communication pipe for sending binary data to the DAP, specified by the handle `hBinToDap`, and stores the function termination status in the return variable `ret`.

See Also

`dapflshi`

dapgavl

Get the number of bytes available for reading from a communication pipe.

```
numBytes = dapgavl(handle)
```

Parameters

`handle`

Handle previously opened for reading text or binary data sent by a DAP.
`double`

Return Values

`numBytes`

Number of bytes available for reading if successful, or -1 if function failed.
`double`

Description

This function gets the number of bytes available for reading from a communication pipe previously opened for reading data from the DAP, as specified by `handle`. When the call is successful, the returned number `numBytes` is a number of data bytes already buffered by the server and available immediately for the application on the host to read and process. The returned value is in units of bytes, so for the case of numeric data, the application must divide the returned value by the length of a data element (in bytes) to determine the number of data element available. An application that reads the reported number of bytes, or less, will not have the application process *blocked* and awaiting data that the DAP has not yet produced. However, the value is only a lower bound, and might not represent *all* of the data actually available in the host buffer storage.

Unless the application has other critical processing that cannot be delayed even for a short time while waiting for new data, it is generally better to use `dapgetm` with a *timeout* specification.

Example

```
ret = dapgavl(hBinFromDap)
```

Get the number of bytes available for reading in the pipe specified by handle `hBinFromDap`. The returned number of data bytes available is stored in the return value argument `ret`.

See Also

`dappavl`

dapgetm

Get a matrix of data sent from a DAP through a communication pipe.

```
[dataM <, numData>] = dapgetm(handle, [mrows, ncols], dtype, ...
    <, timeWait, timeOut> )
```

Parameters

`handle`

Handle to a communication pipe opened for reading data.
`double`

`[mrows, ncols]`

Dimensions of the matrix to take from the com pipe.
`1x2 row matrix`

`dtype`

The type of data in the returned matrix `dataM`.
`keyword string`, must be *‘int16’*, *‘int32’*, *‘double’* or *‘float’*

`timewait`

Optional maximum limit on the time for the first data to arrive, in millisconds.
`double`

`timeout`

Optional maximum limit on the total time for the operation to finish, in millisconds.
`double`

Return Values

`dataM`

Data matrix taken from the com pipe, zero-matrix if function failed.
`matrix of double`

`numData`

Optional result variable, the number of matrix items if successful, or -1 if function failed.
`double`

Description

This function receives a matrix of data from a communication pipe previously opened for reading data from a DAP, as specified by `handle`. The dimension of the data matrix is specified by a two-element row matrix with arguments `mrows` and `ncols`.

The returned data matrix is filled column by column, so that each row corresponds to one data channel source, and each column corresponds to one sampling pass through all of the channels. The type of data being read is specified by the parameter `dtype,` which must be one of the following string values:

'int16'	specifies 16-bit integer
'int32'	specifies 32-bit integer
'double'	specifies double precision floating point
'float'	specifies single precision floating point

The `timewait` and `timeout` parameters are used as a pair – if one optional argument is used, the other must be specified also. The `timewait` interval specifies the amount of time to allow before the data transfer begins. If no data are available during this interval, it is assumed that the DAP is not producing any data, and the function returns early. The `timeout` interval specifies the total amount of time allowed to accumulate all of the data for the full matrix. If these arguments are omitted, the following defaults are supplied:

`timewait`	100 (milliseconds)
`timeout`	20000 (milliseconds)

If the data fetch operation is successful, the full matrix of data is stored in the return argument `dataM.` If a second result variable is specified, the number of items read is stored in the argument `numData.` Unlike some other commands, this function knows the data type, and it can return its count in *items* rather than *bytes*. For a successful transfer, the count of terms received will will equal the matrix size. If the count of received terms is from 0 to something less than the matrix size, there is a timeout and the matrix is not valid – the terms not received will have artificial zero values. If there is some other kind of problem the return value is -1.

Example

```
[dataM, nterms] = dapgetm(hBinFromDap, [3, 1024], ‘int16’);
```

Get data from the com pipe specified by `hBinFromDap`. It reads 1024 data sample values from each of three data channels, converting the 16-bit fixed point data to yield a 3-by-1024 term matrix `dataM` in the normal MATLAB `double` data type. Default timing values are used. The number of data terms is stored in `ret.` A successful operation returns an `nterms` value of 3072.

```
dataM = dapgetm(hBinFromDap, [40, 1], ‘double’, 400, 500);
```

Get data from the com pipe specified by `hBinFromDap`. It gets forty data items from one data source and places them into a 40-element column vector. Because the data type as sent from the DAP is already double format, there are no numeric conversions. If no data are available in 400 milliseconds, or if the operation does not finish in 500 milliseconds, the function terminates early and some of all terms could be zero values instead of actual data.

See Also

`dapputm`

dapgstr

Get a string of characters from a Data Acquisition Processor communication pipe.

```
[outputStr <, numBytes>] = dapgstr(handle <, timewait>)
```

Parameters

`handle`

Handle to a communication pipe previously opened for reading data from DAP.
`double`

`timewait`

Optional time limit to wait for a complete text line.
`double`

Return Values

`outputStr`

Characters read from the DAP com pipe.
`string`

`numBytes`

Optional, the number of bytes read from the target pipe if successful; 0 if function failed.
`double`

Description

This function reads a line of text from the com pipe specified by handle and returns it as a MATLAB string in variable `outputStr`. This command is useful primarily for retrieving error messages and various text information requested from the DAP.

The parameter `timewait` specifies the longest allowed time to complete this operation. If omitted the default value of 1000 milliseconds is applied.

The number of characters received in the returned text is indicated by optional return variable `numBytes`. If there is no text to return, the value of `numBytes` is set to 0, and `outputStr` is set to an empty string. There is no way to distinguish a failure to receive any text from other kinds of failures.

DAPIO32 interprets text from the DAP as *text lines* terminated by CR (carriage return) control characters. If data comes from files that follow MS-DOS conventions, with lines terminated by CR and LF (linefeed) pairs, the linefeed control character is ignored. If data comes from directly from a file that follows UNIX/Linux conventions with lines terminated by LF characters only, these lines will not be correctly recognized.

In contrast, MATLAB treats strings as objects containing text, with no concept of text line or termination characters. Before returning the text to MATLAB as a string, the `dapgstr` function detects and removes the line termination control characters. The terminating CR character is considered a valid part of the input text line, and included in the `numBytes` count, even though it is not stored in the returned string. Any LF characters are considered extraneous and are neither placed into the returned string nor counted.

Normally, any positive return value indicates that text was returned successfully. However, if there is an exceptionally long text line, or if a timeout interval occurs, the function must return before processing a termination character. For such timeout or excessive text cases, the MATLAB `size` property of the string will equal the returned `numBytes` count. For the cases where the end of the line was reached normally and the CR character discarded, the MATLAB size property will be less by 1 character.

Example

```
daptext = dapgstr(hTextFromDap)
```

Get a string of zero or more characters from the DAP communication pipe specified by the handle `hTextFromDap`, and stores the string in the return argument `daptext`.

```
[textFromDap, ret] = dapgstr(hTextFromDap, 100)
```

Get a string of characters from the DAP com pipe specified by the handle `hDapSysout` and store the string in the return argument `textFromDap`. The maximum time to wait for a complete string is 100 milliseconds, or 0.1 seconds. The number of characters in the string, plus one, is saved in the return argument variable `ret`.

See Also

`dappstr`

dapmdin

Install a downloadable command module and load to DAP board(s).

```
<boolean> = dapmdin(handle, filename <, flags>)
```

Parameters

`handle`

Handle to a DAP or a DAPcell server host, previously opened with *'write'* access.
`double`

`filename`

Local or UNC path name for the module file to be installed.
`string`

`flags`

Optional string with space-separated keywords specifying the installation mode.
`string keywords`: must be *nocopy*, *noreplace*, *noload*, forceregister*,* or *forceload*.

Return Values

`boolean`

Optional return code reporting success or failure.
double

Description

This function installs a module specified by `filename` to the DAP(s) specified by `handle`. If `handle` is associated with a single DAP, the module is installed to the associated DAP board. If `handle` is associated with a DAPcell server, the module is installed on all DAP boards in the server's host system.

Installation options can be specified using keywords placed in the optional `flags` string. For full information, please see the description of *`bmFlags`* under the `DapModuleInstall` function in the *DAPIO32 Reference Manual*. If the `flags` argument is omitted, default values are supplied; these will be satisfactory for most applications.

Installation of a module is a persistent operation that places the location and identity of a module in the host system registry. Once a module is installed, it will be reloaded automatically each time the system boots, and each time the DAPcell services are started, until the module is uninstalled. For full information about the option flags, please see `bmFlags` under the function `DapModuleInstall` in *DAPIO32 Reference Manual.*

Examples

```
ret = dapmdin(hdap, 'c:\custom.dlm');
```

Install the `custom` module from the root directory of the host's local C: drive to the DAP associated with handle `hdap`, using default installation options. This will cache a copy the file to the DAP software directory created by the DAPtools SETUP program, replacing any existing module by the same name. A copy of the module is then loaded into the target DAP board or boards, as long as this does not produce a conflict with the current DAPL system configuration.

```
ret = dapmdin(hdap, 'c:\custom .dlm', '');
```

This means exactly the same thing as the previous example, since default options are requested.

```
ret = dapmdin(hdap, 'ztruncm.dlm', 'noCopy noLoad');
```

Install the `ztruncm` module from the current MATLAB working folder to the DAP associated with handle variable `hdap`. Specify that the module should be loaded from its original path, not cached at a location reserved by the DAPtools software. Also specify that loading of the module should be deferred until the next time that the DAPcell server is started or the system is rebooted.

See Also

`dapmduin, dapmdld, dapmduld`

dapmdld

Load a downloadable command module to DAP board(s).

```
<boolean> = dapmdld(handle, 'filename' <,'flags'>)
```

Parameters

`handle`

Handle to a DAP or a server, previously opened with *'write'* access.
`double`

`filename`

UNC or local host path \ name of the module to be loaded.
`string`

`flags`

Optional blank-separated keywords for downloading mode.
`String keywords:` must be *'noreplace'* or *'forceload'*.

Return Values

`boolean`

Optional return code reporting success or failure.
double

Description

This function loads a module from the location specified in the `filename` string to the DAP(s) specified by `handle`. The module is not installed in the host system, consequently, restarting the DAPcell software or rebooting the system will undo the load operation. If `handle` refers to a single DAP, the module is loaded to the target DAP board. If handle refers to a DAPcell server, the module is loaded to all DAP boards on the machine hosting that server.

The loading mode can be set using option keywords in the optional `flags` argument. For full information about the option flags, please see `bmFlags` under the function `DapModuleLoad` in *DAPIO32 Reference Manual.*

Examples

```
  ret = dapmdld(hDap, 'c:\fbcontrol.dlm', ‘noreplace’);
```

Load the `fbcontrol` module to a DAP specified by handle `hDap`, taking the original copy of the file from the root directory of the local `C:\` disk drive, with an instruction to to not replace any previous copy of that module if it is already loaded to the DAP board.

```
  ret = dapmdld(hdap5, '\\hdap5\shared\fbcontrol.dlm');
```

Load the `fbcontrol` module to a DAP specified by previously-opened handle `hdap5`. This previously-opened handle points to a DAP board located on the remote station `'//host5/Dap0'`. The module file to load is located on that same host, in a *share folder* specified by the UNC path. Default loading options are used; consequently, the module is loaded to the specified DAP board only if there is no conflict with the configuration currently present in the DAPL system on that board.

See Also

`dapmdin, dapmduin, dapmduld`

dapmduin

Uninstall a downloadable command module and remove from DAP board(s).

```
<boolean> = dapmduin(handle, modname <,flags>)
```

Parameters

`handle`

Handle previously opened for *'write'* access to a DAP or a server.
`double`

`modname`

Base name of module to be uninstalled, without path or file type extension.
`string`

`flags`

Optional list of blank-separated keyword options specifying the uninstallation mode.
`string keywords`: must be *'noload'*, *'forceregister'*, *'forceload'* or *'removedependents'*.

Return Values

`boolean`

Optional return code reporting success or failure.
double

Description

This function uninstalls a module specified by argument `modname` from the DAP(s) specified by variable `handle`. The handle variable must be previously opened for the *'write'* access mode. If `handle` refers to a single DAP, it uninstalls the module from the target DAP board. If handle refers to a DAPcell server, it uninstalls the module from all DAP boards on that server host.

The uninstallation mode can be adjusted by providing an optional `flags` string. For full information about the uninstall options, see the discussion of `bmFlags` under the function `DapModuleUninstall` in the DAPIO32 Reference Manual.

An uninstallation operation is persistent. Once the module uninstallation is complete, that module is unavailable for all applications that might use the affected DAP boards, until the module is reinstalled.

Example

```
      ret = dapmduin(hDap, 'dapliir');
```

Uninstall the `dapliir` module from the DAP board associated with the handle `hDap` using the default uninstallation options. The module will be uninstalled only if there are no conflicts with the current configuration present in the DAPL system on that DAP board.

See Also

`dapmdin`, `dapmdld`, `dapmduld`

dapmduld

Unload a module from DAP board(s) without uninstalling it.

```
<boolean> = dapmduld(handle, modname <, flags>)
```

Parameters

`handle`

Handle to a DAP or a server, previously opened with *'write'* access.
`double`

`modname`

Base name of module to be unloaded, without path or file type extension.
`string`

`flags`

Optional space-separated list of keyword options specifying the unloading mode.
`string keywords`: must be *'forceload'* or *'removedependents'*.

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function unloads a module `modname` from the DAP(s) associated with `handle`. If `handle` is associated with a single DAP, the module is unloaded from the specified DAP board. If `handle` is associated with a DAPcell server, the module is unloaded from all DAP boards on that server host.

Unloading options can be specified in the `flags` argument. For more information, please see `bmFlags` under function `DapModuleUnload` in the *DAPIO32 Reference Manual*.

Unlike the `dapmduin` function, the effects of this function are not persistent. If the module was previously installed, it remains installed, and it will be automatically reloaded into the DAPL system the next time the DAPLcell Server is restarted or the system is rebooted.

Examples

```
ret = dapmduld(hDap, 'testm');
```

Unload `testm` module from the DAP specified by handle `hDap` with default unloading mode: do not force unloading the module from the target DAP boards if there is some kind of configuration conflict, and do not remove dependents of the module during unload.

```
ret = dapmduld(hDap, 'dapliir', ‘removedependents’);
```

Unload the `dapliir` module from the DAP specified by handle `hDap,` specifying an option to resolve dependency conflicts by removing from the current DAPL configuration any element referring to the unloaded module. This will leave the DAPL system in an unbroken state with the specified module erased from DAP board memory, but because of the erased dependencies, the parts of the configuration that remain might not be fully operable.

See Also

`dapmdin, dapmduin, dapmdld`

daplogpd

Initiates a disk logging session between a DAP communication pipe and a disk file.

```
<boolean> = daplogpd( handle, flags, filepath, fileShareMode,
    openFlags, fileFlagAttr, maxCount, blockSize
    <, [xflength, timewait, timeout]> )
```

Parameters

`handle`

Handle to a communication pipe previously opened with *'diskio'* access..
`double`

`flags`

A list of space-separated keywords, specifying logging mode options.
`keyword string`: must be *ServerSide*, *FlushBefore*, *FlushAfter, MirrorLog, AppendData*, or *BlockTransfer*.

`filepath`

Path\name of a primary disk log file and optionally a secondary mirror log file.
`string`

`fileShareMode`

A keyword option specifying file share properties of the disk log file.
`keyword string`: must be *'read'* or *'write'*.

`openFlags`

A list of space-separated keywords, specifying file opening options.
`keyword string`: must be *CreateNew*, *CreateAlways*, *OpenAlways.* or *OpenExisting*.

`fileFlagAttr`

A list of keyword options specifying file share properties of the disk log file.
`keyword string`: must be *normal*, *encrypted, writethrough*, or *sequentialscan*.

`maxCount`

Maximum number of bytes to write to the log file, or 0 to specify no limit.
`double`

`blocksize`

Number of bytes to write in each disk-write operation, or 0 to use the default.
`double`

`xflength`

Optional, number of bytes to get from the com pipe at a time, or 0 to use the default.
`double`

`timewait`

Time limit in milliseconds for new data to arrive from communication pipe, or 0 to use the default.
`double`

`timeout`

Time limit in milliseconds for logging process to complete, or 0 for no time limit.
`double`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function initiates a disk logging session between a DAP communication pipe identified by argument `handle` and a disk file identified by the address `filename`. The operation continues as an independent process until a time limit condition is satisfied, or until the initiating process closes the `handle` to terminate the connection to the DAP data channel.

This function does not use the facilities of MATLAB to process the log file, so the application *does not* open a file and obtain a file handle variable in the usual way. Instead, the file access is provided by the DAPcell Server. The path to the file can be a UNC address to a *share file*, or a file system *path\file* name for files on the local host machine.

The application must previously open a connection to the DAP communication pipe for sending binary data out from the DAP, using a `dapopen` function call, retaining the returned `handle` variable value. Instead of opening the connection in the usual *'read'* mode, the communication pipe connection is opened with *'dapio'* mode.

The `daplogpd` function establishes an association between the communication pipe and the disk file, and initiates the logging activity in a separate process, returning control to the caller. While logging runs in the background, the MATLAB application can continue doing other processing. The application has no further direct interaction with the communication pipe or file until after the logging operation is done. Logging continues independently until a time limit condition is satisfied, or until the initiating process closes the `handle` using a `dapclose` function call to terminate the connection to the DAP communication channel.

There are lots of configurable options. Most applications can use defaults values for all of the options. For keyword options, defaults are requested by specifying an empty string. For number options, defaults are requested by specifying a value of 0. For complete information about the options and what they do, please refer to `DapPipeDiskLog`, `TdapPipeDiskLog,` and `TDapBufferGetEx` sections of the *DAPIO32 Reference Manual*.

- `flags` keywords control the behavior of the disk-logging operation. If the *MirrorLog* option is included in the list, a secondary file name must also be specified within the string, separated from the primary file name with a semicolon character. For more information, see *`bmFlags`* under `TDapPipeDiskLog` in *DAPIO32 Reference Manual*.

- `fileShareMode` keywords specify exclusion properties for file access. The default can be used unless other processes must concurrently access the logged data in real time as the data are written.

- `openFlags` keywords specify options for how to treat pre-existing or non-existing disk files. For more information, see *`dwOpenFlags`* under `TDapPipeDiskLog` in *DAPIO32 Reference Manual*.

- `fileFlagAttr` keywords specify options for processing the data stream. For more information, see *`dwFileFlagsAttributes`* under `TDapPipeDiskLog` in *DAPIO32 Reference Manual*.

- `maxsize` specifies a maximum data grouping to write to the log file in any given write operation. Disk controller buffering rarely has difficulties of this sort, and most applications should specify 0 to let the Windows system decide how to write the data.

- `blocksize` specifies a minimum data grouping to write to 'filename' at a time. When data are known to always arrive in certain multiples, for example 1024 values from a data transform block, specifying a restricted transfer block size can sometimes improve efficiency. Specify 0 to use the default value of 8192 bytes.

- `maxCount` specifies an upper limit on the total amount of data in bytes to log. Specify 0 if there is no predetermined limit on the amount of data to log.

The optiona larguments `xflength, timewaitt,` and `timeout` control the timing of data transfers in the communication pipe. If one of the optional parameter is used, all three must present. For more information, see `TdapBufferGetEx` in *DAPIO32 Reference Manual*

- `xflength` specifies the size of data blocks, in bytes, to move through the communication pipe in each transfer cycle. For more information, see *`iBytesGetMin`* and *`iBytesGetMax`* under `TdapBufferGetEx` in *DAPIO32 Reference Manual* – both fields are set to the `xflength` value. Some applications that process data in known, fixed, moderately large block sizes can improve efficiency by adjusting this setting. If the size is too small, it could force the system the to "thrash," causing rapid transfer cycles that move very little data. Specify 0 to get the default behavior of arbitrary transfer sizes.

- timewait specifies the maximum time interval in milliseconds to wait for new data to transfer through the communication pipe. If no new data appear in the specified time, it is presumed that the DAP has stopped sending data. Logging is then terminated and the file is closed. Specify 0 to use the default interval of 1000 milliseconds.

- timeout specifies the maximum time interval for the entire logging process to finish, regardless of how much or data arrives. After this time interval expires, the logging process is terminated, and the file is closed. Specify 0 if there is no predetermined logging time limit.

Since the daplogpd function returns to the caller immediately after the logging is started, it has no means to inform the calling application when the logging process terminates due to a data or time limit. You can use features of the dapquery command to determine whether the logging process is stopped.

Example

```
daplogpd (hDap, ‘flushBefore’, ‘c:\logfile.log’, ...
    ‘’, ‘’, ‘’, 0, 0)
```

Log the data from the DAP specified by handle hDap to a file called logfile.log in the root directory of the local c: disk drive. The target pipe specified by hDap is first flushed, which might take a while. After that, the DAP can be started and the data recorded. Default values for other parameters are used, so recording will continue until the data stream stops.

```
ret = daplogpd(hDap, ‘serverSide’, ‘\\PC2\logs\testdat.log’, ...
    'read',‘CreateAlways’, ‘normal’, 0, 0, 4096, 100, 0)
```

Log the data from the DAP communication pipe specified by handle hDap, previously opened for *'diskio'* access on a remote station called PC2. The log file *testdat.log* is also on the PC2 station and is identified by its UNC address. The serverSide option is specified to tell the daplogpd function not to open a disk file locally. Other applications can have *'read'* access to the logged data file concurrent with the writing of the file. The CreateAlways option allows the log file to overwrite a previously existing file with the same name. Restrict data block transfers through the communication pipe to a fixed length of 4096 bytes. If no new data are available after waiting 100 milliseconds, assume that the data stream has stopped and terminate the logging operation. The function reports whether the logging process starts successfully in return variable ret.

```
daplogpd (hDap, ‘serverSide MirrorLog’, ...
 ‘c:\datlog\file.log;\\Archive\shared\mirror.log’, ‘’, ‘’, ‘’, 0, 0)
```

Log the data from the DAP specified by handle hDap to a file with name file.log located on the local host in folder c:\datlog. Also write a second *mirror* copy of the logged data in a file called mirror.log, located at a remote system called Archive, in its *shared* network folder. Default values for other parameters are used.

See Also

dapquery

dapopen

Opens a handle to a specified DAPcell server, DAP board, or communication pipe.

```
handle = dapopen(uncpath, mode)
```

Parameters

`uncpath`

UNC path identifying the Server, DAP or communication pipe to access.
`string`

`mode`

Operating mode of the target to be opened.
`keyword string`, must be one of : *'write'*, *'read'*, *'query'* or *'diskio'*

Return Values

`handle`

Handle value for the opened DAP, communication pipe or server.
Nonzero if successful, 0 if function failed.
`double`

Description

This function establishes a connection to a server, DAP or communication pipe, depending on the form of the `uncpath` specification. The function returns a number handle that is used by other functions, somewhat similar to the way that MATLAB manages data file access.

The `mode` argument specifies the access mode.

write	opens a connection for writing data with exclusive access
read	opens a connection for reading with exclusive access
query	opens a connection to query information without exclusive access
diskio	opens a connection for direct-to-disk I/O access

The handle of a communication pipe opened with *diskio* mode can be used later to initiate and terminate a direct pipe disk I/O session using `dapfedpd` and `daplogpd` functions. It can also be used to query disk I/O status using the `dapquery` function. The direct-to-disk services are only available with the *DAPCell Local Server* or *DAPcell Networking Server* software versions.

More information about connections and handles is available under the `DapHandleOpen` function in the *DAPIO32 Reference Manual*.

Examples

```
hBinFromDap = dapopen('\\Server2\Dap0\$binout', 'read')
```

Open the communication pipe `$binout`, which is on `Dap0` located in the remote machine `Server2`, for *'read'* access to data provided by the DAP board. The value of the returned `hBinFromDap` handle variable is retained for later use.

```
hDapQ = dapopen('\\.\Dap0', 'query')
```

Open a handle to the DAP board identified as `Dap0` on the local machine, for querying information about that board. The handle value is returned in variable `hDapQ`.

See Also

`dapclose, dapfedpd, daplogpd, dapquery`

dappavl

Get the number of bytes space available for writing via a specified communication pipe.

```
numBytes = dappavl(handle)
```

Parameters

`handle`

Handle to a communication pipe previously opened for writing data.
`double`

Return Values

`numBytes`

Number of bytes space available if successful; -1 if function failed.
`double`

Description

This function gets the number of bytes spaces available for writing through the communication pipe previously opened in *'write'* mode, and identified by `handle`. The return value in variable `numBytes` is a lower bound on the number of bytes of available storage, independent of the data type of transferred elements. An application is always safe to write that number of data bytes to the pipe without waiting for the operation to complete.

Since the return value `numBytes` is not a rigorous lower bound, it is possible that there is much more buffering capacity than what is currently visible to the pipe. In most applications, where timing is not absolutely critical, it is better to use the `dapputm` function with a timeout to perform data transfers more efficiently.

Example

```
ret = dappavl(hBinToDap)
```

Get the number of byte spaces available in the pipe specified by handle `hBinToDap` for writing new data immediately. The returned size is returned in variable `ret`.

See Also

`dapgavl`

dappstr

Write a string of characters (command line) via a DAP communication pipe.

```
<numBytes> = dappstr(handle, cmdStr)
```

Parameters

`handle`

Handle to a communication pipe previously opened for writing data to a DAP.
`double`

`cmdStr`

Characters (command line) to send through the communication pipe.
`string`

Return Values

`numBytes`

Optional. The number of bytes sent to the target pipe if successful; 0 if function failed.
`double`

Description

This function writes the characters from string `cmdStr` to the target com pipe specifies by `handle.` At the end of the string, this function attaches a CR (carriage return) character to indicate the termination of a line before delivering the text to the transfer pipe. Each character written corresponds to one byte. The number of bytes written is stored in the optional return argument `numBytes`, counting one additional byte for the line termination character. If no return argument is given, any failure is an error condition that will terminate the MATLAB application with an error message.

The most common use of this function is to send commands directly to the DAPL command line interpreter. However, other communication pipes can be set up to send various text messages directly to custom command tasks.

See the `dapgstr` function for a discussion of text lines and termination characters.

Examples

```
dappstr(hTextToDap, ‘start’)
```

Send a `START` command to the DAPL system, to begin processing in a configuration downloaded previously. To send commands to the DAPL system, the reserved `$SysIn` communication pipe must be specified when using the `dapopen` function to prepare the handle variable `hTextToDap` for *'write'* access.

```
   ret = dappstr (hCustomTextIn, ‘Press5: 10050’)
```

Send a text command line directly to a custom command task. The communication pipe for performing this text transfer must be previously created and configured, and opened for *`'write'`* mode operation. The command line `'Press5: 10050'` is meaningful only to custom command processing. The custom command task must be specified in the DAPL configuration script and started before performing the data transfers. The number of byte written, including the termination character added at the end of the line, is reported in the return argument `ret`. For this particular message, there are 10 text characters plus 1 more for the termination, so the value of `ret` is 11 if the message transfer is successful.

See Also

`dapgstr`

dapputm

Put a matrix of data to a communication pipe.

```
<numData> = dapputm(handle, dataM, dataType <, timeWait, timeOut>)
```

Parameters

`handle`

Handle to a commuication pipe, previously opened in *'write'* mode.
`double`

`dataM`

A MATLAB data matrix to transfer.
`matrix of double`

`dataType`

A keyword specifying the data type the DAP is to receive.
`string keyword`, must be one of: *'int16'*, *'int32'*, *'double'* or *'float'*

`timewait`

Optional time limit to wait for storage to become available, or 0 to use the default.
`double`

`timeout`

Optional time limit for the entire transfer operation to complete, or 0 to use default.
`double`

Return Values

`numData`

Optional. If specified, this variable reports the number of values sent to the target pipe when the transfer is successful or timed out; or reports -1 for failures.
`double`

Description

This function transfers the data from matrix `dataM`, converts it into the data type specified by the `dataType` argument, and places the data into the communication pipe associated with `handle`. The size of the matrix is determined automatically from the object properties. The matrix is transferred column by column.

The `dataType` keyword string specifying the data type must be one of the following.

'int16'	specifies 16-bit integer, represented by two bytes
'int33'	specifies 32-bit integer, represented by four bytes
'double'	specifies double, represented by eight bytes
'float'	specifies floating point data, represented by four bytes

This `dapputm` function provides some options for timing control. The two timing options `timewait` and `timeout` must either be both omitted, or both specified. If the transfer pipe cannot accept any new data for a period of `timewait` milliseconds, or if the entire transfer operation fails to complete within `timeout` milliseconds, the function returns immediately and reports the number of terms that were sent successfully. The default values, if you omit these arguments or specify a 0 value, are as follows.

`timewait`	default 100 milliseconds
`timeout`	default 20000 milliseonds (20 seconds)

Conversions of data types can sometimes cause truncation or range errors, so be careful to round and scale appropriately, particularly when using fixed point *'int16'* or *'int32'* data types. Many calculations are done in Matlab with well-normalized numbers, with values ranging from -1.0 to +1.0. These values will be truncated or rounded to the three discrete values -1, 0, or +1 when converted to integer form, losing almost all numeric information – probably not what was intended. Use MATLAB functions and operations to scale values so that they fit a more useful integer range before the transfer.

For example, output signal generation has relatively rigid scaling requirements. The data type for the transfer must be `'int16'.` The valid numerical range for this data type is `-32768` to `+32767,` which corresponds to the full output range of the digital-to-analog converter devices. (Sometimes, you might prefer to use a slightly more balanced range `-32767` to `+32767` range, though this rarely makes any difference.) Depending on the DAP hardware configuration options, the physical output voltage ranges are -5V to +5V, or -10V to +10V. Use the MATLAB *`floor`* or *`round`* functions prior to the transfer if you need to directly control the conversions to integer values.

The number of data written to the DAP com pipe is stored in the optional return argument `numData`. If the function terminates successfully, the return count value will equal the number of terms in the delivered matrix. If there is a timeout event, the transfer could be partial, as small as 0 but less than full size. If the function fails, the returned value `a` is `-1`. Be sure to check the return value to make sure that all of the sent data could be delivered.

Examples

```
Params = [129.9, 0.0263, 36.66, 0.038, 140.0, 8.88; ...
          12992, 26333, -32666, 38000, 14000, 888; ...
          12930, 26308, 32650, 38000, 14000, 888];

numSent  = dapputm(hBinToDap, Params, ‘double’);
```

Send a data matrix `Params` as data type `double` to the com pipe specified by `hBinToDap`. The number of matrix elements sent to the com pipe is stored in the return variable `numSent`. A `double` data type is used for the transfers, to avoid loss of precision that would result from conversions.

```
  numSent  = dapputm(hBinToDap, inputParams, ‘int32’, 5000, 0);
```

Similar to the previous example, sending the same data set but converting the values to 32-bit integers before making the transfer. Fractions will be dropped as the values are converted. If the transfer does not start within 5 seconds or finish within the default 20 seconds, the function will terminate the transfer before completion, and the number of terms delivered will then be less than the matrix size.

See Also

`dapgetm`

dapquery

Query for information about a server, DAP, or communication channel associated with a handle.

```
[queryResult <, boolean>] = dapquery(handle, ‘queryKey’)
```

Parameters

`handle`

Handle to a previously-opened DAP, com pipe or server connection.
`double`

`queryKey`

“Query key” message text required to access the desired information.
`string`

Return Values

`queryResult`

Result of the query. Data type of the return value depends on the query key.
`string` or `double`

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function queries for information associated with the source specified by `handle`.

There are four types of handles, depending on the connections associated with the handle.

1. NULL handle
   The handle variable is artificially assigned a value of 0. Only a few queries use this kind of handle. The received information not specific to any host, DAP board, or communication pipe.

2. Handle to a server PC
   The handle value comes from a `dapopen` function that specifies a UNC path to a *DAPcell Server* host, but not an individual DAP board.

3. Handle to a Data Acquisition Processor
   The handle value comes from a `dapopen` function that specifies a UNC path to a DAP board on a specific *DAPcell Server* host.

4. Handle to a communication pipe on a Data Acquisition Processor.
   The handle value comes from a `dapopen` function that specifies a UNC path to a communication pipe on a specific DAP board and *DAPcell Server* host.

The kinds of query operations that are possible depend on the handle that you provide. If you have already opened a connection to the appropriate server, DAP, or communication channel for some other purpose, using *'read'* or *'write'* mode, you can also use that handle for dapquery function calls. Otherwise, it is recommended to open a handle with the *'query'* mode, since this avoids unnecessary exclusion constraints.

The information that the `dapquery` operation returns depends on the `queryKey.` This *key* is a text containing keywords and various other notations in a string. Be careful to match the case and special delimiter characters exactly. Depending on the query, the result is reported back as a text `string`, or as a `double` numeric value.

For complete information about query keys and what they can do, please see `TDapHandleQuery` and `DapHandleQuery` sections in the *DAPIO32 Reference Manual*. For some return values that have a text form, conversions to numeric form are possible using the MATLAB function *`Str2Num.`* You can examine the return value using the MATLAB `class` function to determine which data type you are receiving.

Examples

```
hQuery = dapopen('\\.\Dap0', 'query');
...
[qResult, ret] = dapquery(hQuery, 'DapName');
```

Using a handle `hQuery` referring to a the DAP board named `Dap0` on the local host machine, with *'query'* mode access, query for identity information from that DAP board. If successful, the return value `ret` is set equal to 1, and the detailed product name and model number for the referenced DAP board will be returned as a text string in the `qResult` variable.

```
hBinFromDap = dapopen('\\.\Dap0', 'dapio');
...
qResult = dapquery(hBinFromDap, 'DapDiskIoStatus');
```

Using handle `hBinFromDap` that was previously opened for direct-to-disk data logging with the `daplogpd` function, at a later time query the status of the logging activity. If the logging is currently busy, this query will return a status string `'The disk I/O is active.'` If the logging has terminated because of reaching a size limit or timeout, this query will return a status string `'The disk I/O is completed.'`.

See Also

`dapcpc, daplogpd, dapfedpd`

daprecnf

Redirects the output of `dapcnfig` to a specified text file.

```
<boolean> = daprecnf(filepath)
```

Parameters

`filepath`

Path and file name for text. Specify an empty string to cancel redirection.
`string`

Return Value

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function allows an application developer to temporarily redirect the output of the `dapcnfig` command so that it goes to an external text file instead of the normal communication text pipe, for purposes of testing. The output file will show the command file after macro substitution processing was performed. While the redirection is in effect, the `dapcnfig` function sends no command text to the DAP board. When the testing is done, clear the redirection by calling the `daprecnf` function again, specifying an empty string for the `filepath` .

The `filepath` string specifies the file to receive the text. It can be a UNC address to a *share* file, or a local file system *path\filename* expression. The `daprecnf` function does not use a MATLAB file I/O handle, and instead, the DAPcell Server will open the file directly. The file is closed and disassociated when the redirection is cleared.

Example

```
ret = daprecnf(‘c:\Temp\redir.txt');
dapcnfig(hTextToDap, ‘Testing.dap’);
daprecnf(‘’);
```

The `daprecnf` function temporarily redirects the command text produced by the `dapcnfig` function, with the text before processing coming from the file `Testing.dap` in the current working folder. The results are written to the text file `redir.txt` in the local `c:\Temp` folder. Immediately after this test, the temporary redirection is canceled by calling `daprecnf` again with an empty string argument.

See Also

`dapcnfig`

dapreset

Resets the target DAP board(s), stopping all activity and clearing the configuration.

```
<boolean> = dapreset(handle)
```

Parameters

`handle`

Previously opened handle to a single DAP or a server with *'read'* or *'write'* access.
`double`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function performs the equivalent of a RESET command sent through a predefined `$SysIn` communication pipe, but with some subtle differences.

- Exclusive access to the DAP board is used, not just access to one of the handles, so there is no possibility that other processes can interfere with the operation.
- The DAPL system command interpreter does not receive and process any command text, so this can circumvent certain lock-up situations in which higher-priority processing blocks processing of command messages.
- The RESET operation is known to be finished when the function returns. When sending a RESET command with the `dappstr` function, you only know when the command was delivered.

The reset operation requires exclusive *'read'* or *'write'* access to the DAP board or boards. If the handle refers to a server, any DAP boards that are already reserved by other *'read'* or *'write'* processing cannot be reset. For an individual DAP board, a handle referring to that DAP must first be established using the `dapopen` function and specifying *'read'* or *'write'* access.

If this function is unable to reset the DAP board or boards as requested, it will report failure upon return. For more detailed information, see the function `DapReset` in the *DAPIO32 Reference Manual*.

Example

```
   ret  = dapreset(hDap);
```

Reset the DAP associated with handle `hDap,` which was previously opened with the *'write'* access mode.

See Also

`daprinit`

daprinit

Reloads the DAPL system and all installed modules.

```
<boolean> = daprinit(handle)
```

Parameters

`handle`

Handle to DAP board or server, previously opened with *'read'* or *'write'* access
`double`

Return Value

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function performs a hardware reset of the specified DAP board(s) and reloads the DAPL operating system, along with all installed modules registered for the board(s). This is the same as the initialization performed at system boot. It is suggested for purposes of disaster recovery only.

If argument `handle` refers to a single DAP, only this one DAP board is reloaded and restarted. If the handle refers to a server, all of the DAP boards on the host with that server will be reloaded and restarted.

This is a disruptive operation. All configured activity is abruptly terminated. All undelivered data on the target DAP board or boards is lost. Any opened handles referring to a reloaded DAP board and its communication pipes become immediately invalid. All configurations and declarations previously downloaded will be erased. All downloadable command modules not registered by installation will become unavailable. The downloading and activation of the DAPL system takes significant time.

Example

```
ret  = daprinit(hDap);
```

Cleans and reinitializes a DAP board referenced by handle `hDap`, by downloading and starting a clean copy of the DAPL system. The result is reported in the return variable `ret`.

See Also

`dapreset, dapmdin, dapmduin, dapmdld, dapmduld`

dapsetpa

Set the value of a substitution macro to be applied to configuration file downloads.

```
  <boolean> = dapsetpa(paraNumber, paraValue)
```

Parameters

`paraNumber`

A number specifying the index of the substitution macro to be set.
`double`, with integer value in the range 1 to 100

`paraValue`

The replacement text to assign to the macro.
`string`

Return Values

`boolean`

Optional return code reporting success or failure.
`double`

Description

This function assigns a text to a specified substitution macro identified by the integer index number `paraValue`. This assignment overwrites any previous default value or assigned value for identified macro. If the new substitution text is an empty string, the macro text is cleared.

Macro text substitution is applied by embedding special text notations of the form `%n` into the text of a DAPL configuration script, where `n` is the text representation of an integer number in the range 1 to 100. Later, when the script file is sent to a DAP board using the `dapcnfig` function, the characters of these special notations are replaced by characters retrieved from the macro substitution list.

This was once an interesting idea. But by today's standards, a practice of embedding unviewable process configuration fragments of uncertain persistence into a communication server should be considered archaic and potentially dangerous. There are much better text preprocessing tools available today that your applications can invoke to apply macros substitutions, using whatever notations you want, retaining full control of the command text to be downloaded. The macro substitution feature is still supported for legacy applications, but is not recommended for any new applications.

Examples

```
  dapsetpa(2, ‘’);
```

Clear any previous text value for substitution macro number 2 to an empty string in memory.

```
ret = dapsetpa(35, ‘500’);
...
dapcnfig(hTextToDap, ‘c:\Measurements\test.dap’);
```

Set the value of substitution macro number `35` to the character sequence `‘500’` in DAPserver memory. The return value `ret` reports the success or failure of this operation. Later, as the `dapcnfig` function downloads the DAPL configuration commands from the file `test.dap` in the `C:\Measurements` folder to the DAP board associated with the command text handle `hTextToDap`, each occurrence of the character sequence `%35` is replaced by the substitution text `500`.

See Also

`dapclrpa`

--- End of document ---