CARMEN PROJECT

Code Analysis, Repository & Modelling for e-Neuroscience

# THE MATLAB NDF TOOLBOX USER GUIDE

**Document History**

| Status | Date | Reason for Release | Authors |
|---|---|---|---|
| V 1.0 | 08.07.2010 | Initial release of the document. | Dr. Bojian Liang |

## Table of Contents

# 1. Introduction

The Neurophysiology Data translation Format (NDF) is a data format developed within the CARMEN project [1]. NDF provides a standard for sharing data, specifically for data transfer between the various analysis software applications / services in the CARMEN system. NDF can be also a useful data format on a researcher’s desktop.

An NDF dataset consists of a configuration file in XML format which contains metadata and references to the associated host data files. Using a separate header allows services or users to extract the necessary information about the data set from a relatively small header file without need to download the full data set over the network. This also provides means for the portal to display detailed metadata of a remote NDF data set without needing to access the binary data files remotely. NDF specifies a set of the most commonly used experimental data entities as “NDF internal data types". NDF can include images and image series data as a basic NDF data type. NDF also supports annotation/marker event data in XML format as a special experimental event data type.

Data processing services can output data types that may not be represented by formats used for primary data. NDF provides two extendable “semi-defined" data types for applications to create (or use) a new data type as its output (or input). The configuration file provides seamless access to these different representations based on the applications used to read the data.

A special XML data element, ***History,*** is included within the header file for recording data processing history. This element contains the full history (recording chain) of previous processing and provides important information for both users and machines to enable understanding of a particular experiment, to validate the algorithms and for other researchers to accurately repeat a reported experiment.

The NDF API has been implemented as a C library. The NDF API translates the XML tree/nodes to C style data structures and insulates the data structures within the binary data file from the clients. The NDF API also provides a standard way for data structure memory management for NDF data application programming.

The NDF MATLAB toolbox is an implementation on top of the NDF data I/O API. It provides high level support for NDF data I/O within the MatLab environment. The toolbox contains a set of object oriented MatLab classes. These include:

- Two main data I/O classes, **ndfread** and **ndfwrite**, providing the required NDF read/write support.
- One derived data I/O class **ndfsvcread** provides NDF data reading as well as CARMEN services input arguments parser functionalities.
- Eight NDF data info classes act as carriers to take metadata from an **ndfread** object or insert metadata into an **ndfwrite** object.
- Five NDF data classes act like the data info classes but are used for numeric data.
- An NDF history class for recording the previous data processing history chain.

All NDF classes are designed as *handle classes*. This manual assumes that the user understands about object oriented programming in MatLab as well as the differences between the MatLab value class and handle class.

The numeric data of the NDF internal data types are stored as MAT file format. Details of the MAT file format can be found in document [2]. All NDF numeric data used in MatLab environment are MatLab native numeric classes. The following conventions are applied:

- All one-dimension data values are in n-by-1 arrays.
- For fixed length segment data, segments are represented as column vectors, i.e. an ***m***-by-***n*** array represents ***n*** segments each with length ***m***.
- For variable length segment data, segments are stored in a **k**-by-**1** array, one segment after another with the time-offset of each segment in ascending order, where **k** is the sum of all data points of all the segments in the array. An additional **n**-by-**1** matrix is used to identify the end points of each segment whilst the first segment starts from the beginning of the array.
- Data read from NDF data set by the NDF-toolkit maintain their original data types in order to minimize memory usage. The real data values must take into account the analogue-to-digital (ADC) settings (for analogue data) or time resolution (for time stamps).

This document describes how to use the NDF toolbox for NDF data I/O within the MatLab environment. This chapter provides an overview of the NDF Data format and the structures of the NDF data I/O toolbox. The next four chapters describe classes defined within the NDF toolbox. The following chapter introduces the use of an embedded NDF multiple formats input module. The last chapter explains how to use the NDF toolbox for NDF data I/O in the MatLab environment by providing some examples.

# 2. The NDF I/O Classes

The NDF toolbox consists of three data I/O classes:

- The NDF data input class -- **ndfread**
- The NDF data output class -- **ndfwrite**
- The NDF data input class with service arguments parser -- **ndfsvcread**,

All the three NDF Data I/O classes use a common set of constant strings as identifiers for different NDF data types. These constants can be viewed using the method **objName.ndftypes**, where **objName** is the object name of one of the three NDF data I/O classes. The following table shows the constant names and the data type strings.

**Table 1. Data Type strings and Constant Names**

| Constants Name | Data Type String | Descriptions |
|---|---|---|
| TIMESERIES | 'timeseries' | Identifier for NDF data types **ndftimeseriesdata** and **ndftimeseriesdatainfo** |
| NEURALEVENT | 'neuralevent' | Identifier for NDF data types **ndfneuraleventdata** and **ndfneuraleventdatainfo** |
| EVENT | 'event' | Identifier for NDF data types **ndfeventdata** and **ndfeventdatainfo** |
| SEGMENT | 'segment' | Identifier for NDF data types **ndfsegmentdata** and **ndfsegmentdatainfo** |
| GMATRIX | 'matrix' | Identifier for NDF data types **ndfgenericmatrixdata** and **ndfgenericmatrixdatainfo** |
| USERDEFINED | 'userdefined' | Identifier for NDF data type **ndfuserdefineddatainfo** |
| IMAGE | 'image' | Identifier for NDF data types **ndfimagedata** and **ndfimagedatainfo** |

## 2.1. The NDF Input class

The NDF input class **ndfread** provides functionalities for NDF data and metadata input. On creation, the constructor reads information from the NDF header file. The object can only be created if a valid NDF header file is specified. Further information about the data can be extracted using member functions (methods) of this class. All data members in the class are read-only. The object doesn't read binary data of an NDF data set into memory. The binary data are read using methods provided by the object and assigned to an NDF data object on return.

### 2.1.1. The NDF input Class constructor

The NDF input object is created by providing a valid path of an NDF header file to the class constructor:

***A*** **= ndfread(*‘ndf_filename’*)**

where ***‘ndf_filename’*** is the NDF header file path. This will create a read-only NDF data input object ‘***A***’. On creation, the header file is interpreted and read into memory. General data info and the number of channels of each NDF data type are displayed by default. The object manages all data information extraction and data reading operations by the relevant methods. The main information can also be displayed by calling the **disp** method:

***A*****.disp();**

This command displays all available stream types and channel count of each data type. A set of "get" methods are used to extract metadata from the object.

### 2.1.2. The NDF Input Class Properties

Properties of a MatLab class are similar to data members of a C++ class. The properties of the NDF input class include a list of data info structures to provide information for the different types of channel and also the other metadata. Data info about the NDF data set can only be accessed using relevant ’get_datainfo‘ methods. These ‘get_datainfo’ methods are discussed in the next section. The other properties may be accessed directly. Properties that can be directly read without need to use the "get" methods include:

**Table 2. Public Accessible Properties**

| **Property Name** | **Description** | **Data Type** |
|---|---|---|
| ndfVersion | The NDF specification version | MatLab char |
| dataID | ID of the NDF data set. | MatLab char |
| ndfFilename | The NDF header file name | MatLab char |
| ndfFileDir | Directory contains the header file and data set | MatLab char |
| ChannelCount | The NDF data channel count list of different types | MatLab structure |
| history | The NDF history data record chain | **ndfhistory** |

These properties can be accessed by **objectName.propertyName**. For example, command

***d* = A.ndfFileDir**

will assign the NDF source directory name to variable ***d***. Property **ChannelCount** is a MatLab structure and contains the following fields:

**Table 3. Field Names of Property *ChannelCount***

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| imageDataCnt | Number of image data set | MatLab int32 |
| timeSeriesDataCnt | Number of time series data channels | MatLab int32 |
| segmentDataCnt | Number of segment data channels | MatLab int32 |
| neuralEventDataCnt | Number of spike time data channels | MatLab int32 |
| eventDataCnt | Number of experimental event data channels | MatLab int32 |
| genericMatrixDataCnt | Number of generic matrix data set | MatLab int32 |
| userDefinedDataCnt | Number of user defined data channels | MatLab int32 |

Each field can be accessed by using **objectName.ChannelCount.fieldName**. For example,

***cnt* = A.ChannelCount.timeSeriesDataCnt**

will assign the number of time series data channels to variable ***cnt***. In addition to access property **ChannelCount** directly, method **channelcnt( typestring)** can also be used to get the channel count of a specified data type. For example, the time series data channel count can also be obtained by:

***cnt* = A.channelcnt( A.TIMESERIES ) [or cnt = A.channelcnt( 'timeseries' ) ]**

## 2.1.3. The NDF Input Class Methods

The NDF input class provides "get" methods to extract information and data from an NDF data set. The "get" methods can be classified into three categories: the NDF "get_datainfo" methods, the NDF "get_data" methods and the "get_parameter" methods".

- The NDF input class "get_datainfo" methods extract the NDF data info of a specified data type from the NDF header file. With the except of the **getgeneraldatainfo** method, all the other "get_datainfo" methods are called in the same way with the following syntax:

  **objectName.methodName(** ***channel_num*** **).**

  For example, command

  ***info*** **=** ***A*****.gettimeseriesdatainfo( 1 );**

  will return an **ndftimeseriesdatainfo** object that contains the metadata of the first time series data channel. The number of available time series data channels is defined by the class property **A.ChannelCount.timeSeriesDataCnt.** The following table lists the available NDF "get_datainfo" methods:

**Table 4. The NDF Input Class *get-datainfo* Methods**

| Method Name | Description | Output Data Type |
|---|---|---|
| getgeneraldatainfo | Gets general info of the NDF data set | ndfgeneraldatainfo |
| gettimeseriesdatainfo | Gets data info of a specified time series data channel. | ndftimeseriesdatainfo |
| getsegmentdatainfo | Gets data info of a specified segment data channel. | ndfsegmentdatainfo |
| getneuraleventdatainfo | Gets data info of a specified neural event data channel. | ndfneuraleventdatainfo |
| geteventdatainfo | Gets data info of a specified experimental event data channel | ndfeventdatainfo |
| getgenericmatrixdatainfo | Gets data info of a specified generic matrix data channel. | ndfgenericmatrixdatainfo |
| getimagedatainfo | Gets data info of a specified image data channel. | ndfimagedatainfo |
| getuserdefineddatainfo | Gets data info of a specified user defined data channel. | ndfuserdefineddatainfo |
| getinfo | All-in-one method to get data info of a specified data type | The NDF data info object of the specified type |

The **getgeneraldatainfo** method returns the general data info of the NDF data set. The syntax of the method is as the follows:

***genInfo*** **=** ***A*****.getgeneraldatainfo();**

An all-in-one get data info method can be used to extract the data info of specified data type and channel. This method is available for all the three NDF data I/O classes. The syntax of using the **getinfo** method is,

***info*** **=** ***A*****.getinfo(** ***ndftypeString*****,** ***channel_num*** **);**

Where the ***ndftypeString*** is one of the NDF data type constant string identifier. For example, the following command also returns an **ndftimeseriesdatainfo** object of the first time series data channel:

***info* = *A*.getinfo( 'timeseries', 1);**

- The NDF input class "get_data" methods read in data chunk of specified channel and data type. Apart from the **getgenericmatrixdata** method that reads only the full data set of a given channel, the other methods support partial data reading. The NDF input class “get_data” methods return an NDF data object of the specified type, which contains the binary data read from the NDF data file as well as the relevant metadata. Details of the NDF data classes are discussed in Chapter-4.

The command syntax of calling the NDF "get_data" method are:

***data* = objectName.getMethodName( *channel_num* )**
***data* = objectName.getMethodName( *channel_num, start_idx* )**
***data* = objectName,getMethodName( *channel_num, start_idx, end_idx* );**

For example, command

***D* = *A*.gettimeseriesdata( *channel_num* );**
reads time series data from a specified channel to data object ***D*** (full channel read).

***D* = *A*.gettimeseriesdata( *channel_num, idx* );**
reads time series data chunk of a specified channel, from ***idx*** to the end of the channel, to data object ***D***.

***D* = *A*.gettimeseriesdata( *channel_num, idx1, idx2* );**
reads time series data chunk of a specified channel, from ***idx1*** to ***idx2*** (inclusive), to data object ***D***.

An all-in-one **getdata** method can also be used to get data (or data chunk) of a specified data type of a given channel:

***data* = objectName.getdata( *ndftypeString, channel_num* )**
***data* = objectName.getdata( *ndftypeString, channel_num, start_idx* )**
***data* = objectName,getdata( *ndftypeString, channel_num, start_idx, end_idx* );**

where ***ndftypeString*** is one of the NDF data type constant string identifiers. For example, if ***ndftypeString*='timeseries'** (the constant is **objectNmae.TIMESERIES),** the **getdata** method is equivalent to method **ndfgettimeseriesdata**. Using the all-in-one get methods, the above three examples can also be done by,

***D* = *A*.gettdata( 'timeseries', *channel_num* );**
reads time series data from a specified channel to data object ***D*** (full channel read).

***D* = *A*.getdata( 'timeseries', *channel_num, idx* );**

reads time series data chunk of a specified channel, from ***idx*** to the end, to data object ***D***.

***D* = *A*.getdata( 'timeseries', *channel_num, idx1, idx2* );**

reads time series data chunk of a specified channel, from ***idx1*** to ***idx2*** (inclusive), to data object ***D***.

The NDF data class **ndfuserdefineddata** is a semi-defined data type and the NDF API (hence the NDF MatLab toolbox) doesn't provide method to read the user defined data. In the current version, a method for reading image data has not been implemented for the NDF toolbox. There are quite a few software packages that can read the standard image formats. Thus, the "get_data" methods only apply to the five internal NDF data types. The available NDF "get_data" methods are listed in the following table.

**Table 5. The NDF Input Class *get_data* Methods**

| Method Name | Description | Output Data Type |
|---|---|---|
| gettimeseriesdata | Reads time series data chunk of a specified channel. | ndftimeseriesdata |
| getsegmentdata | Reads segment data chunk of a specified channel | ndfsegmentdata |
| getneuraleventdata | Reads neural event data chunk of a specified channel | ndfneuraleventdata |
| geteventdata | Reads binary experimental data chunk of a specified channel | ndfeventdata |
| getgenericmatrixdata | Reads a specified NDF generic matrix data set. | ndfgenericmatrixdata |
| getdata | All-in-one method to read NDF data set/chunk of a specified type. | NDF data object of the specified type |

- The "get_parameter" methods are used to extract additional parameters/information from the data set. These methods can be classified into three groups. Each group applied to four internal NDF data types and an "all-in-one" method can also be used to perform the same functionality by specified the NDF data type string (see, NDF data type constant string identifiers in page-6). The available "get_parameter" methods of an NDF input class are listed in the following table,

**Table 6. The NDF Input Class *get-parameter* Methods**

| Method Name | Description | Output Data Type |
|---|---|---|
| geteventdataduration | Gets the time instance of the last data point of a event data channel | MatLab double |
| getneuraleventdataduration | Gets the time instance of the last data point of a neural event data channel | MatLab double |
| gettimeseriesdataduration | Gets the time instance of the last data point of a time series data channel | MatLab double |
| getsegmentdataduration | Gets the time offset of the last data segment of a segment data channel | MatLab double |
| getduration | All-in-one method to get time offset of the last data point of a specific type | MatLab double |
| geteventdataindexes | Returns the start and end indexes of a time interval of a specified event data channel | MatLab doubles |
| gettimeseriesindexes | Returns the start and end indexes of a time interval of a specified time series data channel | MatLab doubles |
| getsegmentindexes | Returns the start and end indexes of a time interval of a specified segment data channel | MatLab doubles |
| getneuraleventindexes | Returns the start and end indexes of a time interval of a specified neural event data channel | MatLab doubles |

| | | |
|---|---|---|
| t2indexes | All-in-one method to convert time interval to point indexes | MatLab doubles |
| geteventitemcnt | Returns the number of items in a binary experimental event data channel | MatLab int64 |
| gettimeseriesitemcnt | Returns the number of items in a time series data channel | MatLab int64 |
| getsegmentitemcnt | Returns the number of segment in a segment data channel | MatLab int64 |
| getneuraleventitemcnt | Returns the number of items in a neural event data channel | MatLab int64 |
| getitemcnt | All-in-one method to get item count of a specified channel and a given data type | MatLab int64 |

Command line syntax:

1) The “get_duration” methods

   ***tLen* = objName.geteventdataduration( *channel_num* );**

   ***tLen* = objName.getneuraleventdataduration( *channel_num* );**

   ***tLen* = objName.gettimeseriesdataduration( *channel_num* );**

   ***tLen* = objName.getsegmentdataduration( *channel_num* );**

The all-in-one method of the above commands are:

   ***tLen* = objectName.getduration( *ndftypeString, channel_num* );**

2) The “get_indexes” methods

   **[*start_idx, end_idx*] = objName.geteventdataindexes( *channel_num, start_offset, end_offset* );**

   **[*start_idx, end_idx*] = objName.gettimeseriesindexes( *channel_num, start_offset, end_offset* );**

   **[*start_idx, end_idx*] = objName.getneuraleventindexes( *channel_num, start_offset, end_offset* );**

   **[*start_idx, end_idx*] = objName.getsegmentindexes( *channel_num, start_offset, end_offset* );**

The all-in-one method for the above commands is:

   **[*start_idx, end_idx*] = objectName.t2indexes( *ndftypeString, channel_num, start_offset, end_offset* );**

On these "get_indexes" method, if argument ***end_offset*** is set to -1, the returned ***end_idx*** is the index of the last data point in the data channel. The indexes are searched within the semi-open time interval, **[*start_offset, end_offset*)**. If there is no data point within the input time interval, both the return values (***start_offset*** and ***end_idx***) are set to -1.

3) The “get_itemcnt” methods

   ***nItem* = objName.geteventitemcnt( *channel_num* );**

   ***nItem* = objName.gettimeseriesitemcnt( *channel_num* )**

   ***nItem* = objName.getsegmentitemcnt( *channel_num* );**

   ***nItem* = objName.getneuraleventitemcnt( *channel_num* );**

The all-in-one method for the above commands is:

   ***nItem* = objectName.getitemcnt( *ndftypeString, channel_num*);**

The return values of the above "get_itemcnt" methods are in MatLab int64 type. This is to make consistent with the NDF API library. However, since MatLab doesn't support arithmetic operations of data type int64, it should be casted to double type whenever arithmetic operations are required. These methods directly extract the data item count from the binary data file. The item count can also be obtained from the data-info structure that is extracted from the NDF header rather than from the binary data file. If the data files are well formed, the item count from both sources should be the same. However, there is chance that the item count of a channel is not defined in the NDF header, e.g. on an unfinished data steaming/recording procedure. In this case, the data item count can be obtain using the 'get_itemcnt' method rather than from the ***itemCount*** property of an NDF data-info object.

The following is an example of using the 'get_parameter' method:

> ***LastT*** **= objName.getsegmentdataduration( 1 );**
> **or. *LastT* = objName.getduration( 'segment', 1);**

The above function call returns the time instance of the last data segment and assigns it to variable ***LastT***. You may want to get segment data that offsets in a specified time interval, e.g. in the last half time. You can firstly convert the time interval [ ***LastT***/2, ***LastT***] to indexes, then call the "get_data" method:

> **[*start_idx, end_idx*] = objName.getsegmentindexes( 1, *LastT*/2, -1);**
> **or. [*start_idx, end_idx*] = objName..getsegmentindexes( 'segent', 1, *LastT*/2, -1);**

Since the time interval is defined as a semi-open interval, [***LastT*/2, *LastT***) will not return index of the last data point. In order to return the last data point, -1 is used to indicate that the last data point is required. Then call the "get_data" method to get the segment data:

> ***data*= objName.getsegmentdata( 1, *start_idx, end_idx*);**
> **or. *data* = objName..getdata( 'segment', 1, *start_idx, end_idx*);**

On success, this returns an **ndfsegmentdata** object ***data*** with metadata in property ***data.DataInfo*** and a cell array in property ***data.Data*** that contains the segment data and indexes.

- In addition to the above methods, a method **duphdr** can be used to duplicate the current NDF header to a file. This method can be used to print out the NDF header in a more readable manner. The syntax of method **duphdr** is:

  **objName.duphdr( *filename* );**
  **objName.duphdr( *filename*, force-overwritten);**

  Argument ***filename*** is the path of the output NDF header file. It must not be the same as the path of input NDF header. On success, the method returns 1. Otherwise, it returns 0. If the output filename already exist, user will be prompted to select continue or cancel the writing. On the second case, a user can define the value of argument **force-overwritten = 1** to allow the function to overwrite the existing file without prompt, when **force-overwritten = 0,** the function is identical as the first case.

## 2.2. The NDF Output Class

The NDF output class **ndfwrite** acts like a status machine to manage and control the NDF data output. Each NDF output object is associated with one NDF data set. After the NDF output object is created, metadata and other information can be added to the object. Data chunks for each channel can be written to the data file sequentially in multiple-runs output mode. Multiple channels of data can be written to file in parallel, i.e. data from one channel can be 'written' to file without needing to wait for the data output from other channels to finish. The NDF output object manages the data/data-info objects pair of each channel and assigns the status of the output procedure of each channel to the data objects. The data output procedure of a data channel finished whenever the "**writedata**" method returns 1. The object automatically checks the completeness of all output operations and data elements before the NDF header can be written to file. Only when all the registered data/data-info pairs are finished writing to file, will the NDF header write method "**writendfheader**" be performed. The NDF output object will be locked for further data output once the NDF header has been written to file and the NDF data set output procedure ends.

### 2.2.1. The NDF Output Class constructor

The NDF output class constructor supports two modes, **create** and **append** mode. The default mode is '**create**'. Under create mode, an NDF output object is created by providing a valid output file path as an argument to the **ndfwrite** class constructor. The constructor will check the existance of the output directory that is used for the NDF header file and binary data files output. If the directory doesn't exist, the constructor will try to create a new one. If the directory can’t be created, the object will not be created and an error message will be returned. If the directory exists but is not empty, a warning message is shown. It is not recommended to write NDF data to a not empty directory since the NDF function always appends a new variable to an existing MAT file. The NDF functions only check for duplicated variable names for variables belonging to the object. The duplicated variable names on the original file are not checked.

Command Syntax:

***w* = ndfwrite( ‘*output_filepath*’)**

This command creates an **ndfwrite** object '***w***' for NDF data output. To add general information to the new object automatically, an existing NDF file can be used as a template. The information on the template (such as Laboratory, Investigator) will automatically be used to fill the new object during the construction procedure. Command line

***w* = ndfwrite( ‘*output_filepath*’, ‘*template_filepath*’)**

creates an **ndfwrite** object and duplicates the general data info from the template file to the new object.

If a user wants to append new data channels to the existing NDF data set, the '**append**' mode can be used. This is achieved by providing key word '***append***' as the third argument to the NDF output object constructor:

**w = ndfwrite( ‘*output_filepath*’, ‘’, '*append*')**

In '***append***' mode, the second argument (the template file path) is not used whether or not it is specified. The '***output-filepath'*** must be a valid NDF data header file otherwise the object construction will fail. The '***append***' mode doesn't create a new directory for the data and there will be no warning for a non-empty directory. Under the '***append***' mode, all output data are merged into the input data file.

If a user wants to wrap one or more existing MAT files that are NDF binary data specified manually, the '**header only**' mode can be used. This is specified by providing key work '**hdronly**' as the third argument to the NDF output object constructor:

***w* = ndfwrite( '*output_filepath*', '', '*hdronly*')**

In '***header only***' mode, if the output file has already existed, the object is automatically combined with 'append' mode. In this case, the second argument (the template file path) is not used whether or not it is specified. There will be no '**writedata**' method available from the **ndfwrite** object. The NDF data info object is created then added to **ndfwrite** object directly without need to associate it to an NDF data object.

### 2.2.2. The NDF Output Class Properties

Properties of an **ndfwrite** object are the same as an **ndfread** object except that the ***history*** property of an **ndfwrite** object is writable. This allows the new processing parameters to be directly appended to the new data set. The other properties are updated indirectly and driven by the relevant methods. For example, if a new data info object is successfully added to the object, the relevant channel count will be automatically increased by 1. As for the **ndfread** class, in addition to directly accessing property "**ChannelCount"**, the "**channelcnt"** method can also be used to obtain the number of channels of a specified NDF data type.

### 2.2.3. The NDF Output Class methods

The NDF output class **ndfwrite** provides "get_datainfo" methods to extract information already added to the object. These "get_datainfo" methods are the same as those in the NDF input class **ndfread**. The NDF output class also provides methods for status management of data output and processing.

- The NDF output class "get_datainfo" methods extract the NDF data info of a specified data type from the object. Except for the **getgeneraldatainfo** method, all the other "get_datainfo" methods are called in the same way as,

  **objectName.methodNam(*channel_num*).**

  For example:

  ***info* = *w*.gettimeseriesdatainfo(1);**

  will return an **ndftimeseriesdatainfo** object that contains the metadata of the first time series data channel. The number of available time series data channels is defined by the class property ***w*.ChannelCount.timeSeriesDataCnt.** The NDF "get_datainfo" methods include:

**Table 7. The NDF Output Class *get_datainfo* Methods**

| Method Name | Description | Output Data Type |
|---|---|---|
| getgeneraldatainfo | Gets general info of the NDF data set | ndfgeneraldatainfo |

| | | |
|---|---|---|
| gettimeseriesdatainfo | Gets data info of a specified time series data channel. | ndftimeseriesdatainfo |
| getsegmentdatainfo | Gets data info of a specified segment data channel. | ndfsegmentdatainfo |
| getneuraleventdatainfo | Gets data info of a specified neural event data channel. | ndfneuraleventdatainfo |
| geteventdatainfo | Gets data info of a specified experimental event data channel | ndfeventdatainfo |
| getgenericmatrixdatainfo | Gets data info of a specified generic matrix data channel. | ndfgenericmatrixdatainfo |
| getimagedatainfo | Gets data info of a specified image data channel. | ndfimagedatainfo |
| getuserdefineddatainfo | Gets data info of a specified user defined data channel. | ndfuserdefineddatainfo |
| getinfo | All-in-one method to get data info from a specified data type | NDF data info object of the specified type |

- In addition to the "get_datainfo" methods, the NDF output class also provides the following methods for NDF data output management:

**Table 8. The NDF Output Class Data Output Methods**

| Method Name | Description | Output Data Type |
|---|---|---|
| writedata | Outputs a data object that may be just a data chunk of the data channel to NDF data files. | 0 data chunk written ok.<br>1 full channel written ok.<br>An error message if written failed. |
| updatedatainfo | Couples a data object with a data info object and update the data info object with metadata set up during the data output operation. | The handle of the data info object with updated contents. |
| adddatainfo | Adds a specified data info object to the **ndfwrite** object | The number of channel in the list. |
| writendfheader | Writes the NDF header to file. This will finish the data output operation and lock the **ndfwrite** object. | No return value. |
| unlockoverwritten | Unlocks the **ndfwrite** object in order to overwrite the header file. | No return value |
| modifydatainfo | Modifys the contents of a specified data info object that has been added to the output object | Number of items overwritten |

The following steps show the command line syntax for a typical sequence of operations that creates an NDF data file with one data channel.

I. Writing data (or data chunk) to NDF data set.

***w*.writedata( *data* );**

This will write an NDF data object (***data***) that contains binary data or data chunk to an NDF data set. Before calling this method, no metadata is required to set to the object with the exception of member "***compress***" that must be set to 1 if a compress data stream is expected to be used. By analysing the data member ***data.Data,*** this method will automatically update the relevant metadata such as item count, data type etc. of the data object on return. Calling the "**updatedatainfo**" method after the whole data channel is finished writing to file will pass these metadata to an NDF data info object. (Details of the NDF data info object and NDF data object can be found in Chapter-3 and Chapter-4).

II. Updating and coupling a data info object with a data object.

**_w_.updatedatainfo( _data, data_info_);**
**_w_.updatedatainfo( data, _data_info, setsplitinfo_flag_ );**

The first case couples the ***data_info*** and ***data*** object pair and updates the ***data_info*** object with metadata from the data object. The second case provides further options for the ***data_info*** updating. By default, the ***setsplitinfo_flag*** is set to 1, indicating that the data file split information is also copied from the data object to the ***data_info*** object. If ***setsplitinfo_flag*** = 0, the split data parameters will not be copied. This is useful when the data object is duplicated from an ***ndfread*** object and the file split information is not going to be set to the ***data_info*** object.

III. Adding data info to the object.

**_w_.adddatainfo( _datainfo_ );**

adds a data info object to object ***w.*** To call this method, the NDF data info object must be firstly registered with an NDF data object by calling method "***updatedatainfo***".

IV. Writing the NDF header to file,

**_w_.writendfheader();**
**_w_.writendfheader( '_filepath_');**

The first instance writes the NDF header to default NDF header file that is set when the **ndfwrite** object is created. If a different header file path is defined as in the second instance, the NDF header will be written to the specified path.

V. Unlock the **ndfwrite** object,

**_w_.unlockoverwriten();**

This will unlock the object to allow overwriting of the existing header file.

VI. Modify the contents of a data info object that has been added to the **ndfwrite** object. Normally, a data info object should be fully defined before it can be added to an **ndfwrite** object. It cannot be changed after that because some of the elements referenced by the full data set must be set programmatically based on the other elements and data file structures. However, there is a chance (in particular when the toolbox is used as user desktop tool) that mistakes might be found afterward. In this case, this function provides means to modify some of the contents of a data info structure that has been added to the object.

Syntax:

**_w_.modifydatainfo( _index, datainfo_ );**

Argument ***index*** is the index of the specified data info in the object. The type of data info to modify is implicitly defined by the second argument, i.e. it is the same type as the object of the second argument. Argument ***datainfo*** is the new data info object that is the source to update the original one. Only the local referenced parameters in object ***datainfo*** are applied. It is not necessary but recommended to use the "get_datainfo" method to extract the data info object from the target as a template then change some of the field values that do not fit. This function is then called to complete the modification. For example, if element 'memberID' of the second time series data info needs to change, e.g. to value 100. We can do this in the following way:

```
%Extract the second time series data info object as template
tminfo = w.gettimeseriesdatainfo( 2);

%Change element 'memberID' to 100. Other fields can also
%be changed in this phase if required.
tminfo.DataInfo.memberID = 100;
%Replace the contents of the old data info object with the new one.
w.modifydatainfo( 2, tminfo);
```

This will change the contents of the second time series data info object except the global referenced elements from object ***w*** using the contents of object ***tminfo***.

## 2.3. The NDF Input Class with Arguments Parser

The CARMEN system uses XML services input argument files to pass parameters from the portal to the data processing services. The NDF input class with argument parser (the NDF class ***ndfsvcread***) is implemented to interpret arguments within the XML input argument files before the NDF data I/O object can be created. The class inherits from the NDF input class with additional function parsing the NDF input parameters from the input files during the construction procedure. All methods defined within the NDF input class are also available in this class.

### 2.3.1. The CARMEN Service Input Argument File

A CARMEN services input argument file is an XML file used to pass parameters and settings from the CARMEN portal to a service. It looks like:

```
 <ndfserviceinput>
     <filename>theNdfHeader.ndf</filename>
    <datatype>neuralevent</datatype>
    <member>
       <memberindex>0</memberindex>
       <channelindex>< /channelindex>
       <startindex>< /startindex>
       <endindex> </endindex>
       <timefrom > </timefrom>
       <timeto> </timeto>
    </member>
</ndfserviceinput>
```

Each tag within the file defines a set of input parameters:

- Element <filename> defines the file path (absolute or relative) of the NDF header file. This element must appear once only.
- Element <datatype> defines the NDF data type to process. This element must appear once only. The data types for NDF data format can be ***timeseries, event, neuralevent,***

***segment, matrix, userdefined***, and ***image*** (see, NDF data type constant string identifiers table in page-6).

- Optional element <member> defines additional parameters for data processing.
    - All <member> children elements appear zero or one time.
    - Multiple <member> elements can be defined.
    - <memberindex> defines the NDF data group ID. If it is defined, only the channels with the same member ID are processed. If this tag is not defined, channels with any member ID are used.
    - <channelindex> is the channel number indexes in CSV format representing the channel numbers to be processed. If it is not defined, all channels in the NDF data set are used. If <memberindex> is defined, only channels with the same member ID are used.
    - <startindex> and <endindex> define the data item interval within the channels to be processed
    - <timefrom> and <timeto> are double type floating point data values which define the time interval in seconds of each data channel to be processed. If <startindex> is used, these elements are ignored.
    - <channelindex>, <startindex> and <endindex> are all zero-based indexes.
    - Empty index string (or with value -1) implies the default values are used.

The CARMEN services input argument files are normally generated by the services user's interface of the CARMEN portal.

### 2.3.2. Constructor

On construction, the NDF input class with arguments parser firstly reads and interprets the input argument XML file. Then the NDF input class constructor is called to finally construct the object. Only if there is a valid service input arguments file and its contents contain a valid NDF header file, can the object be constructed.

Command syntax

***A* = ndfsvcread( ‘*input_argfilepath*’)**

will create an **ndfsvcread** object '***A***' for CARMEN service NDF data output. The constructor returns an empty object if the construction failed.

### 2.3.3. Properties

The class inherits all the private and public properties from its superclass, the NDF input class **ndfread**. Two new read-only properties are defined to store parameters read from the input argument file.

**Table 9. NDF Class ndfsvcread Specified Properties**

| Property | Descriptions | Data Types |
|---|---|---|
| inputDataType | the NDF data type string | MatLab char |
| MemberList | Matrix of structure <member> | MatLab Matrix |

Property **MemberList** is an n-by-1 MatLab matrix with n **member** elements. The **member** element is a MatLab structure with the following fields:

**Table 10. Field Names of Property MemberList**

| Field Name | Descriptions | Data Type |
|---|---|---|
| flag | Bitwise flag indicates whether specified field is set with data received from the input argument file. | MatLab uint32 |
| memberIndex | The NDF data group ID | MatLab unit32 |
| channelList | A CSV format string lists the channel numbers of specified channels. | MatLab char |
| startIndex | The start index of the data chunk to process | MatLab double |
| endIndex | The end index of the data chunk to process | MatLab double |
| timeFrom | The start time instance of the data chunk to process | MatLab double |
| timeTo | The end time instance of the data chunk to process | MatLab double |

The class provides a set of bit-wise constants to identify if a specified field in a member structure exists. These include **MEMERID, CHANNELLST, STARTINDEX, ENDINDEX, TIMEFROM** and **TIMETO.** For example, the follow code segment tests if field **memberindex** in the first member element of the list is set from the input argument file:

```
if  bitand( A.MemberList(1).flag, A.MEMBERID)
     ....
end
```

Other properties inherited from the superclass **ndfread** can be found in Chapter-2.1.1

## 2.3.4. Methods

In addition to the methods defined in the superclass, additional methods are defined in this class to extract parameters and data within the **ndfsvcread** object. The following table lists methods that are defined within the **ndfsvcread** but not in superclass **ndfread**.

**Table 11. NDF Class ndfsvcread Specified Methods**

| **Method Name** | **Descriptions** | **Output Data Type** |
|---|---|---|
| getmemberlistcnt | Returns the number of items within the **MemberList** property matrix. | MatLab double |
| getchannelidx | Extracts channel indexes of a specified member from the CSV string to a n-by-1 numeric matrix. | n-by-1 MatLab uint32 matrix |
| getallchannelidx | Returns channel indexes of all input members defined within the **MemberList** properties. | n-by-1 MatLab uint32 matrix |
| getmemberchannelidx | Returns a list of channel indexes belong to a given member ID within an NDF data set. | n-by-1 MatLab uint32 matrix |
| getparamfromID | Returns a member parameter set defined by specified member ID | MatLab structure |
| getchannelcnt | Returns the channel count of a specified member or all members in the member list. | MatLab double |
| getspecdata | Reads data chunk of a given channel of a specified data type | NDF data object |
| getspecdatainfo | Gets data info of a given channel of specified data type | NDF data info object |
| getspecdatachannelcnt | Returns all available data channel count of the specified data type in the NDF data set regardless of the member ID. | MatLab uint32 |
| getspecdatatype | Returns the data type string defined by property **inputDatatype** | MatLab char |
| getspecdataitemcnt | Returns the data item count of a given channel of the specified data type. | MatLab int64 |

| | | |
|---|---|---|
| getspecindexes | Converts the time interval to index range. | MatLab double |
| getspecdataduration | Returns the time instance of the last data point in seconds | MatLab double |

Command syntax:

I. Get member item count of property ***MemberList***

   ***cnt* = objName.getmemberlistcnt();**

II. Get the input channel indexes list by interpreting a given element in property ***MemberList***. If a list of channels is defined in the input argument file by tag <channelindex>, channel indexes of the specified elements are used. Otherwise, all channel indexes in the input NDF data are used. The channel indexes list will be further filtered by the member IDs defined by tag <memberindex>. If <memberindex> tag is not defined or empty, the channel indexes list will not be filtered:

   ***idxLst* = objName.getchannelidx( index );**

III. Get all channel indexes of members defined within property ***MemberList*** regardless of the **memberID**

   ***idxLst* = objName.getallchannelidx();**

IV. Get channel indexes of a given member ID in the NDF data set. This method returns all indexes of channels with the same ***memberID*** in the data set regardless of the ***Memberlist*** that is defined by the input argument file.

   ***idxLst* = objName.getmemberchannelidx ( *memberID*);**

V. Get the member parameter set by a given ***memberID***. This method returns a **member** structure containing parameters of all specified channels with the same ***memberID*** in the ***MemberList*** matrix:

   ***paramStruct* = objName.getparamfromID( *memberID* );**

VI. Get channel count of a specified element or all member in the member list

   ***cnt* = objName.getchannelcnt( *index* );**
   ***cnt* = objName.getchannelcnt();**

   The first instance gets the channel count of the ***index_th*** member of the **MemberList**. The second instance gets the channel count of all members defined within the **MemberList.** This function refers to channels that are specified by the input argument file (whilst method "**channelcnt**" refers to channels of a specified data type in the NDF data set).

VII. Read a chunk of data of type defined within the object by given channel number and index range.

***data* = objName.getspecdata( *channel_num* );**
***data* = objName.getspecdata( *channel_num, startidx* );**
***data* = objName.getspecdata( *channel_num, startidx, endidx*);**

The first instance reads all data points of a given channel. The second instance reads data chunk of a given channel from the start index to the end of data channel. The third instance reads data chunk defined by [***startidx, endidx***] of a given channel.

VIII. Read data info of type defined within the object by given channel number.

***datainfo* = objName.getspecdatainfo( *channel_num*);**

IX. Get all available data channel counts of the specified data type in an NDF data set regardless of the member IDs specified by the input argument file.

***cnt* = objName.getspecdatachannelcnt( *channel_num*);**
***cnt* = objName.getspecdatachannelcnt( *channel_num, typestr*);**

The first instance returns the channel count of the default data type**.** The second instance returns the channel count of data type defined by input argument, ***typestr***. This function refers to data channels in the NDF data set.

X. Get the data type string of the current object. This method returns the default data type string specified by the input argument file.

***datatypeStr* = objName.getspecdatatype( );**

XI. Get data item count of a given channel of the specified data type.

***cnt* = objName.getspecdataitemcnt( *channel_num*);**
***cnt* = objName.getspecdataitemcnt( *channel_num, typestr*);**

The first instance returns the data item count of a given channel of the default data type. The second instance returns the data item count of a given channel of data type defined by input argument "***typestr***".

XII. Convert the time interval to index range for a given channel.

**[*startidx, endidx*] = objName.getspecindex( *timefrom, timeto, channel_num*);**
**[*startidx, endidx*] = objName.getspecindex( *timefrom, timeto,***
***channel_num, typestr*);**

The first instance converts the input time interval to the index range of a given channel of the default data type. The second instance converts the input time interval to the index range of a given channel of data type defined by input argument "***typestr***".

XIII. Obtain the time instance of the last data item of a given channel in seconds.

***timeoffset* = objName.getspecdataduration( *channel_num*);**
***timeoffset* = objName.getspecdataduration( *channel_num, typestr*);**

The first instance returns the time instance of the last data point (or segment) of a given channel of the default data type. The second instance returns the time instance of the last data point (or segment) of a given channel of data type specified by input argument "***typestr***".

Other methods defined in the superclass **ndfread** can be found in Chapter-2.1.3

# 3. The NDF Data Info Classes

The NDF toolbox defines eight NDF data info classes corresponding to the eight NDF data info structures defined within the NDF API. The NDF data info classes are used as a carrier for data channel metadata input and output. All NDF data info classes are MatLab handle classes. Therefore, they cannot be duplicated by just assigning an object to another variable. A "clone" method must be called in order to create a duplication of an NDF data info object. An NDF data info object can be created by calling the class constructor explicitly or calling the "get_datainfo" methods of an **ndfread** or **ndfwrite** object. The following table lists the NDF data info classes defined within the toolbox.

**Table 12. The NDF DataInfo Classes**

| **NDF Data Info Class Name** | **Description** |
|---|---|
| ndfgeneraldatainfo | Data class for general data info of the data set |
| ndftimeseriesdatainfo | Data class for metadata of a time series data channel |
| ndfsegmentdatainfo | Data class for metadata of a segment data channel |
| ndfneuraleventdatainfo | Data class for metadata of a neural event data channel |
| ndfeventdatainfo | Data class for metadata of a experimental event data channel |
| ndfimagedatainfo | Data class for metadata of a image data channel |
| ndfgenericmatrixdatainfo | Data class for metadata of a generic matrix data channel |
| ndfuserdefineddatainfo | Data class for metadata of a user defined data channel. |

## 3.1. The Data Info Object Constructor

An NDF data info object can be constructed by calling the class constructor. A 'get-datainfo' method of an NDF data I/O class also creates an NDF data info object.

Command line syntax:

***obj* = ndfDataClassName();**

Example:

***tmDataInfoObj* = ndftimeseriesdatainfo();**

This creates an NDF time series data info object ***tmDataInfoObj.***

## 3.2. The Data Info Object Properties

There are only two public accessible properties in an NDF data class as shown in the following table.

**Table 13. Public accessible Properties of the NDF Data Info Classes**

| **Property Name** | **Description** |
|---|---|
| DataInfo | MatLab structure storing the metadata |
| NdfFilepath | Full path of the NDF header file |

### 3.2.1. The *DataInfo* Properties

The ***DataInfo*** property of the NDF data info object is a MatLab structure and each field is directly accessible rather than using the 'get' and 'set' methods. For example, if **G** is an NDF general data info object, **G.laboratory** will display the laboratory name and **G.laboratory='Carmen Lab'** will assign a laboratory name to the field.

I. The ***DataInfo*** property of the General Data Info class

| Field Name | Descriptions | Data Type |
|---|---|---|
| description | General description of the data set | MatLab char |
| laboratory | The Laboratory name | MatLab char |
| investigator | Name of the investigator | MatLab char |
| specimenID | The Specimen identifier | MatLab char |
| createDate | Date when the data set was created | MatLab char |
| createTime | time when the data set was created | MatLab char |
| recordID | Identifier of the data set | MatLab char |

II. The ***DataInfo*** property of the Time Series Data Info class

| Field Name | Description | Data Type |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Name of the data file that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| unitOfMeasurement | Unit of measurement of the data channel. | MatLab char |
| acquisitionEquipment | Name of the data acquisition equipment | MatLab char |
| equipmentSettings | Text describes the settings of the equipment. | MatLab char |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| endDate<br>endTime<br>endDecimalSecond | The end date string, in ccyy-mm-dd format<br>The end time string, in HH:MM:SS format<br>The fragment of the end time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| samplingRate | Data sampling rate in Hz. | MatLab double |
| transducerType | Transducer type sting | MatLab char |
| adcPrecision<br>adcZeroOffset<br>adcResolution<br>adcValueUnit | Precision of the ADC system, say 12-bit<br>Zero offset of the digitized data<br>The quantization step of the ADC system<br>Unit of measurement | MatLab int32<br>MatLab double<br>MatLab double<br>MatLab char |
| lpfCutoffFreq<br>lpfType<br>lpfOrder | The low-pass filter cut-off frequency<br>Type of the low-pass filter<br>Order of the low-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| hpfCutoffFreq<br>hpfType<br>hpfOrder | The high-pass filter cut-off frequency<br>Type of the high-pass filter<br>Order of the high-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| timeOffset | Time offset in seconds from the start time to the first data point | MatLab double |

| | | |
|---|---|---|
| itemCount | Number of points in the data channel | MatLab int64 |
| varName | Name of the variable | MatLab char |
| probePosition | Coordinates of the probe. | MatLab char |
| matLabel | Label of the variable in a MAT file. | MatLab char |
| extraParams | Additional parameters defined by the data | MatLab structure |
| extraInfo | Additional information defined by the data | MatLab char |
| recommendApp | The application name that recommended to 'open' the data | MatLab char |

Conventions are applied to some of the optional fields.

1. ADC settings are valid (enabled) if both **adcPrecision** and **adcResoluton** are non-zero.
2. Low-pass filter settings are applied if both **lpfOrder** and **lpfCutoffFreq** are non-zero
3. High-pass filter settings are applied if both **hpfOrder** and **hpfCutoffFreq** are non-zero
4. **extraParams** is defined as a structure with field names, "name", "value" and "unit".
5. **matLabel** is a name that corresponds to the **varName** but meet the specification as a valid MatLab variable name whilst **varName** can be any reasonable string. You are not necessary to provide a **matLabel** name. The NDF output object can do this for you. This convention applies to all other NDF data info objects that are with the **varName** and **matLabel** pair.

III. The ***DataInfo*** property of the Segment Data Info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| fixedLength | 1 for fixed length segment, 0 for variable length segment | MatLab uint32 |
| dataFilename | Name of the data file that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| unitOfMeasurement | Unit of measurement of the data channel. | MatLab char |
| acquisitionEquipment | Name of the data acquisition equipment | MatLab char |
| equipmentSettings | Text describes the settings of the equipment. | MatLab char |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| endDate<br>endTime<br>endDecimalSecond | The end date string, in ccyy-mm-dd format<br>The end time string, in HH:MM:SS format<br>The fragment of the end time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| sampleingRate | Data sampling rate in Hz. | MatLab double |
| transducerType | Transducer type sting | MatLab char |
| adcPrecision<br>adcZeroOffset<br>adcResolution<br>adcValueUnit | Precision of the ADC system, say 12-bit<br>Zero offset of the digitized data<br>The quantization step of the ADC system<br>Unit of measurement | MatLab int32<br>MatLab double<br>MatLab double<br>MatLab char |
| lpfCutoffFreq<br>lpfType<br>lpfOrder | The low-pass filter cut-off frequency<br>Type of the low-pass filter<br>Order of the low-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| hpfCutoffFreq<br>hpfType<br>hpfOrder | The high-pass filter cut-off frequency<br>Type of the high-pass filter<br>Order of the high-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |

| | | |
|---|---|---|
| triggerType<br>triggerThreshold<br>segmentLeftTimeSpan<br>segmentRightTimeSpan | Type of trigger threshold*<br>The trigger threshold. The type is defined by triggerType.<br>Left time span of segment from the trigger point<br>Right time span of the segment from the trigger point | MatLab int32<br>MatLab double<br>MatLab double<br>MatLab double |
| timeOffset | Time offset in seconds from the start time to the first data point | MatLab double |
| itemCount | Number of segments in the data channel | MatLab int64 |
| varName | Name of the variable | MatLab char |
| probePosition | Coordinates of the probe position. | MatLab char |
| matLabel | Label of the variable in a MAT file. | MatLab char |
| extraParams | Additional parameter defined by the data | MatLab structure |
| extraInfo | Additional information defined by the data | MatLab char |
| recommendApp | The application name that recommended to 'open' data | MatLab char |

***triggerType** defines the type that parameter **triggerThreshold** represents**.** **triggerType**=0, triggerThreshold is not used. **triggerType**=1, triggerThreshold is the signal level. **triggerType**=2, triggerThreshold is the signal slope. **triggerType**=3, triggerThreshod is the shape similarity. **triggerType**>3, triggerThreshold may be application defined and it should be described in the **extrraInfo** entry.

Conventions are applied to some of the optional fields.

1. ADC settings are valid (enabled) if both **adcPrecision** and **adcResoluton** are non-zero.
2. Low-pass filter settings are applied if both **lpfOrder** and **lpfCutoffFreq** are non-zero
3. High-pass filter settings are applied if both **hpfOrder** and **hpfCutoffFreq** are non-zero
4. ***extraParams*** is defined as a structure with field names, "name", "value" and "unit".
5. ***extraInfo*** is defined as a structure with field names, "infotext", "elementID", "sortedID" and "sourceID".

More details about the data info fields can be found in NDF specification document [1],

IV. The ***DataInfo*** property of the Neural event data info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Name of the data file that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| acquisitionEquipment | Name of the data acquisition equipment | MatLab char |
| equipmentSettings | Text describes the settings of the equipment. | MatLab char |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| endDate<br>endTime<br>endDecimalSecond | The end date string, in ccyy-mm-dd format<br>The end time string, in HH:MM:SS format<br>The fragment of the end time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| sampleingRate | Data sampling rate of the raw time series data in Hz | MatLab double |
| adcPrecision<br>adcZeroOffset<br>adcResolution<br>adcValueUnit | Precision of the ADC system, say 12-bit<br>Zero offset of the digitized data<br>The quantization step of the ADC system<br>Unit of measurement | MatLab int32<br>MatLab double<br>MatLab double<br>MatLab char |

| | | |
|---|---|---|
| transducerType | Transducer type sting | MatLab char |
| lpfCutoffFreq<br>lpfType<br>lpfOrder | The low-pass filter cut-off frequency<br>Type of the low-pass filter<br>Order of the low-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| hpfCutoffFreq<br>hpfType<br>hpfOrder | The high-pass filter cut-off frequency<br>Type of the high-pass filter<br>Order of the high-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| timeResolution | Time resolution in seconds of the event data | MatLab double |
| timeOffset | Time offset in seconds from the start time to the first data point | MatLab double |
| itemCount | Number of points in the data channel | MatLab int64 |
| varName | Name of the variable | MatLab char |
| probePosition | Coordinates of the probe. | MatLab char |
| matLabel | Label of the variable in a MAT file. | MatLab char |
| extraParams | Additional parameter defined by the data | MatLab structure |
| extraInfo | Additional information defined by the data | MatLab structure |
| recommendApp | The application name that recommended to 'open' data | MatLab char |

Conventions are applied to some of the optional fields.

1. ADC settings are valid (enabled) if both **adcPrecision** and **adcResoluton** are non-zero.
2. Low-pass filter settings are applied if both **lpfOrder** and **lpfCutoffFreq** are non-zero
3. High-pass filter settings are applied if both **hpfOrder** and **hpfCutoffFreq** are non-zero
4. ***extraParams*** is defined as a structure with field names, "name", "value" and "unit".
5. ***extraInfo*** is defined as a structure with field names, "infotext", "elementID", "sortedID" and "sourceID".
6. ***timeResolution*** is used to recover the real time instance value from the event data, it may or may not be equal to the reciprocal of value ***samplingRate.***

More details about the data info fields can be found in NDF specification document [1].

V. The ***DataInfo*** property of the Experimental event data info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Name of the data file that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| recordType | 0 indicates an annotation data set, 1 a binary event data | MatLab int32 |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| timeResolution | Time resolution in seconds of the event time offset data | MatLab double |
| itemCount | Number of points in the data channel | MatLab int64 |
| varName | Name of the variable | MatLab char |
| matLabel | Label of the variable in a MAT file. | MatLab char |
| extraInfo | Additional information defined by the data | MatLab char |

Field ***recordType*** defines the type of event data in an NDF data set. ***recordType***=0 indicates the event data are stored in an XML annotation file. Otherwise, they are stored in a MAT file as binary data pair, (offset, value). Field ***extraInfo*** is a plain text string.

VI. The ***DataInfo*** property of the Image data info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Name of the data file that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| acquisitionEquipment | Name of the data acquisition equipment | MatLab char |
| equipmentSettings | Text describes the settings of the equipment. | MatLab char |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| endDate<br>endTime<br>endDecimalSecond | The end date string, in ccyy-mm-dd format<br>The end time string, in HH:MM:SS format<br>The fragment of the end time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| imageWidth | Width of the image in pixels | MatLab int32 |
| imageHeight | Height of the image in pixels | MatLab int32 |
| frameCount | Number of frames in the data | MatLab int32 |
| frameRate | Frames per second for image sequence | MatLab double |
| extraParams | Additional parameter defined by the data | MatLab structure |
| recommendApp | The application name that recommended to 'open' data | MatLab char |

***extraParams*** are defined in a structure with field names, "name", "value" and "unit".

VII. The ***DataInfo*** property of the Generic matrix data info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Data file name that contains the binary data. | MatLab char |
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| unitOfMeasurement | Unit of measurement of the data channel. | MatLab char |
| varName | Name of the variable | MatLab char |
| matLabel | Label of the variable in a MAT file. | MatLab char |
| appDefinedData | Additional description of the data defined by application | MatLab char |
| applicationID | Identifier of application that defines the data set | MatLab char |

VIII. The ***DataInfo*** property of the User defined data info class

| **Field Name** | **Description** | **Data Type** |
|---|---|---|
| memberID | Identifier of a group of channels | MatLab int32 |
| dataFilename | Name of the data file hat contains the binary data. | MatLab char |

| | | |
|---|---|---|
| dataLocation | URI if the data set is in different location from the header file | MatLab char |
| applicationID | Identifier of application that defines the data set | MatLab char |
| unitOfMeasurement | Unit of measurement of the data channel. | MatLab char |
| acquisitionEquipment | Name of the data acquisition equipment | MatLab char |
| equipmentSettings | Text describes the settings of the equipment. | MatLab char |
| startDate<br>startTime<br>startDecimalSecond | The start date string, in ccyy-mm-dd format<br>The start time string, in HH:MM:SS format<br>The fragment of the start time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| endDate<br>endTime<br>endDecimalSecond | The end date string, in ccyy-mm-dd format<br>The end time string, in HH:MM:SS format<br>The fragment of the end time in seconds | MatLab char<br>MatLab char<br>MatLab double |
| sampleingRate | Data sampling rate in Hz. | MatLab double |
| transducerType | Transducer type sting | MatLab char |
| adcPrecision<br>adcZeroOffset<br>adcResolution<br>adcValueUnit | Precision of the ADC system, say 12-bit<br>Zero offset of the digitized data<br>The quantization step of the ADC system<br>Unit of measurement | MatLab int32<br>MatLab double<br>MatLab double<br>MatLab char |
| lpfCutoffFreq<br>lpfType<br>lpfOrder | The low-pass filter cut-off frequency<br>Type of the low-pass filter<br>Order of the low-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| hpfCutoffFreq<br>hpfType<br>hpfOrder | The high-pass filter cut-off frequency<br>Type of the high-pass filter<br>Order of the high-pass filter | MatLab double<br>MatLab char<br>MatLab int32 |
| varName | Name of the variable | MatLab char |
| probePosition | Coordinates of the probe. | MatLab char |
| extraParams | Additional parameter defined by the data | MatLab structure |
| extraInfo | Additional information defined by the data | MatLab structure |
| userInfo | Additional metadata provides by the application | MatLab char |
| recommendApp | The application name that recommended to 'open' data | MatLab char |

Conventions are applied to some of the optional fields.

1. ADC settings are valid if both **adcPrecision** and **adcResoluton** are non-zero.
2. Low-pass filter settings are applied if both **lpfOrder** and **lpfCutoffFreq** are non-zero
3. High-pass filter settings are applied if both **hpfOrder** and **hpfCutoffFreq** are non-zero
4. ***extraParams*** is defined in a structure with field names, "name", "value" and "unit".
5. ***extraInfo*** is defined in a structure with field names, "infotext", "elementID", "sortedID" and "sourceID".
6. ***userInfo*** is a string of MatLab char type. It can be a plain text string or an XML node tree with tags defined by the application.

More details about the data info fields can be found in NDF specification document [1].

### 3.2.2. The *NdfFilepath* Property

Property ***NdfFilepath*** is a string of MatLab char type that defines the full path of the NDF header file. This property can be used as an identifier to match a data info objet with an **ndfread** or **ndfwrite** object.

## 3.3. The Data Info Object Methods

1. The following methods apply to all NDF data info classes,

| **Method** | **Description** | **Output Data type** |
|---|---|---|
| disp | Displays the key parameters that are normally need to set by default | |
| dispall | Displays all parameters | |
| clone | Creates a new object the is the clone the current object | NDF data info object |
| match | Check if the object signature is the same as the input | 1 match, 0 not match |
| uiedit | A GUI for editing the DataInfo structure | |

Syntax,

I. Display the metadata

**objName.disp();**

will display the contents of the main parameter of the object .

II. Display all parameters

**objName.dispall();**

will display the contents in the ***DataInfo*** property .

III. Clone a data info object

***NewObj*** **= objName.clone();**

This create a new object with the same contents of the original object.

IV. Check the signature of a **DataInfo** object

**objName.match(** ***signature*** **)**

where ***signature*** is a string represented the signature of an NDF data info object. The method returns 1 if the signature matches the object. Otherwise, returns 0.

V. Edit the contents of the **DataInfo** property from a GUI

**objName.uiedit();**

This will invoke a GUI with the values column editable for a user to change the value of each field. The GUI doesn't include fields "***extraInfo***" and "***extraParam***". These fields can only be changed using appropriate methods provided by the class. When finished editing, click the "OK" button to accept the results or "Cancel" button to discard the editing. Some fields are associated with the NDF data files, if these fields are set by the "**updatedatainfo**" method, you must leave these fields intact. A good way to avoid desynchronize a field with the NDF data files is to edit the **DataInfo** structure before calling the “**updatedatainfo**” method to couple the NDF data info object with the binary data set.

2. The following methods applied to classes **ndftimeseriesdatainfo**, **ndfsegmentdatainfo**, **ndfneuraleventdatainfo** and **ndfuserdefineddatainfo:**

| **Method Name** | **Description** | **Output Data Type** |
|---|---|---|
| hasadc | Checks if the ADC setting is available | MatLab double |
| haslowpass | Checks if the low-pass filter setting is available | MatLab double |
| hashighpass | Checks if the high-pass filter setting is available | MatLab double |
| addextrainfo | Adds a set of additional metadata | |
| addextraparams | Adds a set of additional parameters | |

Syntax:

I. Check if the ADC setting is available

   **objName.hasadc();**

   The method returns 1 if the ADC setting is valid, otherwise, returns zero.

II. Check if the low-pass filter setting is available.

   **objName.haslowpass();**

   The method returns 1 if the low-pass filter setting is valid, otherwise, returns zero

III. Check if the high-pass filter setting is available.

   **objName.highpass();**

   The method returns 1 if the high-pass filter setting is valid, otherwise, returns zero

IV. Add a set of additional metadata.
   The syntax of this method varies from class to class.
   i. NDF class **ndftimeseriesdatainfo**

      **objName.addextrainfo(** ***textinfo*** **);**

      Input argument ***textinfo*** is a text string for additional information about the data.

   ii. NDF class **ndfsegmentdatainfo**

      **objName.addextrainfo(** ***textinfo*****);**
      **objName.addextrainfo(** ***textinfo, sortedID*****);**
      **objName.addextrainfo(** ***textinfo, sortedID, sourceID*****);**

      Input arguments:
      ***textinfo*** **--** text string for additional information about the data
      ***sortedID*** **--** the spike sorting index if applicable.
      ***sorceID*** **--** the data source index if applicable.

      This method adds a set of extra info data to the object.

iii. NDF class **ndfneuraleventdatainfo**
The syntax of this class is the same as that for class **ndfsegmentdatainfo.**

iv. NDF class **ndfuserdefineddatainfo**

**objName.addextrainfo(** ***textinfo*****);**
**objName.addextrainfo(** ***textinfo, sourceID*** **);**

Input arguments:
***textinfo*** **--** text string for additional information about the data
***sorceID*** **--** the data source index if applicable.

This method adds a set of extra info data to the object.

V. Add a set of extra parameters

**objName.addextraparams(** ***name, value*****);**
**objName.addextraparams(** ***name, value, unit*****);**

Input arguments:
***name*** -- text string represents the parameter name
***value*** -- the parameter numeric value
***unit*** -- unit of measurement in string type when applicable.

The method adds a set of additional parameters to the object.

3. The "addextra_" methods for other NDF data info classes.

I. Add extra data info to object of class **ndfeventdatainfo.** The method for binary experimental event data info contains only a text string.

Syntax:

**objName.addextrainfo(** ***textinfo*****);**

Input arguments:
***textinfo*** **--** text string provides additional information about the data.

This method will add additional info text to the **ndfeventdatainfo** object.

II. Add extra parameter to object of class **ndfimagedatainfo**

Syntax:

**objName.addextraparams(** ***name, value*****):**
**objName.addextraparams(** ***name, value, unit*****);**

Input arguments:
***name*** -- text string represents the parameter name
***value*** -- the parameter numeric value
***unit*** **--** unit of measurement in string type when applicable.

The method adds a set of additional parameters to the object.

# 4. The NDF Data Classes

The NDF toolbox provides five NDF data types (classes) for storage of different types of binary neurophysiology data and experimental event data. The NDF data types are used as the input and output elements of an NDF data I/O class. It can also be used as the input and/or output arguments of function calls of a CARMEN data processing service. The advantage of using NDF data objects is that the NDF classes are handle classes. A MatLab handle class does not create new instance when assigned to a new variable or is passed through multiple levels of function calls. This can help to avoid unnecessary memory usage. The binary data are stored in class properties, ***Data***, as a conventional MatLab matrix. The following table lists the available NDF data classes defined within the toolbox:

**Table 14. The NDF Data Classes**

| NDF Data Type Name | Descriptions |
|---|---|
| ndftimeseriesdata | Data type for time series data |
| ndfsegmentdata | Data type for segmented time series data |
| ndfneuraleventdata | Data type for neural event data |
| ndfeventdata | Data type for binary experimental event data |
| ndfgenericmatrix | Semi-defined data type for high dimensional data matrix |

## 4.1. The Data Object Constructor

NDF data object can be constructed by calling the class constructor. A 'get-data' method of an NDF data I/O class can also create an NDF data object.

Command line syntax

***obj* = ndfDataClassName();**

Example:

***tmObj* = ndftimeseriesdata();**

This creates an NDF time series data object.

## 4.2. The Data Object Properties

There are only two public accessible properties in NDF data class as shown in the following table.

**Table 15. Public Accessible Properties of the NDF Data Classes**

| Property Name | Descriptions |
|---|---|
| Data | Stores the binary data in conventional MatLab numeric matrix |
| DataInfo | Metadata of the data set |

### 4.2.1. The *Data* Property

The ***Data*** property uses the following conventions to store data:

- Time-series data are in an n-by-1 numeric matrix. Any numeric data type is supported.
- Neural event data (spike time) is in an n-by-1 numeric matrix. Any numeric data type is supported.

- Fixed length segment data are in a 2-by-1 or 3-by-1 cell array. The first cell (n-by-1 numeric matrix) represents index offset. The second cell (m-by-n numeric matrix) represents n segments each with length m. The optional third cell (n-by-1 uint8 type matrix) represents the sorted ID.
- Variable length segment data are in a 3-by-1 or 4-by-1 cell array. The first cell (n-by-1 numeric matrix) represents index offset. The second cell (n-by-1 uint32 type matrix) represents the end position of the n segments. The third cell is a L-by-1 numeric matrix with the n segments saved one after anther in sequence, where L is the number of data points of all the segments. The optional fourth cell (n-by-1 uint8 type matrix) represents the sorted ID. Figure-1 shows the layout of a variable length segment data set in a 4-by-1 cell. The data set with four segments, each with length 3, 5, 4 and 6 respectively. The second cell shows the end point of each segment as 3, 8, 12 and 18.

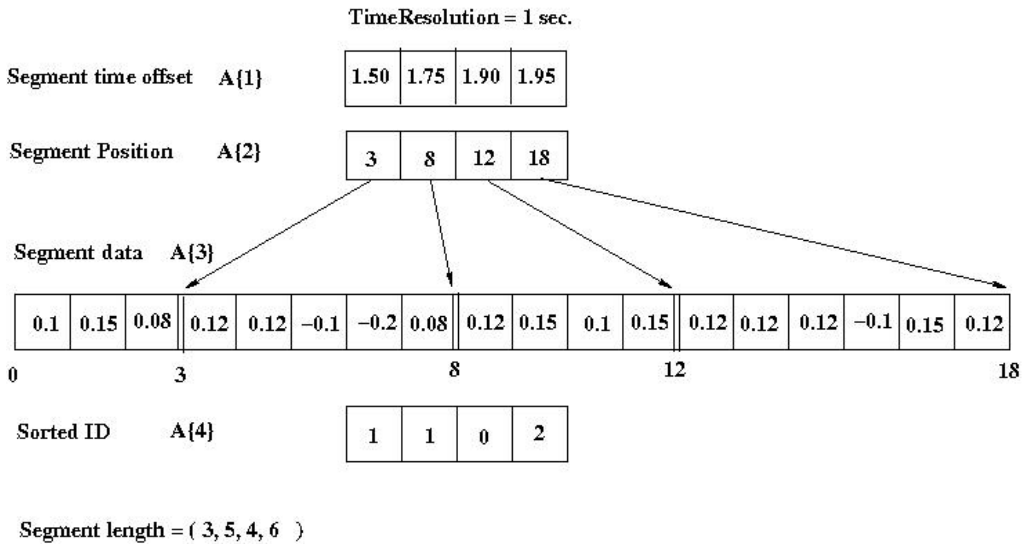

**Figure 1 Variable Length Segment Data Layout**

- Binary experimental event data are in a 2-by-1 cell array: the two cells in the cell array are both n-by-1 numeric matrices. The first cell is the time offset of the events. The second cell is the event values.
- Data values stored in the ***Data*** property maintain their original data type as in the original data file. The real data values must be recalculated according to the ADC settings (for analogue data) or the time resolution (for time stamp). The ADC settings or time resolution are defined in the NDF data info object. The following conventions are applied on calculating the real data values:
  1) If the ADC settings are enabled (see Section-3.2.1-II, III in Page-24 and 25), the real values of analogue data are calculated by:

$$v_i = V_0 + \Delta \cdot V_i$$

where, $v_i$ is the real signal value.
$V_0$ is the zero offset, ***adcZeroOffset.***
$\Delta$ is the quantization step, ***adcResolution.***
$V_i$ is the data value stored in the ***Data*** property

  2) Real values of time stamp are calculated by:

$$t_i = r_t \cdot T_i$$

where $t_i$ is the real time stamp value.
$r_t$ is the time resolution, ***timeResolution***
$T_i$ is the data value stored in the ***Data*** property

### 4.2.2. The *DataInfo* Property

The ***DataInfo*** property is a MatLab structure and each field is directly accessible rather than using the 'get' and 'set' methods.

I. The ***DataInfo*** Property of the Time series data class

| **Field Name** | **Descriptions** | **Data Type** |
|---|---|---|
| itemCount | Number of data points in the object | MatLab int64 |
| dataType | Data type ID for internal use | MatLab int32 |
| varName | Name of the data set | MatLab char |
| dataFilename | Name of the MAT file to store the data | MatLab char |
| targetDir | Directory name the NDF data set located | MatLab char |
| compress | Flag for compressed data | MatLab int32 |
| matLabel | The data element name in a MAT file | MatLab char |
| internalP | The hidden internal parameter for data I/O. | |

II. The ***DataInfo*** Property of the Segment data class

| **Field Name** | **Descriptions** | **Data Type** |
|---|---|---|
| itemCount | Number of data segments in the object | MatLab int64 |
| segLength | Number of data points in each segment for fixed length segment data. -1 indicates a variable length segment data. | MatLab int32 |
| dataType | Data type ID of segment data for internal use | MatLab int32 |
| indexType | Data type ID of index data for internal use | MatLab int32 |
| classIDValid | Flag indicates the existing of valid class ID | MatLab int32 |
| timeResolution | Time resolution in seconds | MatLab double |
| varName | Name of the data set | MatLab char |
| dataFilename | Name of the MAT file to store the data | MatLab char |
| targetDir | Directory name the NDF data set located | MatLab char |
| compress | Flag for compressed data | MatLab int32 |
| matLabel | The data element name in a MAT file | MatLab char |
| internalP | The hidden internal parameter for data I/O. | |

The ***timeResolution*** of a segment data set is equal to the reciprocal of the sampling rate defined within the NDF data info object.

III. The ***DataInfo*** Property of the Neural event data class

| **Field Name** | **Descriptions** | **Data Type** |
|---|---|---|
| itemCount | Number of data points in the object | MatLab int64 |
| dataType | Data type ID of data for internal use | MatLab int32 |
| varName | Name of the data set | MatLab char |
| dataFilename | Name of the MAT file to store the data | MatLab char |

| | | |
|---|---|---|
| targetDir | Directory name the NDF data set located | MatLab char |
| timeResolution | Time resolution in seconds | MatLab double |
| compress | Flag for compressed data | MatLab int32 |
| matLabel | The data element name in a MAT file | MatLab char |
| internalP | The hidden internal parameter for data I/O. | |

IV. The ***DataInfo*** Property of the Experimental event data class

| **Field Name** | **Descriptions** | **Data Type** |
|---|---|---|
| itemCount | Number of data points in the object | MatLab int64 |
| dataType | Data type ID of the event data for internal use | MatLab int32 |
| timstampType | Data type ID of the time stamp data for internal use | MatLab int32 |
| timeResolution | Time resolution in seconds | MatLab double |
| varName | Name of the data set | MatLab char |
| dataFilename | Name of the MAT file to store the data | MatLab char |
| targetDir | Directory name the NDF data set located | MatLab char |
| compress | Flag for compressed data | MatLab int32 |
| matLabel | The data element name in a MAT file | MatLab char |
| internalP | The hidden internal parameter for data I/O. | |

V. The ***DataInfo*** Property of the Generic matrix data class

| **Field Name** | **Descriptions** | **Data Type** |
|---|---|---|
| dataType | Data type ID for internal use | MatLab int32 |
| rank | Rank of the matrix | MatLab int32 |
| dims | Matrix represents the data dimensions | MatLab int32 |
| varName | Name of the data set | MatLab char |
| dataFilename | Name of the MAT file to store the data | MatLab char |
| targetDir | Directory name the NDF data set located | MatLab char |
| compress | Flag for compressed data | MatLab int32 |
| matLabel | The data element name in a MAT file | MatLab char |
| internalP | The hidden internal parameter for data I/O. | |

## 4.3. The Data Object Methods

Methods for NDF data classes are used for manipulating object status and metadata of the data set.

### 4.3.1. Methods for the Time Series Data Class

The public accessible methods for **ndftimeseriesdata** class are listed in the following table,

| **Method Name** | **Descriptions** | **Output Data Type** |
|---|---|---|
| disp | The metadata display method | |
| getnitemtosave | Gets the current upper boundary of the data saving setting | MatLab int64 |
| getsignature | Gets the object signature | MatLab char |
| setnitemtosave | Sets the upper boundary for data saving | |
| setsignature | Sets the object signature | MatLab double |
| cloneparam | Clones internal parameter from source object | |

Command line syntax:

I. Display the meta data

   **objName.disp();**

   will display the contents of the ***DataInfo*** property of the object.

II. Get data saving upper boundary

   ***nItem*** **= objName.getnitemtosave();**

   returns the number of items to be going to save using a multiple-run saving mode.
   ***nItem*** > 0 , multiple-run mode. Data saving finishes when the saved data item count is equal to the value of ***nItem***.
   ***nItem*** = 0 , single-run mode. Data saving finishes on the first saving operation.
   ***nItem*** < 0, multiple-run. Data saving will continue until positive boundary is set.

III. Get the object signature

   ***sig*** **= objName.getsignature();**

   returns the object signature.

IV. Set data saving mode and boundary

   **objName.setnitemtosave(** ***nItem*** **);**

   sets the object saving mode and boundary.
   ***nItem*** > 0 , multiple-run mode. Data saving finishes when the saved data item count is equal to the value of ***nItem*****.**
   ***nItem*** = 0 , single-run mode. Data saving finishes on the first saving operation.
   ***nItem*** < 0, multiple-run mode. Data saving will continue until positive boundary is set.

V. Set the object signature

   ***x*** **= objName.setsignature(** ***signature*** **);**

   ***signature*** is a string represent as the object identifier. This method is usually called by NDF output object internally and the signature is created using rules specified by the class.

   ***x*** is the status of the 'set' operation. ***x***=1 if the 'set' operation is successful. ***x***=0 if the signature string is invalid.

VI. Clone internal parameters from a guest data object.

   **objName.cloneparam(** ***src_obj*** **);**
   **objName.cloneparam(** ***src_obj*, '*noprivate*'** **);**

   The first instance copies parameters both with public and private access from the guest object, ***src_obj*.** The second instance only copies parameters with public access from the guest object. The internal parameters of an NDF data object are the status of a NDF data saving status machine. It is updated on each calling of the "**writedata**" method of a NDF output object. Notice, if private parameters are copied, it will removed from the source object.

### 4.3.2. Methods for the Segment Data Class

Methods for class **ndfsegment** are the same as the methods for the time series data class with an additional method '**getsegment'** for the extraction of a specified segment. This method is applicable to both fixed and variable length segment data. The syntax of the **'getsegment'** method of a segment data object '**obj'** is as follows:

**seg = obj.getsegment(** ***segment_index*****);**

where argument ***segment_index*** is the index of a specified segment.. The method returns a **k-**by-**1** matrix containing the **k** data points of the segment. For fixed length segment data (where property ***objName.segLength*** > 0), the ***n****-th* segment can also be obtained directly from the data property of the object:

**seg = obj.Data{ 2}(:,n);**

For variable length segments (where property **objName.*segLength*** = -1), the ***n****-th* segment can be obtained by applying both the position matrix and the data matrix:

**seg = obj.Data{3}( 1: obj.Data{2}(1), 1);** ( for n=1 )
**seg = objData{3}( obj.Data{2}(n-1): obj.Data{2}(n), 1) ;** ( for n>1 )

Method '**getsegment'** is simply the two-in-one form of the above two calls.

### 4.3.3. Methods for the Neural Event Data Class

Methods for class **ndfneuraleventdata** are the same as the methods for the time series data class.

### 4.3.4. Methods for the Experimental Event Data Class

Methods for class **ndfeventdata** are the same as the methods for the time series data class.

### 4.3.5. Methods for the Generic Matrix Class

There are only **getsignature** and **setsignature** methods available for the **ndfgenericmatrix** class. The syntaxes of the two methods are the same as that in the methods for the time series data class.

# 5. The NDF History Class

The NDF history data class is designed for recording the data processing history chain.

## 5.1. The History Object Constructor

Call the NDF history class constructor to create an NDF history object.

Command Syntax

***histObj* = ndfhistory( 'Command_*line*', *'Comments_and_settings'*,**
***'Start_date_time', 'End_date_time'*);**

Argument ***Command_line*** is the full command line that is used to invoke the data processing application. If the command line argument is not available, an empty string can be used. Argument ***Comment_and_settings*** is a free form string for the comments and settings of the data processing procedure. The third argument ***Start_date_time*** is a string that defines the start date and time of the processing. If the string is empty, the current date time is used. The fourth argument ***End_date_time*** is a string representing the data processing end time. The date time string must be in format **'ccyy-mm-ddTHH:MM:SS'.** If all the four input argument strings are empty, the object will be created with only the start date and time being set. User can also set any of these properties by the specific method after the object is created.

## 5.2. The History Object Properties

The NDF history data class has only one public accessible property, ***Data***, It is an n-by-4 cell array, where n is the number of record within the property ***Data.*** Each record (one row) represents as a 1-by-4 cell array. The first cell element is the start date and time string. The second cell element is the end date time string. The third cell element is the command line string and the fourth cell element is the comments and setting string. Normally, these cell elements should be set using the proper methods.

## 5.3. The History Object Methods

Methods defined in the NDF history data class are listed in the following table:

| **Method Name** | **Descriptions** | **Output** |
|---|---|---|
| disp | Displays the history records | |
| getcount | Gets the number of history records | MatLab int32 |
| getrecord | Gets specified history record | A 1-by-4 cell array |
| setcmdlinestr | Sets the command line string | |
| setsettingstr | Sets the commend and settings string | |
| setenddatetime | Sets the end date time string | |
| setstartdatetime | Sets the start date time string | |
| plus (+) | Operator to append history record to the object | |

Command line syntax:

I. Display the meta data

   **objName.disp();**

   will display all history records within the object.

II. Get history record count

   ***cnt* = objName.getcount();**

   returns the number of the records in the object.

III. Get history record

   ***record* = objName.getrecord( *record_num* );**

   ***record_num*** is the index of the record to extract. The method returns a 1-by-4 cell array containing four strings representing the start date time, the end date time, the command line string and the comments/settings.

IV. Set command line string

   **objName.setcmdlinestr( *cmd_string, record_num*);**

   sets the ***cmd_string*** to the ***record_num**-th* history record. For example,

   **w.setcmdlinestr( 'x2ndf  data.mcd', 1)**

   will record the command line that is used to convert MCD  data to NDF into record 1. This indicates that the data set was created by calling application “x2ndf” and converted from a data set “data.mcd”. This method can only apply to an existing record, i.e. argument ***record_num*** must be valid index.

V. Set comments and settings string

   **objName.setsettingstr( *setting_string, record_num*);**

   sets the ***setting_string*** to ***record_num**-th* record . This method can only be applied to an existing record. The ***setting_string*** message describes the settings and parameters used to generate the data set. Additional description of the processing of the data can also be added to the string. This message is used when the command line is not available, e.g. data is generated from a GUI application.

VI. Set end date/time string

   **objName.setenddatetime( *datetime_str, record_num*);**

   sets the end date/time string  ***datetime_str*** to the ***record_num**-th* record **.**

VII. Set start date/time string

   **objName.setstartdatetime( *datetime_str, record_num*);**

sets the start date time string ***datetime_str*** to *the* ***record_num****-th* record**.**

VIII. Append history record to the object (the **plus** method).

***objName = objName + SrcObj;***
***objName + SrcObj;***

Since **ndfhistory** is a handle class, both instances append the history record from ***SrcObj*** to ***objName***. The plus operator also sorts the history record chain by the start date/time in ascending order. ***Notice, in both cases the operator will change the contents of the first operand. It is critical to use the correct order of the two operands.***

# 6. The NDF Multiple Data Formats Input Module

The NDF MatLab toolbox comes with a data input module to load data with different formats from disk to the MatLab work space. The core of the input module is a windows application, the *NDF data converter*. On top of the NDF data converter is the MatLab code in order for the application to work within the MatLab environment. With the help of the package ‘wine’, the NDF input module will work on all platforms that can run ‘wine’, such as Linux, BSD, Solaris and Mac OS X. For systems that can’t run ‘wine’, the NDF data input module will work only for NDF data input. The NDF Data Input Module can work in pure command mode or invoke a file input GUI for user to select input file from the displayed directory tree. The general syntax of NDF input module looks like this:

**obj = ndfxload( *‘input_filename’*, *‘output_dir’*, *‘output_filename’*, *autodelete, uifile*);**

By default, argument ‘output_dir is set to the system temporary directory; argument ‘output_filename is set to the same as the input data file name with extension ‘ndf’; argument ‘autodelte’ and ‘uifile’ are set to 1. Different settings of the input arguments results in different use modes.

## 6.1. Using the NDF Input Module for data input

For all non-NDF format data sets, the NDF data input module firstly converts the input data set to NDF data format and saves the data to the system temporary directory by default. Then an **ndfread** object of the new NDF data set is created. The temporary NDF data files are removeed once the **ndfread** object is deleted from the MatLab workspace. For NDF data format, the NDF data input module is identical with using the NDF data input class (**ndfread**). The syntax of open a data set using the NDF data input module is as the follows:

**obj = ndfxload( *‘input_filename’* );**

The module converts the input data file to NDF format and creates a **ndfxread** object with the information read from the NDF data set. The **ndfxread** class is identical with the **ndfread** class with the exception of that the **ndfxread** class will remove the temporary NDF data set when the object is destroyed. By default, the temporary NDF data set is save to the system temporary directory. A user can specify the directory for saving the converted NDF data set:

**obj = ndfxload( *‘input_filename’*, *‘output_dir’* );**

This may be useful if the system temporary directory has limited space. By default, the converted NDF data file is set the same name as the input data file but with the extension name ‘ndf’. A user can override this by:

**obj = ndfxload( *‘input_filename’*, *‘output_dir’*, *‘output_filename’*);**

When the NDF Toolbox is used as a user’s desktop tools, it may helpful to have data input GUI open for selecting the input data file. To do this, simply leave the input filename argument of the above command as empty, a file input dialog box will be invoked for file selection:

**obj = ndfxload( ‘’);** **(**or simply, **obj=ndfxload)**
**obj = ndfxload( ‘’, *‘output_dir’*);**

**obj = ndfxload( *‘’, ‘output_dir’, ‘output_filename’*);**

For NDF data input, only the first instance is meaningful, the other arguments will be ignored.

## 6.2. Using the NDF Input Module as an Data Converter

Instead of deleting the output NDF data set on destroying the NDF data read object, the output NDF data set can be reserved for later use after the NDF data read object is destroyed by setting the ‘***autodelte***’ flag to 0. In this case, the NDF data input module works like a multiple data formats to NDF data converter.

**obj = ndfxload( *‘input_filename’, ‘’, ‘’, 0*);**
**obj = ndfxload( *input_filename’, ‘output_dir’, ‘’, 0*);**
**obj = ndfxload( *‘input_filename’, ‘output_dir’, ‘output_filename’, 0*);**

As in the data input mode, if the ‘***input_filename***’ argument is not defined, a file input dialog box will be invoked for user to select the input file.

## 6.3. Using the NDF Input Module in pure command line mode

If the environment doesn’t support graphic interface, a flag can be set to prevent the file input dialog box from invoked. This is done by setting argument ‘***uifile***’ to zero:

**obj = ndfxload( *‘input_filename’, ‘’, ‘’, autodelete, 0*);**
**obj = ndfxload( *input_filename’, ‘output_dir’, ‘’, autodelete, 0*);**
**obj = ndfxload( *‘input_filename’, ‘output_dir’, ‘output_filename’, autodelete, 0*);**

Depending on the usage, argument ***‘autodelete’*** can be as 1 or 0.

# 7. Programming Using the NDF Data I/O Toolbox

The NDF MatLab toolbox is a set of object oriented MatLab classes. To perform a specific task such as data input or data output, an object of a relevant class must be created first. Then the relevant methods can be used to fulfil the required tasks. Before reading the following chapter, you may like to have an NDF data set to hand. If you don't have one, you can create it using the NDF input module, **ndfxload**,

## 7.1. NDF Data Input

All NDF data input tasks are performed by the NDF input class **ndfread** (or its subclass **ndfsvcread**). The NDF input object is created by calling the class constructor and passing a valid NDF header file path to it. For example, assuming the NDF header path is "***D:\Data\dataset1.ndf***", the following command creates an NDF input object ***A*** for the specified NDF data set (dataset1.ndf):

***A* = ndfread( *'D:\Data\dataset1.ndf'*);**

The object is read-only hence its contents are unchangeable. On successfully creation, object ***A*** contains metadata read from the input header file. Object ***A*** also provides methods for data input from the NDF data set specified by the input header file. Method **disp()** can be used to display the main information that are required for starting the data input. The output message on calling method **disp()** looks like:

```
>> A.disp
NDF data input object
General Data Info
NDTF version: '1.0.1'
Data ID: 'C5511C56-FC5E-4DBA-9FDF-343FD85F0C26'
Description: 'Testing NDTF data creation'
Laboratory: ''
Investigater: 'John Smith'
SpecimenID: '20030508X'
RecordID: '0003988A'
Create Date/Time: 2008-03-28T18:18:18
Data Channel Count:
              imageDataCnt: 0
         timeSeriesDataCnt: 1
            segmentDataCnt: 2
        neuralEventDataCnt: 2
              eventDataCnt: 3
      genericMatrixDataCnt: 1
        userDefinedDataCnt: 0
>>
```

The above message shows the contents of the object private data property **GenaralDataInfo** as well as the contents of the public data property **ChannelCount**. The **GeneralDatainfo** data object can be extracted by method **getgeneraldatainfo**():

```
>> genInfo = A.getgeneraldatainfo

genInfo =
General DataInfo:
description: 'Testing NDTF data creation'
laboratory: ''
investigater: 'John Smith'
specimenID: '20030508X'
createDate: '2008-03-28'
createTime: '18:18:18'
recordID: '0003988A'
>>
```

The **getgeneraldatainfo** method returns an **ndfgeneraldatainfo** object that is one of the NDF data info classes described in Chapter-3. Method **disp()** of an NDF data info object normally displays the key parameters only. The key parameters are the ones that may need additional attention or may be the required parameters to describe the data set. To display the full contents of an NDF data info object, method **dispall()** is used. All parameters are stored in property ***DataInfo*** and can be directly accessed using the relevant field of the property. For example, the following command assigns the data set description string to variable ***strDesp***:

```
>> strDesp = genInfo.DataInfo.description

strDesp =

Testing NDTF data creation

>>
```

By design, all field names of a data structure in an NDF object are the same as the names displayed when method **disp()** ( or **dispall()** ) is called. This provides a way to quickly identify the field name from the MatLab environment rather than having to remember each of them. The publicly accessible property **ChannelCount** provides important information about which data types and how many channels of each data type are available within the data set. It is a conventional MatLab structure and hence can be treated as normal MatLab variable. For example, the following command displays the contents of the **ChannelCount** member:

```
>> A.ChannelCount

ans =

          imageDataCnt: 0
     timeSeriesDataCnt: 1
        segmentDataCnt: 2
    neuralEventDataCnt: 2
          eventDataCnt: 3
  genericMatrixDataCnt: 1
    userDefinedDataCnt: 0

>>
```

Based on the number of available channel counts for each data type, a "get_datainfo" method can be used to extract the metadata of each channel. For example, the following command returns an **ndftimeseriesdatainfo** object that contains metadata of the first time-series data channel:

```
>> tmInfo = A.gettimeseriesdatainfo(1)

tmInfo =

Main Parameters:
dataFilename: 'Create_ContData.mat'
varName: 'ch1'
startDate: '2008-03-28'
startTime: '18:08:08'
samplingRate(Hz): 25000
itemCount: 500000
>>
```

Use ***tmInfo*.dispall()** to view the full information from within the object.

The binary data of a data channel can be obtained based on the metadata in an NDF data info object, e.g. the number of items in the data channel. The "get_data" methods are used to read data (full channel or data chunk) from a specified data channel. The "get_data" method returns an NDF data object. Within the data object, the metadata related to the binary data are stored in property "***DataInfo***" and the binary data are stored in property "***Data***" as a normal MatLab numeric matrix. For example, command

```
>> tmData = A.gettimeseriesdata(1)

tmData =

Time series data.
itemCount: 500000
dataType: 7
varName: 'ch1'
dataFilename: 'Create_ContData.mat'
targetDir: '.\'
compress: 0
matLabel: 'Elm1'

>>
```

reads time-series data from the first time-series data channel to data object ***tmData*** (full channel read). ***tmData.DataInfo*** is a MatLab structure contains the metadata and ***tmData.Data*** is a MatLab numeric matrix contains the binary data. You may use, for example, ***plot( tmData.Data(1:200), 1) )*** to plot the data chunk. Command

```
>> tmData = A.gettimeseriesdata(1, 100)

tmData =

Time series data.
itemCount: 499901
dataType: 7
varName: 'ch1'
dataFilename: 'Create_ContData.mat'
targetDir: '.\'
compress: 0
matLabel: 'Elm1'
```

```
>>
```

reads time-series data of the first data channel from 100th data point to the end of the data into data object ***tmData***. To read a data chunk with explicitly specified start and end index, use

```
>> tmData = A.gettimeseriesdata(1, 1000,  2000)

tmData =

Time series data.
itemCount: 1001
dataType: 7
varName: 'ch1'
dataFilename: 'Create_ContData.mat'
targetDir: '.\'
compress: 0
matLabel: 'Elm1'

>>
```

From the displayed metadata, one can see the item count actually read. The following code fragment demonstrates the way to loop through all the time-series data channels:

```
A = ndfread( 'D:\Data\dataset1.ndf');
for i=1:A.ChannelCount.timeSeriesDataCnt
    %read the data of channel i
    Data = A.gettimeseriesdata(i);

    %Your data processing code should replace these  lines
    for k=1:Data.DataInfo.itemCount
        Data.Data(k)  = …
            Data.Data(k) * mod( double(k), 10);  %dummy processing of each point.
    end
    plot( Data.Data( :,  k) );  %plot the data

    % clean up the memory to save space if you don't need the object anymore.
   clear Data;
end
```

## 7.2. NDF Data Output

The NDF output object acts like a status machine to control all data integration, writing and coupling operations. There are two ways to create an NDF output object, **ndfwrite**.

```
W = ndfwrite('output_filepath');
```

will create an empty NDF data output object ***W***. Whilst command

```
W = ndfwrite('output_filepath', 'template_ndf');
```

will create an NDF data output object and load a general info header from the template file. A general info header provides information about the experiment such as investigator and specimen ID that are often the same within a group of researchers or the same experiment. The template file can be any valid NDF header file with the <**GeneralInfo**> xml tag set as template for new data set.

All "**get_datainfo**" methods are the same as for the NDF input class.

The next step towards writing data to NDF is to create an NDF data object of a specified data type. For example,

***T*** **= ndftimeseriesdata( );**

will create an NDF time-series data object ***T***. The object contains metadata stored in data member ***T.DataInfo***. The numerical data (an N-by-1 matrix) must be assigned to the data member ***T.Data*** before the data can be written to file. The numerical data in MatLab contains information about the data size and type. This information will be set to the NDF data object automatically after calling '**writedata'** method of the NDF output object. You can also set the variable name to the data by assigning a name to structure member ***T.DataInfo.varName***. You may also provide label name to write to a MAT file by assigning ***T.DataInfo.matLabel***. However, a user has no absolute privilege to assign a MatLab label name since the NDF output object may change it, if a duplicate label is found. After the binary data are assigned to the data object, call the "**writedata**" method to write the data to file:

**W.writedata(** ***T*** **);**

This writes time-series data ***T*** to a specified MAT file. The NDF output object detects the type of ***T*** and selects appropriate methods to save the data. On return, ***T*** is assigned to the correct metadata and status about the output data set. The NDF output object also registers the data object ***T*** and assigns an ID to it in order to couple ***T*** to an NDF ***DataInfo*** object later. Now you can clean up ***T.Data*** using ***T.Data = []***, but you should not delete object ***T*** at this stage since ***T*** contains the status of the data writing and is needed to pass the relevant metadata to a corresponding ***DataInfo*** object.

The next step is to create an NDF data info object of the same type of ***T***. then to use method **updatedatainfo()** to couple the two objects:

***I*** **= ndftimeseriesdatainfo( );**
***W*****.updatedatainfo(** ***T, I*** **);**

The NDF output object ***W*** will assign the same ID from the **NDF** data object ***T*** to **NDF** data info object ***I***. This procedure will also copy parameters from data object ***T*** to the data info object ***I***. Now metadata that can be assigned automatically has already been added to object ***I***. However, there may be some other metadata (such as the filter settings) that must be added to the object explicitly. Among such parameters, the ADC settings or "***timeResolution***" are mandatory in order for other applications to recover the real data values from the NDF data set. User must make sure that the equations described in Section-4.2.1 are met. The principles are:

1) If the data values in the ***Data*** object are already the real values, disable the ADC settings by setting the NDF data info object member "***adcPrrecision***" and/or "***adcResolution***" to zero for analogue data; or set "***timeResolution***" to 1 for time stamp data. If an NDF data info object is created and kept intact, the ADC settings are disabled and the parameter "***timeResolution***" is set to 1 by default.

2) If the data values in the ***Data*** object are not the real data values (i.e. recalculation based on the equations in Section-4.2.1 is required), the proper parameters of the ADC settings for analogue data or "***timeResolution***" for time stamp data must be set.

In this example, the ***Data*** object ***T*** is the real data values and the data info object ***I*** created from the class constructor has the ADC settings disabled by default. We don't need to do any thing to the ADC settings. For time series data (also for the segment data), parameter "***samplingRate***" is required in order for the data set to have right timing setting. This can be done by directly accessing the member of structure ***I.DataInfo***. For example, to set the sampling rate, use command

**_I_.DataInfo.samplingRate = 50000;**

This sets the sampling rate to 50000 Hz. After all metadata are set for data info object ***I,*** call

**_W_.adddatainfo( _I_ );**

The NDF output object ***W*** will add ***I*** to the ***DataInfo*** list and make a record about ***T*** and ***I*** to prevent further appending of the same data for a second time. We now have written one channel of time-series data together with the metadata to file. More data can be added in the same way. Then we need to set up the history of the processing:

**_W_.history = ndfhistory( _'cmdline', 'comment'_ );**

and append the previous history data to ***W***:

**_W_.history = _W_.history + _X_.history;**

This assumes that the previous history data was stored in NDF input object ***X***. If object ***W*** is created using ***'append'*** mode, the previous history record is already loaded into property ***W*.history**. In this case, a new record should be appended to the output object ***W*** directly, for example:

**_W_.history = _W_.history + _ndfhistory( 'cmdline', 'cmdment')_;**

Finally, contents in the NDF output object that contains all information on the NDF header must be written to the NDF header file to finalize the data output. Command

**_W_.writendfheader( );**

will write out all metadata stored in the object to the specified NDF header file. After the writing, the object is locked to prevent further writing to the same NDF file. If it is necessary, call

**_W_.unlockoverwritten( );**

to unlock it and obtain is again in order to overwrite the file.

The following is a full example to create an amplitude modulation time series signal and write it to an NDF data set.

```
w = ndfwrite('D:\tmp\ndtf\test.ndf');        %Create an ndfwrite object
a = ndftimeseriesdata;                       %Create an ndftimeseriesdata object
```

```
n = pi/100:pi/100:1000*pi;              %Create a AM signal segment.
a.Data = ((1 + 0.5*sin( 0.2 * n)).* cos( 2*n))';
a.DataInfo.varName='am-1';              %Naming the variable
w.writedata(a);                         %Write the time series data to file
info = ndftimeseriesdatainfo;           %Create a data info object
w.updatedatainfo( a, info);             %Couple the data info with the data object
info.DataInfo.samplingRate=1000;        %Set the sampling rate, may also be others
w.adddatainfo(info);                    %Add the data info to ndfwrite object
h = ndfhistory;                         %Set the history record message
h.setcmdlinestr('Command line that create the data put here');
w.history = h;                          %Add history to the ndfwrite object
w.writendfheader;                       % write the NDF header to file. Done.
```

In many cases, the data set is too large to write to file at once (one-run) as in the above example. The NDF output object supports two writing modes: one-run and the multiple-run mode. To initialize the multiple-run mode, set the total number of items to be saved to the NDF Data object before calling methods ***writedata():***

**_T_.setnitemtosave( _nnnn_ );**

then call ***W.writedata( T )*** multiple times until the total item sum reaches the set number ***nnnn***, where ***T*** contains number of items less than ***nnnn*** For data streaming, the total number of items is not known in advance. In this case, set the total item number to -1 then write the data multiple times. However, explicitly finalizing must be applied to finish the saving of the data. Before the last chunk of data is going to be written to file, call

**_T_.setnitemtosave( _nnnn_ );**

where ***nnnn*** is the number of data items saved up till now plus items in the last chunk of data. Then call ***W.writedata(T)*** to save the last chunk and finish the writing. One important issue on writing data in multiple-run mode is that ***DataInfo*** member of the NDF Data object must be kept intact and should not be deleted. This is because the data saving status is stored in the NDF Data object and updated after each calling of method **writedata()**. Only the ***Data*** member of the object can be changed. The following example shows how data can be read from an existing NDF data file and write to a new NDF file using multiple-run mode. Each "read" operation will destroy the ***DataInfo*** member of the data object. This example shows how the original information in the ***DataInfo*** member can be kept and restored. The example assumes that NDF file "Create_Test5.ndf" exists. The first time series data channel is read then written to a new NDF file.

```
d = ndfread( 'E:\Ndtf\Create_Test5.ndf' );     %Read the input NDF header file
w = ndfwrite( 'E:\temp\data1\data1.ndf' );     % Create an ndfwrite object to write
info = d.gettimeseriesdatainfo( 1 ) ;          %Get data info of time-series data channel 1
idx = 1;                                       %The start index to write
temp = ndftimeseriesdata ;                     %Create a  data object for saving  the status
temp.setnitemtosave( info.DataInfo.itemCount )          %Set total item to save
temp.DataInfo.varName = info.DataInfo.varname;   %Set the varname to be
                                                 %the same as the original data

for  i=1 : floor(  double(info.DataInfo.itemCount)/20000 )      %   Chunk size to
                                                                % write is 20000
     x  = d.gettimeseriesdata( 1, idx, 20000 * i ) ;      %Read 20000 points
     idx = idx + 20000;
     x.cloneparam( temp );                         % Recover the status back to x
```

```
        w.writedata(x);                              %Write the first chunk

        %Since the afterward reading will destroy the status of the status machine, we
        %need to remember the current status before read the next chunk.
        temp.cloneparam( x );                   %Clone the status from x to object temp
    end

    %Write the remaining data tail.
    tail = mod( double(info.DataInfo.itemCount), 20000 );
    if tail>0
        x = d.gettimeseriesdata( 1, idx,  tail );
        x.cloneparam( temp);
        w.writedata( x );                  %This should finish the writing and return 1
    else
        x.cloneparam( temp);               %Get the original parameters back to x.
    end

    %Couple the data info object with the data.
    %This also passes some information from the data object to the data info object.
    w.updatedatainfo( x, info );
    w.adddatainfo( info ) ;          %Then add the data info to the ndfwrite object

    % Add the new message for the processing as history record
    w.history.setsettingstr( 'NDF MatLab toolbox command line converted data
    set.' )

    %Append the history record of the mother data set to keep a
    %full history record
    w.history + d.history
    w.writendfheader               %write the NDF header to finish to processing.
```

## 7.3. The NDF Data Input with Service input Argument parser

The NDF class **ndfsvcread** is a subclass of NDF class **ndfread.** All methods and properties defined in superclass **ndfread** are also available in this class. Although it can work for any data type, the **ndfsvcread** class is designed to work implicitly for a specified data type such as time-series data that is defined from the service input argument file. All methods named with key word "spec" are all for the specified data type (the default data type). For example, rather than call ***gettimeseriersdata(1),*** an **ndfsvcread** object specified for time-series data can simply call ***getspecdata(1)*** to read data of the first channel. This makes the class able to handle all NDF data type in a unique manner. Instead of using an NDF header file path as the class constructor argument, the **ndfsvcread** uses the path of a CARMEN service input argument file as the input of the class constructor. The CARMEN service input argument file may be generated from the CARMEN service portal and defines parameters required by the service. It must also contain a valid file path of the NDF header file. The following command creates an **ndfsvcread** object (where***'Service_Input_FilePath'*** is the path of a CARMEN service input argument file):

***S* = ndfsvcread( *'Service_Input_Filepath'*);**

The service input argument file defines a list data channels for the service to process. These channels can be of different groups. The input argument file use xml tag <member> to identify the group of channels. One of the main roles of the **ndfsvcread** is to parse the input xml file and provide the channel index list for the service to process the data. The following code fragment shows how to work for all defined channel regardless the member IDs. It will work whether or not the <member> element in the input file is specified.

```
 x = ndfsrvread( './srvInpuf.xml')         %Create the object
indexList = x.getallchannelidx();         %Get all channel list regardless the member ID
for cnt=1:size(indexList, 1)              %Loop through all the channels.
data = x.getspecdata( indexList(cnt) );   %Get the data
... other processsing                     %Dummy processing code
clear data;                               %Release the memory
end
```

The following example will work for at least one <member> element specified on the input file:

```
 x = ndfsrvread( './srvInpuf.xml')      %Create the object
%Processing the first set of data specified by the first <member> element
indexList = x.getchannelidx(1);
for cnt=1:size(indexList, 1)             %Loop through the channel index list
data = x.getspecdata( indexList(cnt) );  %Get the data
... other processsing                    %Dummy processing code
clear data;                              %Release the memory
end
```

## 7.4. Some Tips

### 7.4.1. Try not to create multiple instances of the same variable

This example creates a dummy NDF segment data and writes it to an NDF file. The segment data stores in a 2-by-1 cell. The first cell is a 60-by-1 int32 matrix representing the timestamp (index offset) of the segment data and the second cell is a 20-by-60 numeric matrix for the segment data. Rather than to create two separate variables then assign to the ***Data*** member of the NDF data object , this example shows a way to directly access the cell elements for the creation of each data member.

```
w = ndfwrite('D:\tmp\ndtf\test.ndf'); %Create an ndfwrite object
a = ndfsegmentdata;                   %Create an ndfsegmentdata object
a.Data = {[]; zeros(20,60) };         %Create a 2-by-1 cell array with the second cell
                                      %pre-allocated as 20-by-60 doubles matrix
a.Data{1} = int32(1:60)' * 2 ;        %This simulates creating the time stamp with 60
points.

%This simulate to assign values to a 20x60  matrix as the segment data
for i=1:20
for j=1:60
a.Data{2}(i, j) = i * j;
end
end

a.DataInfo.varName='seg-1';                 %Naming the variable
```

```
w.writedata(a);                          %write the data.
info = ndfsegmentdatainfo;               %Create a data info object
w.updatedatainfo( a, info );             %Couple the data info with the data object
info.DataInfo.samplingRate=1000;         %Set the sampling rate, may also be others
w.adddatainfo(info);                     %Add the data info to ndfwrite object
h = ndfhistory;                          %Set the history record message
h.setcmdlinestr('Command line that create the data put here');
w.history = h;                           %Add history to the ndfwrite object
w.writendfheader;                        %Write the header to file . Now all got done.
```

### 7.4.2. The Int64 data member in NDF data I/O Object

The NDF API uses a 64-bit data type for the data item count. Since versions of MatLab earlier than 7.11 (Release 2010b) do not support arithmetic operations for 64-bit integer, data processing functions that use 64-bit data type in those earlier versions should firstly cast to double type if arithmetic operations are required.

### 7.4.3. Access Data by Time Interval

The NDF Data I/O toolbox provides methods to convert a time interval to indexes. This allows applications to extract data chunks for a specific time interval rather than by index range. This method is useful in particular for neural event data and segment data. The following code piece shows how to extract a data chunk by specified time interval.

```
d = ndfread( 'E:\Ndtf\Create_Test5.ndf' );          %Create an ndfread object

%Get start and end indexes of the first neural event channel corresponding to
%time interval [10.1, 20.35). Firstly convert time interval to index range
[ startIdx, endIdx] = d.getneuraleventdataindexes( 1, 10.1, 20.35);

%Then get the data chunk within [10.1, 20.35) by index range
nvData = d.getneuraleventdata( 1, startIdx, endIdx);   ....
```

To get the time value of the last data point of a specified channel, using the "get_duration" methods

```
% Get the maximum time offset of neural event data channel  1.
duration = d.getneuraleventdataduration( 1);
```
[ Or using. ***duration* = *d*.getduration( '*neuralevent*', 1);** ]

This parameter provides information about the time instance of the last data point, i.e. the valid value range of parameter ***timeTo*** for calling "get_indexes" method.

### 7.4.4. Using the compress stream

The NDF internal data types that use MAT file as host can be saved as compressed stream. The NDF API supports partial data read both for compressed and uncompressed data. Data reading is completely transparent to the user. To save a data channel as compressed stream, simply set the ***DataInfo*** member "***compress***" of an NDF data object to 1. For example, if ***tm*** is an **ndftimeseriesdata** object, to save it as compress stream, ***tm.DataInfo.compress*** should be set to 1 before calling the "**writedata**" method of an NDF output object:

```
tm.DataInfo.compress = int32( 1 );
```

Saving data as a compressed stream can reduce the disk space. However, it may take much longer time for partial data reading than that for uncompressed data. A balance must be made depending on the nature of the data set.

### 7.4.5. Using appropriate numeric data type to reduce the file size

Most of the NDF data types support arbitrary numeric data types as the storage type. The real value of an NDF data set is obtained by taking account of the ADC settings (or time-resolution for time stamps) of the data. This provides means to store data in more efficient way. For example, a digitized data may be stored as 16-bit integers together with a set of ADC parameters. This will reduce the data size by a factor of 4 compared to using double floating point data as the storage type. A time stamp can also be saved as an integer together with a factor of time-resolution. Whenever possible, try to save the data in the more efficient way. This not only reduces the storage space but also saves time on downloading data over the network.

The NDF API and NDF toolbox save data to a MAT file of the same type as the NDF data object property ***Data***. To use a specific data type as the storage data type simply set the ***Data*** property of an NDF data object to the type preferred. For example,

***tm*.Data= ones( 1000, 1, 'int32');**

sets property ***Data*** as MatLab int32 type. When the NDF output object "**writedata**" method is called, the NDF API will save the data as int32 type that use 4-byte per data point. Whilst,

***tm*.Data= ones( 1000, 1);**

sets property ***Data*** as MatLab double type. When the NDF output object "**writedata**" method is called, the NDF API will save the data as double type that use 8-byte per data point.

# References

[1]. "The Neurophysiology data Translation Format (NDF)" http://www.carmen.org.uk/standards/CarmenDataSpecs.pdf- retrieved 10th May 2010.

[2]. "Mat-File Format 7.10", The MathWorks, Inc.(2010). http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf.

# Indexes