mdscwrap Library Documentation

Michael Bongard

26 April 2006

Abstract

The mdscwrap library is a cross-platform ANSI C binary that provides simplified access to MDSplus systems for external programs. Interfaces have been written and data structures designed for the C, LabView, and Igor Pro programming languages that enable these languages to transparently read and write data to MDSplus. Details of the programming interfaces are provided, as well as a detailed description of the internal workings of the library.

1 Introduction

MDSplus[1] has become increasingly popular with the fusion research community. It is a system that provides data acquisition, storage, and retrieval; stores data in a tree-based form; and has very good network accessibility. In addition, a C programming interface is provided with the distribution. This C interface enables other programming languages (e.g. IDL) to incorporate MDSplus functionality in their applications.

However, this C API, while functional, is not necessarily easy-to-use. Furthermore, the shared library code that implements this API, MdsLib, is not compatible with LabView on non-Windows platforms as distributed.

Mdscwrap was initially written as an effort to increase LabView support. It evolved into a set of core routines and data structures that provide simplified access to MDSplus data in LabView, C and Igor Pro.

Provided with the library is a driver program for each interface. This program, when run in conjunction with the provided test MDSplus tree, will read and write every possible combination of data types to the system, proving that the data flow is correct (to the accuracy of the tests). The driver also provides coding examples of how to use the LabView and C interfaces.

2 Getting Started

This section describes how to get your hands on a copy of mdscwrap, configure, build, and install it on Linux, OS X, and Windows platforms.

2.1 Obtaining mdscwrap

The source distribution for mdscwrap is provided as a bzip2'd tar file from the Pegasus group website: http://pegasus.ep.wisc.edu.

The extracted package has separate directories for each supported OS, as well as for the language interfaces. All the required files for building on a particular platform are in these directories. C source code of the library is in the 'src' subdirectory. LabView VI's compiled for each particular platform are provided in the /LabView subdirectory.

2.2 Build and Installation

Mdscwrap is implemented as a shared library. Depending on the platform it is installed upon, this takes the form of a shared library (Linux), Framework (OS X), or DLL (Windows). The library may be built with platform-standard build tools, namely GCC for Linux, XCode 2.2 for OS X, and Microsoft Visual Studio 2005 for Windows. The remaining sections detail specific build requirements for your platform. The source may certainly be capable of compilation on other platforms/compilers, but these are the three that have been explictly tested before release – and perhaps more importantly, for which a working build project is provided.

[1] Website: http://www.mdsplus.org

Mdscwrap must be built on top of an already-functioning MDSplus system since it links against the MdsLib C library. The build system assumes that MDSplus has been installed to /usr/local/mdsplus on OS X and Linux systems, and to C:\Program Files\MdsPlus on Windows systems.

Binary copies of the driver are provided in the /lib subdirectory of the distribution, if you wish to use them directly.

2.2.1 OS X

In the OSX subdirectory you will find an XCode 2.2 project file, mdscwrap.xcodeproj. Open this in XCode. Set the build target to 'driver', in 'Release' mode, if it isn't there already. All you should need to do is click on the 'Build' button. This will build the library, *mdscwrap.framework*, as well as a Terminal-based C program, *driver*, that validates the library's operation.

After the build is complete, you will need to manually copy the framework to a suitable location. We recommend /Library/Frameworks, although others may work. This may require root privileges.

2.2.2 Linux

In the Linux subdirectory you will find a Makefile. To build the project, change to the Linux subdirectory and type **make**. Installation requires root privileges. After becoming root, type **make install**. If you wish to remove the library, you may (as root) type **make uninstall**. And, following standard practice, all build products and intermediates may be removed via **make clean**.

By default, the installation places the shared libray *libmdscwrap.so* in /usr/local/lib; the Makefile may be edited to place it in a different location, if desired.

2.2.3 Windows

In the Windows subdirectory you will find two subdirectories, mdscwrap and driver. The 'solution' file for the project is located in mdscwrap\mdscwrap.sln. After loading the solution with Visual Studio 2005, you should be able to build a Release version of the shared library, *mdscwrap.dll*, as well as a command-line based C program, *driver.exe*, that validates the library's operation on Windows, by hitting the F7 key.

After the DLL has been built, copy it to a location of your choice, or simply leave it in the build subdirectory. Placement in C:\WINDOWS\SYSTEM32 is known to work well.

2.2.4 Igor Pro XOP

The Igor Pro language interface takes the form as an XOP binary module that extends Igor named mdsIgor. As XOPs may only be created with the assistance of the proprietary XOP Toolkit from WaveMetrics, the source or binary of the required XOPSupport library may not be provided.

However, the source code to the mdsIgor XOP itself is distributable and provided in the /Igor subdirectory of the distribution. A binary distribution of the mdsIgor XOP itself is also allowed, and is provided compiled for OS X and Windows in respective subdirectories of the /Igor subdirectory.

To install the XOP, copy mdsIgor.xop to the 'Igor Extensions' subdirectory of the Igor Pro Folder. In-Igor help is provided by copying the provided help file 'mdsIgor Help.ihf' to the same directory that the XOP resides in.

Since the source build projects for the XOP are by necessity crippled, no effort has been made to make them easily work on other systems; however, it should be a relatively straightforward process of specifying the location of the XOP Toolkit on your system.

2.3 Validation

Before you incorporate mdscwrap into your codes, it is advised that you run the included driver program. This driver will systematically read and write every data type supported by the library, arrays of every data type, and all possible combinations of Signals to the provided test tree. A command-line driver is built by default along with the mdscwrap shared library file on all platforms, and is provided for each addtional language interface as part of the code distribution.

The command line driver has the following syntax:

driver mode tree path shot [host[:port]]

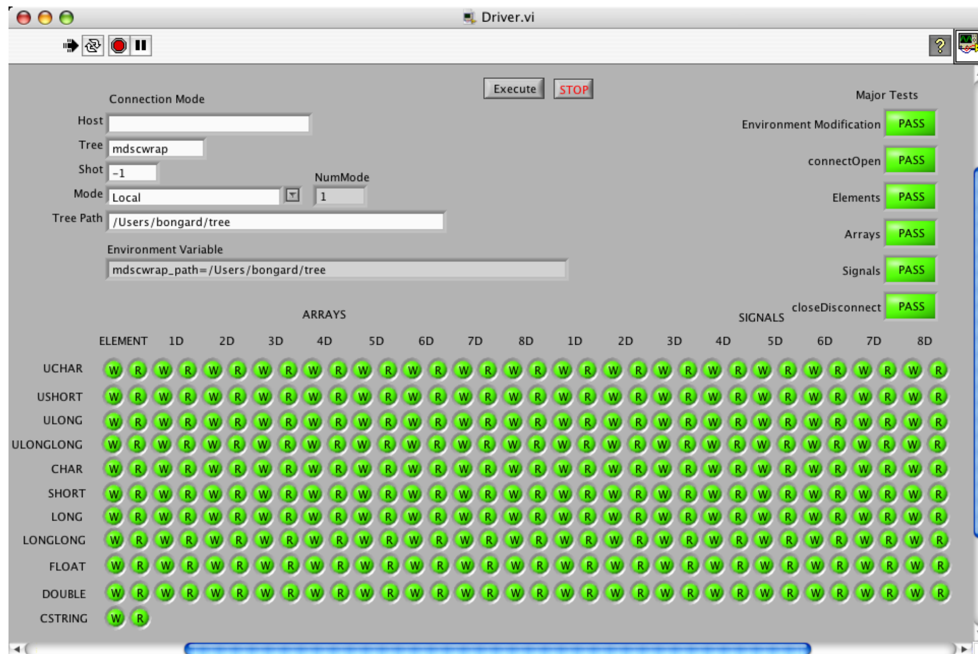

Figure 1: LabView Driver Success

where the *mode* parameter is an integer in $\{1, 2, 3, 4\}$ that defines which connection mode to the tree will be employed, corresponding to local, thin, thick, and distributed client, respectively; *tree* is the name of the tree to open; *path* is the filesystem location of the tree on the system it resides upon; *shot* is the shot number of interest; and the optional *host[:port]* are the network hostname and optional port number to connect to for non-local access.

For instance, if the provided mdscwrap tree is stored on the local filesystem at /home/mdscwrap/tree, the way to test in local mode would be to execute

driver 1 mdscwrap /home/mdscwrap/tree -1

whereas for testing distributed client mode with a mdsplus server *myHost* running on port 12345, where the tree is stored on myHosts's filesystem at /var/trees, the call would be

driver 4 mdscwrap /var/trees -1 myHost:12345

If the command line driver reports that all tests were successful, mdscwrap is good to use for your application. Similarly, a successful run of the LabView driver (Figure 1) verifies the LabView VI's are ready for use.

The Igor Pro driver is provided as an external procedure. The command-line parameters of the C program may be specified in string constants at the top of the file. The driver may simply be invoked by issuing the function *driver()*.

3 Core Functionality

This section describes the core functionality of the mdscwrap library. It begins with a description of the conventions used in MdsLib for describing fundamental C data types, as well as intrinsic MDSplus data structures and terminology such as Signals and tree nodes.

3.1 Terminology

In order to understand how mdscwrap works, one needs a working knowledge of the MdsLib library and its naming conventions, as well as MDSplus-specific constructs. The goal of this section is to provide a brief summary of these. Additional in-depth information is available through the MDSplus website.

MDSplus	C	DTYPE Identifier
Unsigned Byte	unsigned char	DTYPE_UCHAR
Unsigned Word	unsigned short	DTYPE_USHORT
Unsigned Long	unsigned long	DTYPE_ULONG
Unsigned Quadword	unsigned long long	DTYPE_ULONGLONG
Byte	char	DTYPE_CHAR
Word	short	DTYPE_SHORT
Long	long	DTYPE_LONG
Quadword	long long	DTYPE_LONGLONG
32-bit IEEE Float	float	DTYPE_FLOAT
64-bit IEEE Float	double	DTYPE_DOUBLE
Text	const char*	DTYPE_CSTRING

Table 1: MDSplus $\leftrightarrow$ C data type conventions. The DTYPE identifier is a common handle used in mdscwrap programming.

3.1.1 Data Types

MDSplus is capable of handling a wide variety of data types. A subset of these coincide with those of the C programming language. MdsLib has defined a set of numeric constants to delineate data types in several internal functions. These constants take the form DTYPE_*, with mappings shown in Table 1. Complex floating point numbers are supported by MDSplus and MdsLib; however, mdscwrap has not implemented them at this point.

3.1.2 Data Strucures

MDSplus is based on the principle that everything regarding an experimental shot is saved. This is accomplished in a comprehensible manner by using a tree-based model for storing and retrieving data. As such, when reading or writing with MDSplus, one must first open a tree. Then, specific nodes of the tree may be accessed with the read/write functions provided by mdscwrap.

Multidimensional arrays of numeric data types, up to 8-D, are supported natively by MDSplus and mdscwrap. In addition, MDSplus has an intrinsic composite data type, the Signal, which combines an array of data with arrays of time values and optional unit descriptors.

Signals time values (timebases) may either be provided by the user, or when writing to a tree, characterized in a more efficient manner for regularly spaced intervals as a *range*, the triad (start, length, delta).

3.2 Establishing MDSplus Connectivity

Having evolved over the years, MDSplus has several different modes of accessing a tree. These are:

- Local Mode – Trees are located on the machine requesting the data. No network traversal is required. Expression evaluation is performed locally.
- Thin Client – A client computer establishes a connection to a data server via a call to *MdsConnect()*. Expression evaluation is performed by the server, and results sent back to the client. After a client is finished, it calls *MdsDisconnect()* to terminate the session.
- Thick Client – A client computer connects to a data server in such a manner as it appears to the client to be a local connection. (That is, *MdsConnect()* and *MdsDisconnect()* are not required to be called by the program.) The server provides access to the data in the trees, but expression evaluation is performed by the client. All trees must be located on a single data server.
- Distributed Client – Functionally identical to the Thick Client mode, but with the option of housing trees across multiple data servers. Preferred method for remote access of MDSplus data; however, not implemented on the OpenVMS platform.

The way the MdsLib calls distinguish between these connection modes are through the use of environment variables (or Registry entries on Windows) and particular path declaration syntax which must be manually set by the user before a MDSplus-aware program is executed.

3.2.1 Programattic Environment Modification

Some programs do not have easy access to environment modification (such as LabView on OS X), so the traditional methods of specifying tree paths and network connection modes are inaccessible. The mdscwrap library provides methods that will directly access the environment (or Windows registry) and configure the appropriate environment variables (registry keys) programattically when given the requisite information. This enables MdsLib compatibility for environment-challenged applications. Specifically, the following functions are provided:

- **int getEnvironment(const char* name, char* buf, int len)**
  Effectively, a wrapper around the getenv() system call. Given an environment variable name, **name**, and a storage string buffer, **buf**, of length **len**, copies up to len-1 characters of the value of the environment variable value into buf. On Windows, the environment is not modified, but instead a name/value pair is read from the appropriate MDSplus registry location.
  State Change: The external string buffer **buf** is filled. If the environment variable does not exist, a null string is written.
  Return: 1 on success; 0 otherwise.

- **int setEnvironment(const char* name, const char* val)**
  Effectively, a wrapper around the setenv() system call. Given an environment variable name, **name**, and value, **val**, writes the name/value pair to the system environment. On Windows, the environment is not modified, but instead a name/value pair is written to the appropriate MDSplus registry location.
  State Change: Sets an environment variable.
  Return: 1 on success; 0 otherwise.

- **int delEnvironment(const char* name)**
  Effectively, a wrapper around the unsetenv() system call. Given an environment variable name, **name,** removes all instances of it from the environment. On Windows, the environment is not modified, but instead a name/value pair is deleted from the appropriate MDSplus registry location.
  State Change: Deletes environment variables if present.
  Return: 1.

- **int setMdsPathEnvVar(const char* host, const char* tree, const char* path, int mode)**
  Given MDSplus connection information, sets an appropriate environment variable depending on the mode. On Windows, the environment is not modified, but instead a name/value pair is written to the appropriate MDSplus registry location.
  Inputs:
  host – hostname where MdsConnect() should connect; ignored for non-thin client modes
  tree – name of the MDS tree
  path – location of the tree files on the local/remote host
  mode – integer in {1, 2, 3, 4}, corresponding to local, thin, thick, and distributed client modes, respectively.
  State Change: Constructs an appropriate tree path environment variable for the selected connection mode and registers it in the environment.
  Return: 1 on success; 0 otherwise or for invalid parameters.

The easiest way to set things up is to use setMdsPathEnvVar(), which simply uses appropriate combinations of the inputs to absolve the programmer (and end user!) of knowing the MDSplus environment variable syntax.

3.2.2 Simplified Connect/Open/Close/Disconnect Cycle

For a program to access a MDSplus tree via MdsLib, the following cycle is performed:

$$[\text{MdsConnect}\rightarrow]\ \text{MdsOpen} \rightarrow \cdots \rightarrow \text{MdsClose}\ [\rightarrow \text{MdsDisconnect}]$$

where the MdsConnect/MdsDisconnect calls are required if the tree is hosted on a remote server and being accessed by the client in thin mode. Otherwise, the MdsOpen/MdsClose is used for local access and thick/distributed client modes, which use special environment variable syntax to alert MdsLib to use transparent networking in handling the connection.

The mdscwrap library provides an alternative method for establishing a connection to a MDSplus tree:

$$\text{connectOpen} \rightarrow \cdots \rightarrow \text{closeDisconnect}$$

where connectOpen and closeDisconnect abstract away the connection mode decision making, as described below:

- **int connectOpen(const char* host, const char* tree, int shot, int mode)**
  Automates the MdsConnect, MdsOpen cycle. Connects to a mdsip server running at **host** via MdsConnect if necessary, and opens the specified tree, **tree**, shot number **shot**.
  Inputs:
  host – hostname where MdsConnect() should connect; ignored for non-thin client modes
  tree – name of the MDS tree
  shot – shot number to open
  mode – integer in {1, 2, 3, 4}, corresponding to local, thin, thick, and distributed client modes, respectively.
  Return: -1 on error in MdsConnect, -3 on error in MdsOpen, 1 on success.

- **int closeDisconnect(const char* tree, int shot, int mode)**
  Automates the MdsClose, MdsDisconnect cycle.
  Inputs:
  tree – name of the MDS tree
  shot – shot number to open
  mode – integer in {1, 2, 3, 4}, corresponding to local, thin, thick, and distributed client modes, respectively.
  Return: 1 on success, -1 on error in MdsClose.

3.3 Data Passing

The primary strength of the mdscwrap library is the greatly simplified ease of data storage to and retrieval from a MDSplus tree. Routines are provided which encapsulate all requisite MdsLib calls to create descriptors and evaluate TDI expressions. All a programmer requires is a knowledge of the expected data type of the evaluated expression.

To this end, mdscwrap provides generalized read/write routines for the following classes of data:

- Elementals – Singular numeric values
- Arrays – 1-8 dimensional arrays of numeric values, or singular string values (following the C convention of a string being an array of characters)
- Signals – MDSplus-provided composite data type

The associated functions and a subset of wrappers that are simpler to call are detailed in the following sections.

3.3.1 Elementals

Elemental read/write capability is provided via the following functions:

- **int getElement(const char* expr, void* element, int dtype)**
  Generalized get routine. Evaluates a TDI expression, **expr**, which should evaluate to a singular element of type **dtype**. The result is stored in the value of **element**, which is a pointer to the data type specified by **dtype**.
  Return: 1 on success; 0 otherwise.

- **int putElement(const char* node, void* data, int dtype)**
  Generalized put routine. Writes a singular element, **data**, of type **dtype** to a tree location specified by **node**.
  Return: 1 on success; -1 otherwise.

A simple code example is provided:

```
float send = 5.0;
float recv;
if(putElement("\\TOP.FLOATVAL", &send, DTYPE_FLOAT) < 0)
{
   <indicate failure to write>
}
if(!getElement("\\TOP.FLOATVAL", &recv, DTYPE_FLOAT))
{
   <indicate failure to evaluate>
}
/* recv now is equal to 5.0 */
```

The following simplified wrappers are provided to ease programming and code readability. In general, they provide the appropriate DTYPE flag to put/getElement. The put variants have identical return value semantics as putElement; the get variants return a datatype consistent with their name.

- **unsigned char getUChar(const char* expr)**
- **unsigned short getUShort(const char* expr)**
- **unsigned int getULong(const char* expr)**
- **unsigned long long getULongLong(const char* expr)**
- **char getChar(const char* expr)**
- **short getShort(const char* expr)**
- **int getLong(const char* expr)**
- **long long getLongLong(const char* expr)**
- **float getFloat(const char* expr)**
- **double getDouble(const char* expr)**
- **int putUChar(const char* node, unsigned char val)**
- **int putUShort(const char* node, unsigned short val)**
- **int putULong(const char* node, unsigned int val)**
- **int putULongLong(const char* node, unsigned long long val)**
- **int putChar(const char* node, char val)**
- **int putShort(const char* node, short val)**
- **int putLong(const char* node, int val)**
- **int putLongLong(const char* node, long long val)**
- **int putFloat(const char* node, float val)**
- **int putDouble(const char* node, double val)**

Note that the convenience of a return value of the evaluated expression for the get variants comes at the cost of error reporting via the return value. In the event of an evaluation error, the get variants will return 0 for an unsigned type, or -42 for signed and floating point data types.

3.3.2 Arrays

Array read/write capability is provided through the following functions:

- **int getArray(const char* expr, void* buf, int dim, int* shape, int dtype)**
  Generalized get routine for arrays. Evaluates an MDSplus expression, **expr**, which should evaluate to a **dim**-dimensional array. The data is copied to the pre-allocated array **buf**, which is passed as a pointer to void. **buf** must be a **dim**-dimensional array of type **dtype**, with dimension sizes indicated in the **shape** parameter, a 1D array of length **dim** where each index represents the size of the corresponding dimension in **buf**. The expression should evaluate to a data type specified by one of the DTYPE_* constants, passed with the **dtype** parameter.
  Return: 1 on success; -1 otherwise.

- **int putArray(const char* node, const void* data, int dim, int* shape, int dtype)**
  Generalized put routine for arrays. Writes an array of elements of type **dtype** to a tree location specified by **node**. **data** is a pointer to an array of data type specified by **dtype**, **dim** is the number of dimensions in the array, with **shape** being an array of size **dim** that contains the lengths of each dimension of data.
  Return: 1 on success; -1 otherwise.

To clarify the specification, mdscwrap needs to know the expected number of dimensions in the array, its data type, and the shape to properly handle things. A shape array, passed to the **shape** input of these functions, is defined in a straightforward manner: for a M by N [by O ...] array,

shape = [M, N [, O ...]]

That is, for a 3x2 array, shape[0] == 3, shape[1] == 2.

Wrappers have been provided for one dimensional array read/write. Multidimensional arrays must use the generalized put/getArray routines. The following wrappers have the same return value semantics as put/getArray:

- **int getUCharArray(const char* expr, unsigned char* buf, int len)**
- **int getUShortArray(const char* expr, unsigned short* buf, int len)**
- **int getULongArray(const char* expr, unsigned int* buf, int len)**
- **int getULongLongArray(const char* expr, unsigned long long* buf, int len)**
- **int getCharArray(const char* expr, char* buf, int len)**
- **int getShortArray(const char* expr, short* buf, int len)**
- **int getLongArray(const char* expr, int* buf, int len)**
- **int getLongLongArray(const char* expr, long long* buf, int len)**
- **int getFloatArray(const char* expr, float* buf, int len)**
- **int getDoubleArray(const char* expr, double* buf, int len)**
- **int putUCharArray(const char* node, const unsigned char* data, int len)**
- **int putUShortArray(const char* node, const unsigned short* data, int len)**
- **int putULongArray(const char* node, const unsigned int* data, int len)**
- **int putULongLongArray(const char* node, const unsigned long long* data, int len)**
- **int putCharArray(const char* node, const char* data, int len)**
- **int putShortArray(const char* node, const short* data, int len)**
- **int putLongArray(const char* node, const int* data, int len)**
- **int putLongLongArray(const char* node, const long long* data, int len)**
- **int putFloatArray(const char* node, const float* data, int len)**
- **int putDoubleArray(const char* node, const double* data, int len)**

3.3.3 Strings

Strings fall in a grey zone when mapping between MDSplus, where they are provided as an elemental, and C, where they are implemented as a null-terminated 1D array. Since this conversion requires a bit of additional manipulation, the following string functions are provided:

- **int getString(const char* expr, char* buf, int len)**
  Evaluates a TDI expression **expr**, copying the resultant string into the provided character buffer **buf**, of length **len**. getString will truncate and properly null-terminate **buf** in the event that the resultant expression is insufficiently large.
  Return: 1 on successful expression evaluation; -1 on getArray failure.

- **int putString(const char* node, const char* str)**
  Writes a string value **str** to a specified tree location **node**.
  Return: 1 on success; -1 otherwise.

where getString is treated as a call to getArray, while putString is a call to putElement. Their return values are semantically identical to the function they wrap.

3.3.4 Signals

The native MDSplus Signal data structure has no direct analog in C or the other programming languages for which mdscwrap has an interface. Components of a Signal (data arrays and strings) are supported; as such, a (complicated) function is provided which may be used to directly write a Signal to a MDSplus tree node from its constituent components. Simplified wrappers to this function are provided in the native language interfaces in a convenient language-specific data structure, but the definition is provided here for completeness:

- **int putSignalRaw(const char* node, const void* dimData, const int* dimSize, const double** timebase, const char* units, const char** timeUnits, const int* size, const int dtype, int dim, int range, int unitsAvailable)**
  Writes a MDSplus Signal composite object to a specified location in the tree given requisite constituent components.
  Inputs:
  node – Node to write resultant Signal.
  dimData – Pointer to beginning of signal data. Must be a static C array of the same data type implied by the value of **dtype**.
  dimSize – Array containing the number of elements in each data dimension.
  timebase – 2D array of timebase information, consisting of **dim** pointers to double-valued timebase arrays. **timebase[i]** encodes timebase information for **dimData** dimension i+1. It may be either a simple data array of values that correspond to **dimData** or an encoded Range descriptor. This descriptor has length three for each dimension. Element 0 is the starting value, 1 the number of the resultant timebase data points, and 2 the delta (change in value between data points).
  units – String encoding the associated units of the signal data
  timeUnits – Array of strings, encoding the units of each **timebase** dimension.
  size – Array containing the number of elements in each **timebase** dimension.
  dtype – Value of DTYPE_* constant corresponding to the data type represented by **dimData**
  dim – Number of dimensions in **dimData**
  range – Flag denoting whether range information is encoded in **timebase**. 1 if so, 0 otherwise.
  unitsAvailable – Flag denoting whether unit information provided should be stored. 1 if so, 0 otherwise.
  Return: 1 on successful write; -1 on error.

Essentially, putSignal will associate an array of data, **dimData**, with a set of timebases, and optionally unit labels for the data. A timebase is considered to be a 1D double precision floating point array of time values that are associated with the values of a particular dimension of the data. Timebase information may either be provided in a raw format, or for most cases involving digitizers, in a *range* – a three-element double-precision floating point array consisting of, in order, a starting point, the number of samples, and time increment between points. (Range information makes writing

signals much more efficient for network data passing, as three values need to be written to the archive instead of N values for each dimension of a large signal.)

Use of putSignalRaw is not encouraged for languages in which an interface is provided; instead, use the provided Signal data structure and putSignal method for your language, as described in Section 4. Those functions adequately decompose the Signal data structure and appropriately call putSignalRaw for you.

A method for retrieving signals in a similar manner to putSignalRaw is not provided, as the constituent components of a signal may be retrieved from the MDSplus tree with an appropriate sequence of TDI expression evaluations. This is done in manners specific to the representations of the Signal data type in the supported language interfaces, and are described in Section 4.

4 Programming Language Interfaces

This section describes the language-specific customizations of the mdscwrap implementation. This is primarily with the representation of the MDSplus Signal data structure in the target language. Any additional functionality not common to the library (Section 3) is described.

4.1 LabView

The LabView subdirectory of the mdscwrap distribution contains VIs that are wrappers around the functions described in Section 3. Simply add the appropriate VI to your code. Details regarding the implementation of the Signal data structure, a primer on the use of Variants, as well as additional LabView-specific functions are provided in this section.

All VI's are internally documented within the excellent LabView on-demand help system for quick, easy reference. Any discrepancies in documentation should defer to the in-LabView version over this document!

4.1.1 LabView Signals

A MDSplus Signal is comprised of an array of data, a timebase array for each dimension of data, and optional units for both data and timebase. A suitable representation in LabView is through the use of the Cluster data type. Thus, two structures are defined, aptly named Signal and Timebase. They are simply LabView clusters that have a particular form, with methods provided to build and decompose them into their constituents.

Specifically, a Signal is comprised of the following:

1. A numeric array, up to 8D
2. A unit string. If no units are provided, a single space character must be wired.
3. A 1D array of Timebase data structures, with the same number of dimensions as the data array.

A Timebase may take two forms, corresponding to the Range parameter of putSignal. It may either be a raw data set, which makes a *seeded* Timebase, or a set of three parameters specifying the *range.*

After a Signal cluster has been created, it may be written like any other element via the putSignal VI (Figure 2).

A Signal cluster may be readily retrieved like any other data type via the getSignal VI (Figure 3). Access to the components is via the standard unbundle VI in conjunction with the mdscwrap getTimebase VI, which will generate a data array from a ranged Timebase or provide the seed array of a seeded Timebase. (This is necessary for Signal reads from MDSplus, which provides no mechanism for similarly *retrieving* a range.)

Finally, Signal equality may be tested with the provided signalEqual VI.

4.1.2 Variants

The LabView interface makes heavy use of the LabView Variant data type. This allows a VI to accept or return arbitrary data, at the expense of a bit of extra programming hassle. This comes about from turning the retrieved Variant data type back into a LabView-accessible entity. To do this, you need to manually create an object of the same type as the data (e.g. a 3D array of doubles), which you don't necessarily know offhand! An example is shown in Figure 4.

Variant casting is required for retrieval of data for all multidimensional arrays, extracting Signal data, and the outputs of the getElement and getArray VI's.

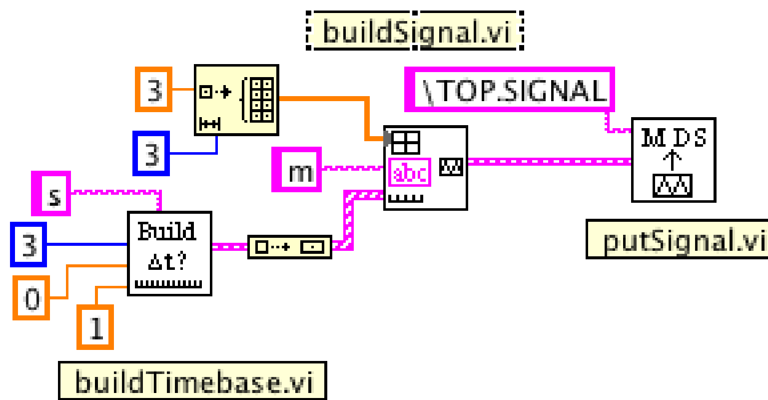

Figure 2: Simple Signal Construction and write. A 1D Signal with units and a ranged Timebase is constructed and written to the node \TOP.SIGNAL.

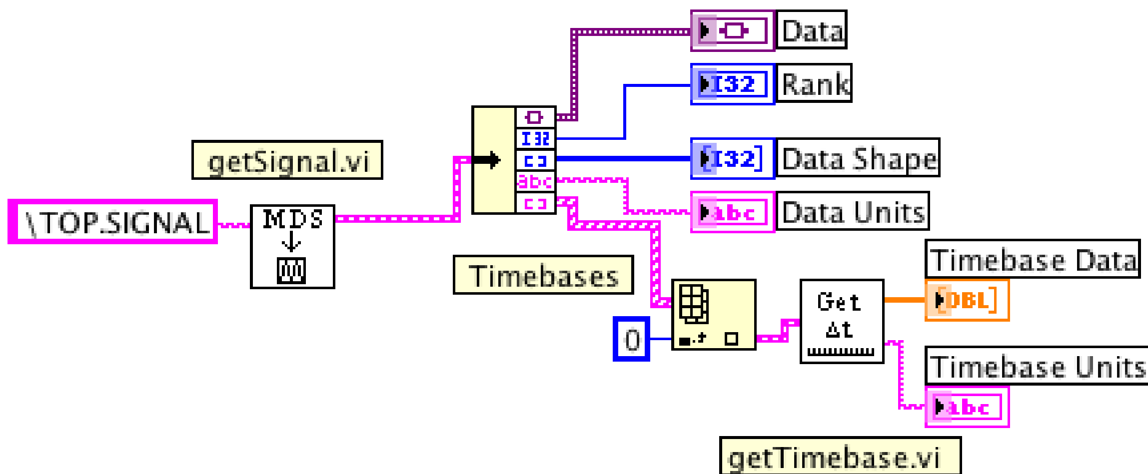

Figure 3: Sample Signal retrieval and decomposition, following the example of Figure 2.

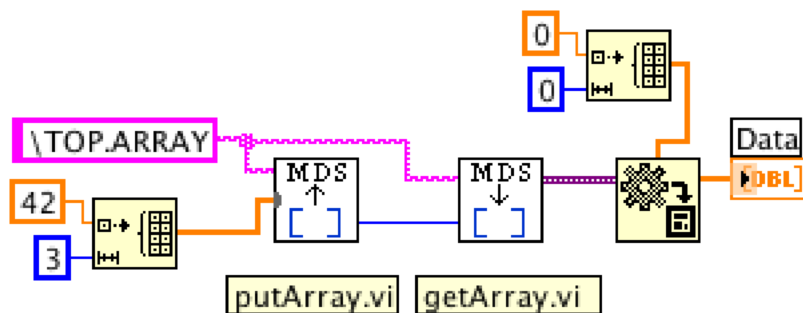

Figure 4: Simple Variant casting example. In principle, getDataType.vi may be used to probe the data type, rank, etc. for unknown sizes and types.

In order to ease the determination of the data type of a LabView Variant item, the getDataType VI was written, since LabView provides no built-in mechanism of probing these data types. It works by flattening the Variant to a string and subsequently parsing the type codes in the first few bytes of the string. Type codes are provided in the LabView help, but are listed in Table 2 of Section 5.2 for reference.

4.1.3 Utility Routines

Two additional routines are provided that are unique to the LabView platform. They relate to the determination of data types and translating them into MDplus DTYPE identifiers (Table 2) :

- getDataType.vi – Determines the data type of a LabView wire, returning a human-readable string and array information, if warranted. Capable of discerning arrays of up to 8 dimensions.
- getDtype.vi – Given a character string describing a data type (such as that produced by getDataType.vi), obtains the approprate DTYPE_* descriptor for this architecture, as supplied by MDSplus.

4.2 C

The C interface is built into the mdscwrap library. To use the Signal representation in your code, include mdscwrap.h from the distribution's src directory and link your program against the mdscwrap shared library.

4.2.1 Signal Representation

The Signal representation of choice for C is the struct. Specifically, the C Signal has the following form:

```
struct Signal {
   void* data;  /* mallocMD() */
   int* shape;
   char* units;
   double** timebase; /* mallocMD() if range=1 */
   int* timeShape;
   char** timeUnits;
   int dtype;
   int dim;
   int range;
};
```

Effectively, all the data in the Signal structure are the components of a call to putSignalRaw() (Section 3.3.4). The comments regarding the allocation of the **data** and potentially the **timebase** parameters via mallocMD are provided so that programmers wishing to manually create Signal objects know how to allocate components in such a way that a memory leak will not occur. Additional details regarding the memory management may be found in Section 5.1.

4.2.2 Signal Creation

By far, the best way to create and use Signals is by using the provided constructor and destructor methods:

- **struct Signal* buildSignal(int dim, int* shape, char* units, double** timebase, int* timeShape, char** timeUnits, int dtype, int rang**e)
  Constructs a Signal from its constituent parts.
  Inputs:
  dim – Number of dimensions in the signal data; must be between 1 and 8, inclusive
  shape – 1D array specifying the number of elements in each dimension of data
  units – Units of the signal data
  timebase – 1D array of timebases, as in putSignal(); see definition of putSignal
  timeShape – 1D array specifying the number of elements in each dimension of the timebases
  timeUnits – 1D array of timebase units, as in putSignal(); see definition of putSignal

dtype – DTYPE_* identifier specifying the type of data to allocate for the signal
range – 0 or 1 depending on whether **timebase** encodes range information, as in putSignal; see definition of putSignal
State Change: A Signal structure is dynamically allocated and zeroed. An appropriate data block is dynamically allocated and initialized. If **units**, **timebase**, or **timeUnits** are non-NULL, their contents are copied into the Signal.
Return: A pointer to the dynamically allocated Signal object; NULL on failure.

- **void freeSignal(struct Signal* sig)**
  Properly deallocates Signal structures allocated by buildSignal().
  Inputs:
  sig – Signal object to be deallocated
  State Change: Any and all memory allocated within the Signal is deallocated, followed by the memory allocated for the Signal itself.

It should be noted again that the buildSignal function will copy its arguments into the freshly allocated Signal. After building a Signal, its parts may be used as a common C struct.

4.2.3 Signal Storage/Retrieval

Signal objects may be directly read and written to an open MDSplus tree through the use of the following functions:

- **int putSignal(const char* node, struct Signal* sig)**
  Writes the contents of a Signal object to a node in the currently opened tree.
  Inputs:
  node – Location to write Signal object
  sig – Pointer to Signal that should be written to the tree
  State Change: The contents of **sig** are properly translated into a call to putSignalRaw and written to the MDSplus tree.
  Return: -3 if bad inputs are provided; otherwise semantically identical to putSignalRaw.

- **struct Signal* getSignal(const char* expr)**
  Given a TDI expression that evaluates to a Signal, allocates and properly retrieves the Signal data.
  Inputs:
  expr – TDI expression that evaluates to a Signal
  State Change: Allocates a Signal structure object, which must be subsequently deallocated via freeSignal.
  Return: Pointer to the newly evaluated, allocated Signal; NULL on failure.

4.2.4 Memory Management Routines

Several helpful memory management routines are provided by the library. Additional detail may be found in Section 5.1.

- **void* mallocMD(int dtype, int dim, int* shape)**
  Dynamically allocates a contiguious (multidimensional) array suitible for use with MDSplus and performs requisite pointer arithmetic to allow seamless use.
  Inputs:
  dtype – DTYPE_* identifier specifying the requested data type to be allocated
  dim – The number of dimensions in the array. dim must be between 1 and 8, inclusively.
  shape – 1D array specifying the number of elements in each dimension. Ex: A 3x5x2 array of doubles would require a call evaluating to mallocMD(DTYPE_DOUBLE, 3, [3,5,2])
  Return: The pointer to the allocated array; NULL on failure.

- **int freeMD(void* data, int dtype, int dim, int* shape)**
  Correctly deallocates a dynamically-allocated (multidimensional) array created by mallocMD.
  Inputs:

data – Value returned from mallocMD() to deallocate
dtype, **dim**, **shape** – Same values as in mallocMD
Return: 1 on success; -1 otherwise

The following functions allow static C arrays to be 'wrapped' in a pointer header structure that enables compatibility with some of the prior functions which require use of mallocMD arrays.

- **void* staticArrayWrap(void* data, int dtype, int dim, int* shape)**
  Given a multidimensional, statically allocated C array, **data**, dynamically allocate an appropriate set of header pointers such that the semantics of the return pointer is that of a set of pointers to pointers. Must be subsequently freed by staticArrayUnwrap(). Practically, functions that require a mallocMD() array may use a wrapped static array.
  Inputs:
  data – Address of a statically allocated multidimensional C array
  dtype – DTYPE_* identifier specifying the requested data type to be allocated
  dim – The number of dimensions in the array. dim must be between 1 and 8, inclusively.
  shape – 1D array specifying the number of elements in each dimension.
  Return: Pointer to the new header; NULL on failure.

- **void staticArrayUnwrap(void* data, int dtype, int dim, int* shape)**
  Properly deallocates the header structure generated by staticArrayWrap().
  Inputs:
  data – Array wrapped with header structure; return of staticArrayWrap()
  dtype – DTYPE_* identifier specifying the requested data type of the array
  dim – The number of dimensions in the array. dim must be between 1 and 8, inclusively.
  shape – 1D array specifying the number of elements in each dimension.

4.2.5 Additional Functions

Several additional functions are provided in the C interface. These are:

- **int signalEqual(struct Signal* sig1, struct Signal* sig2)**
  Equality check for Signal structures.
  Inputs:
  sig1, **sig2** – Signal structures to be compared for equality.
  Return: 1 if **sig1 == sig2**, 0 otherwise.

- **int dataArrayEqual(void* arr1, void* arr2, int dim, int* shape, int dtype)**
  Helper function for checking Array equality.
  Inputs:
  arr1, **arr2** – pointers to data arrays, both of dim-dimensions, *and allocated with mallocMD*
  dim – number of dimensions in **arr1**, **arr2**; must be between 1 and 8, inclusive
  shape – 1D array specifying the number of elements in each array dimension
  dtype – DTYPE_* identifier specifying the type of data storage in **arr1**, **arr2**
  Return: 1 if the data in **arr1** is identical to that in **arr2**; 0 otherwise

- **void* getMDData(void* data, int dtype, int dim)**
  Provides the memory address of the first element of the contiguous data block of a mallocMD array. This enables the use of mallocMD arrays with functions expecting to act on a static C array.
  Inputs:
  data – Pointer to a mallocMD array
  dtype – DTYPE_* identifier specifying the type of array
  dim – Number of dimensions in the array
  Return: Pointer to the data block of the mallocMD array.

- **void* getSigData(stuct Signal* sig)**
  Wrapper of getMDData for Signal structures; provides the memory address of the first element of the contiguous data block of the Signal's data array.
  Inputs:
  sig – Pointer to Signal from which the data block will be extracted
  Return: Pointer to the data block of the Signal's data array.

- **int getSigDataSize(struct Signal* sig)**
  Determines the size, in bytes, of the data block in a given Signal.
  Inputs:
  sig – Signal from which to determine size of data block
  Return: Size of data array of **sig**; -1 on error

- **char* getUnits(const char* expr)**
  Given a TDI expression, **expr**, dynamically allocates a C string that contains the units of the expression, if any; i.e. UNITS_OF(expr) → non-single-space return. Return is NULL if no units are present or an error. Note: A non-NULL return must be subsequently free()'d or a memory leak will occur!

- **int* getShape(const char* expr)**
  Given a TDI expression, **expr**, allocates an integer array of sufficient size and returns the result of evaluating SHAPE(expr). Returns the allocated array, or NULL on failure. Note: A non-NULL return must be subsequently free()'d or a memory leak will occur!

4.3 Igor Pro

Access to slightly modified mdscwrap calls is possible through the use of the mdsIgor XOP module. For reasons of required name uniqueness, all calls to functions in the XOP have the prefix MDS_.

4.3.1 Environment Modification

The following wrappers are provided to enable programattic environment variable modification. It has been noted that unexpected results may be encountered in the repeated use of these routines in an Igor function; however, sequential commands in the Igor terminal or use though an Igor Macro work properly. (It is for this reason that the testEnvironment section of the Igor driver is implemented as a macro.)

- **MDS_getEnvironment(name, len)**
  Wrapper around getEnvironment; provides the value of an environment variable, if any.
  Inputs:
  String name – Environment variable to probe
  Variable len – Maximum number of characters to copy from environment
  Return: String; the value of the environment variable **name**; an empty string if **name** did not exist.

- **MDS_setEnvironment(name, val)**
  Wrapper around setEnvironment; sets an environment variable.
  Inputs:
  String name – Environment variable to set
  String val – Value of newly-set environment variable
  Return: Variable; 1 on success; 0 otherwise.

- **MDS_delEnvironment(name)**
  Wrapper around delEnvironment; removes an environment variable from the environment.
  Inputs:
  String name – Environment variable to remove
  Return: 1

4.3.2 Simplified Connect/Open/Close/Disconnect Cycle

- **MDS_connectOpen(host, tree, shot, mode)**
  Automates the MdsConnect, MdsOpen cycle. Connects to a mdsip server running at host via MdsConnect if necessary, and opens the specified tree, tree, shot number shot.
  Inputs:
  String host – hostname where MdsConnect() should connect; ignored for non-thin client modes
  String tree – name of the MDS tree
  Variable shot – shot number to open
  Variable mode – integer in {1, 2, 3, 4}, corresponding to local, thin, thick, and distributed client modes, respectively.
  Return: -1 on error in MdsConnect, -3 on error in MdsOpen, 1 on success.

- **MDS_closeDisconnect(tree,shot,mode)**
  Automates the MdsClose, MdsDisconnect cycle.
  Inputs:
  String tree – name of the MDS tree
  Variable shot – shot number to open
  Variable mode – integer in {1, 2, 3, 4}, corresponding to local, thin, thick, and distributed client modes, respectively.
  Return: 1 on success, -1 on error in MdsClose.

4.3.3 Elementals

- **MDS_getUChar(expr)**
  Evaluates a TDI expression that should return an unsigned character and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or 0 in the event of failure.

- **MDS_getUShort(expr)**
  Evaluates a TDI expression that should return an unsigned short and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or 0 in the event of failure.

- **MDS_getULong(expr)**
  Evaluates a TDI expression that should return an unsigned long and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or 0 in the event of failure.

- **MDS_getULongLong(expr)**
  Evaluates a TDI expression that should return an unsigned long long and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or 0 in the event of failure.

- **MDS_getChar(expr)**
  Evaluates a TDI expression that should return a character and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or -42 in the event of failure.

- **MDS_getShort(expr)**
  Evaluates a TDI expression that should return a short and returns the value to Igor.
  Inputs:

String expr – TDI expression to evaluate
Return: The evaluated expression or -42 in the event of failure.

- **MDS_getLong(expr)**
  Evaluates a TDI expression that should return a long and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or -42 in the event of failure.

- **MDS_getLongLong(expr)**
  Evaluates a TDI expression that should return a long long and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or -42 in the event of failure.

- **MDS_getFloat(expr)**
  Evaluates a TDI expression that should return a float and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or -42 in the event of failure.

- **MDS_getDouble(expr)**
  Evaluates a TDI expression that should return a double and returns the value to Igor.
  Inputs:
  String expr – TDI expression to evaluate
  Return: The evaluated expression or -42 in the event of failure.

- **MDS_putUChar(node, val)**
  Stores a single value as an unsigned character at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putUShort(node, val)**
  Stores a single value as an unsigned short at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putULong(node, val)**
  Stores a single value as an unsigned long at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putULongLong(node, val)**
  Stores a single value as an unsigned long long at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putChar(node, val)**
  Stores a single value as a character at a specified location in the currently opened tree.

Inputs:
String node – Node in the currently opened tree where **val** should be stored
Variable val – Value to insert into the tree
Return: 1 on success; -1 otherwise.

- **MDS_putShort(node, val)**
  Stores a single value as a short at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putLong(node, val)**
  Stores a single value as a long at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putLongLong(node, val)**
  Stores a single value as a long long at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putFloat(node, val)**
  Stores a single value as a float at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

- **MDS_putDouble(node, val)**
  Stores a single value as a double at a specified location in the currently opened tree.
  Inputs:
  String node – Node in the currently opened tree where **val** should be stored
  Variable val – Value to insert into the tree
  Return: 1 on success; -1 otherwise.

4.3.4 Arrays

Array access is provided both via generalized put/get routines as well as datatype-specific functions which do not require the use of a supplied DTYPE identifier. Unlike the C and LabView interfaces, the datatype variants are compatible with multidimensional arrays.

- **MDS_putArray(node, data, dtype)**
  Stores the contents of a wave as an MDSplus array in the currently opened tree. *Use of a datatype-specific putArray variant is recommended.*
  Inputs:
  String node – Node in the currently opened tree where **data** should be stored
  Wave data – Wave reference to the data that should be placed in the tree
  Variable dtype – C DTYPE_* specifier for the data type that **data** should be recorded as in the tree.
  Return: 1 on success; -1 otherwise.

- **MDS_getArray(expr, name, dtype)**
  Retrieves the contents of a MDSplus array in the currently opened tree and places it in a user-specified Igor

wave. *Use of a datatype-specific getArray variant is recommended.*
Inputs:
String expr – TDI expression that should evaluate to an array
Wave name – Wave reference to the destination data location. The contents of the wave referenced by **name** will be completely overwritten.
Variable dtype – C DTYPE_* specifier for the data type that should be retrieved from the tree.
Return: 1 on success; -1 otherwise.

The variants listed below are functionally identical to put/getArray, without the requirement of specifying the DTYPE identifier:

- **MDS_putUCharArray**
- **MDS_putUShortArray**
- **MDS_putULongArray**
- **MDS_putULongLongArray**
- **MDS_putCharArray**
- **MDS_putShortArray**
- **MDS_putLongArray**
- **MDS_putLongLongArray**
- **MDS_putFloatArray**
- **MDS_putDoubleArray**
- **MDS_getUCharArray**
- **MDS_getUShortArray**
- **MDS_getULongArray**
- **MDS_getULongLongArray**
- **MDS_getCharArray**
- **MDS_getShortArray**
- **MDS_getLongArray**
- **MDS_getLongLongArray**
- **MDS_getFloatArray**
- **MDS_getDoubleArray**

4.3.5 Signals

Igor waves are well-suited to be used as the mapping of the MDSplus Signal object, subject to the same rank constraint of arrays. Units and intrinsic scaling are fundamental to the datatype, providing a direct mapping to a Signal comprised of a data array, optional units, and a timebase.

Raw XY signals are less of an ideal map; however, they may be simulated with little difficulty by using a two wave pair, one consisting of the Signal data and the other being the corresponding timebase dimensions. This pair is named *basenameX*, *basenameY*. If units are present, they will be defined only for the data dimensions, so that when plotted in XY mode, units are properly accounted for on the axes.

Signal management is provided with the following functions:

- **MDS_getSignal(expr, name)**
  Retrieves the contents of a Signal from the currently opened tree and creates requisite waves in the current data folder as appropriate to house its contents. Created waves are double-precision floating point, regardless of the data type of the source, following Igor convention.
  Inputs:
  String expr – TDI expression that should evaluate to a Signal
  String name – Requested basename for the retrieved data. If a ranged wave is evaluated, a single wave with name identical to **name** will be created in the current data folder; otherwise, a pair of waves named **name**X, **name**Y will be created.
  Return: 1 on success; -1 otherwise.

- **MDS_putSignal(node, name, dtype)**
  Translates a specified Igor wave or Signal wave pair into an MDSplus Signal object and writes it to a specified location in the currently opened tree. If units are specified in any portion of the source waves, the Signal is constructed with units.
  Inputs:
  String node – Tree location where the Signal will be stored
  String name – Basename for the source waves in the current data folder. Ranged waves with identical basenames as non-ranged waves will have precedence in placement.
  Variable dtype – C DTYPE_* specifier for the data type that **data** should be recorded as in the tree.
  Return: 1 on success; -1 otherwise.

4.3.6 Omitted Functions

Interfaces to putElement and getElement are not provided, as the elemental wrapper functions provide the requisite functionality. This is also due to differences in passing void pointers (particularly strings) between Igor Pro and external C code.

5 Detailed Implementation Details

This section is devoted to more in-depth discussion of the implementation of the functions described in Section 4.

5.1 Memory Management

The mdscwrap library needs to deal with arbitrary, multidimentional arrays. This poses a problem since the C language is primarily static; that is, multidimensional array references via the [] operator are translated into offsets from the initial data block by the compiler. Thus, if a function is handed only a pointer, *arr*, to a known multidimensional array, an expression such as

```
arr[1][2] = 3;
```

is ill defined, since the compiler cannot know the requisite offsets and pointer arithmetic for arbitrary shapes of *arr*. To circumvent this problem, the mdscwrap library takes the approach of 'arrays of pointers to arrays' to effectively 'wrap' a statically allocated contiguous block of memory with a pointer heirarchy. (Figure 5) This heirarchy is generated by a call to a library static function, ptrAlloc, that uses the following information:

1. Multidimensional array shape
2. Address of first data element
3. Data type of static array

With that information in hand, ptrAlloc may recursively allocate a sufficient number of pointer arrays and perform the requisite pointer arithmetic to create the structure of Figure 5.

This action by ptrAlloc is the lion's share of the work of the staticArrayWrap and mallocMD functions, as an individual code path needs to be provided for each particular data type and dimension for the recursion to work

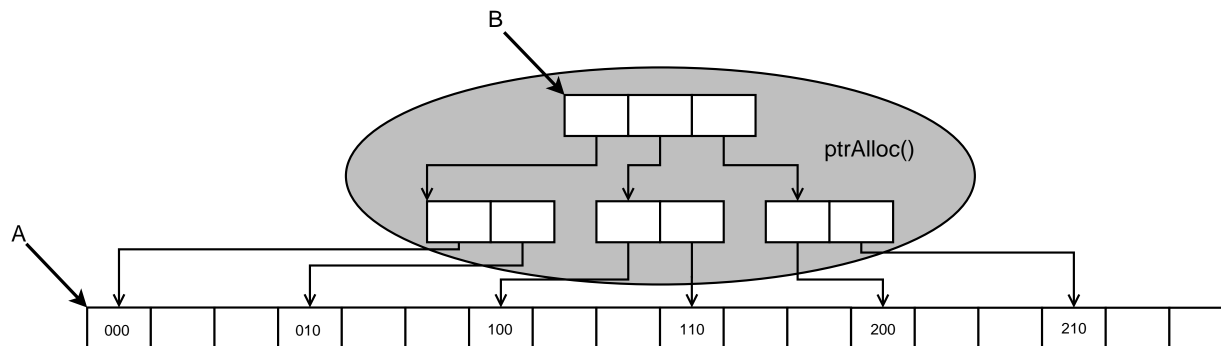

Figure 5: Memory management heirarchy for a 3x2x3 array. Pointer A is the static C array address; pointer B is provided by mallocMD or staticArrayWrap. Note that B[0][0] == A.

DTYPE	TDI KIND	C DTYPE	LabView 8.0
UCHAR	2	2	5
USHORT	3	3	6
ULONG	4	4	7
ULONGLONG	5	5	8
CHAR	6	6	1
SHORT	7	7	2
LONG	8	8	3
LONGLONG	9	9	4
FLOAT	52	10	9
DOUBLE	53	11	10
CSTRING	14	14	48

Table 2: Datatype encodings for several languages.

properly. (Perhaps some C preprocessor magic could eliminate a few lines of code in a future revision.) The only difference betweeen the two functions is where the original static array block comes from. MallocMD allocates its own, while staticArrayWrap is provided with an existing array address. The heirarchy of Figure 5 is safely deallocated by ptrFree, another static library function. StaticArrayUnwrap is a public interface to ptrFree; similarly, freeMD will additionally free the data array.

The getData functions which act on a mallocMD array pointer *arr* work by returning element *arr[0][0]...[0],* which can be seen to be equivalent to the static array address A in Figure 5. This simple conceptual act is obfuscated in the code by the necessity of properly accounting for the data type of the array.

5.2 Data Type Determination

One common task that is necessary when passing data through a strongly typed languge such as C is the determination of the data type of an expression. The TDI language provides a mechanism for probing this information, namely, the KIND function, which provides an integer which enumerates the type. Unfortunately, the return of KIND does not always match the appropriate DTYPE_* identifier used in the MdsLib library! In a similar fashion, the LabView programming environment provides a mechanism (albeit computationally costly) to obtain a similar integer data type encoding (Section 4.1.3). These encodings are provided in Table 2 for reference.

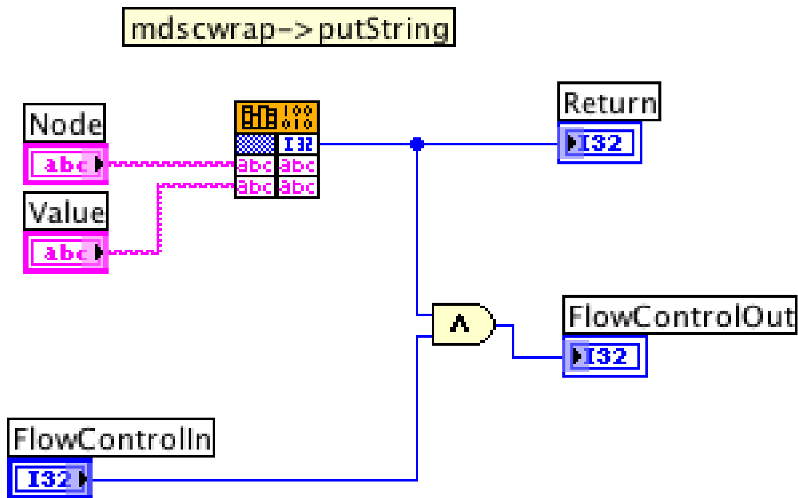

Figure 6: Simple use of a Call Library Node VI.

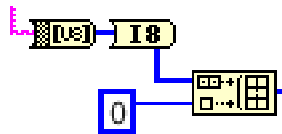

Figure 7: Manual String conversion and null-termination.

5.3 LabView ↔ C Data Passing

LabView has two well-defined interfaces for passing data to an external C library. The method chosen for use with the mdscwrap VI's is through the use of a Call Library Node VI (Figure 6). The use is relatively straightforward and documented in the LabView help system. It becomes easy to pass simple data types and 1D arrays of these data types between LabView and external code. There exists a mechanism to convert a LabView string to a null-terminated C string as well.

Things become slightly less documented and confusing when multidimensional arrays are passed to external code via LabView. It turns out that the LabView data is presented as a statically allocated C array; that is, a single contiguous chunk of memory laid out similarly to the data array of Figure 5. This works fine for some functions, but in the particular case of passing Signal components to putSignalRaw, parsing similar to that of ptrAlloc is performed to create the header structure and deconvolute the 2D array of strings of unit information. This is done on both sides of the transaction, by passing a 2D array of manually null-terminated strings to C (Figure 7) and then processing them (along with the other header scanning and parsing) in putSignalLV.