BEAM

Blacknest BDS Development
Programming Manual – 2.2.x
Preliminary

Project	Blacknest
Date	2020-06-25
Reference	blacknest/BdsDevelopment
Author	Dr Terry Barnaby

Table of Contents

BEAM

1. References

● The BEAM Blacknest support website at: https://portal.beam.ltd.uk/support/blacknest. This provides detailed information on the BDS system. Most of the systems documentation is at: https://portal.beam.ltd.uk/support/blacknest/files/bds/doc.

2. Introduction

This document provides information on developing programs for the BDS system. You should first read the document: BdsUserManual.pdf that provides an overview of the system and how it functions.

There is detailed, doxygen generated, reference documentation for the API provided at bdsApi as well as some examples in C++ and Python in the /usr/bds/bdsExamples directory.

The BDS system and its API's are written in C++ using an object orientated style. The API also has Python 3 and partial PHP bindings available.

3. API Architecture

The BDS API uses an object orientated architecture that employs the beam-lib class library for lower level functionality. The beam-lib class library provides functionality for basics such as string, list and array handling as well as network sockets interfaces and other items.

The BDS API generally has a class to represent each seismic entity. So for example there are the classes Station, Channel, Response, Digitiser etc. Each of these classes has data and function members suited to their nature.

Most BDS class member functions and generic library functions return a BError object to indicate the status of the function call. The BError object has an integer error number value and a string describing the error. When the error number is 0 this indicates all was ok. +ve error numbers indicate an API level error has occurred. Negative error numbers are from the underlying Linux system.

The BString class is used extensively to store variable length ASCII strings. BStrings can be appended and compared easily and have a great deal of additional functionality. The C++ operators are overridden to provide this functionality and to return the BString as a "const char*" for functions that require a traditional 'C' string. The BString class has a BString::str() method that will return its internal string as a "const char*" for printf() calls or for other uses that require this.

The beam-lib library provides integer and floating point types such as BInt32, BUInt16, BFloat64 that are based on the appropriate lower level 'C' type.

In many cases, especially for the Metadata classes, objects of these BDS classes are stored in a MySQL database table with a single database record describing the data contents of the object in question. Sometimes a BDS's class object is stored across multiple data base tables where this make sense. When stored in a database table, the record has fields matching the classes data member names with a field type to match. All of these classes have an "id" field which is an unique integer value describing a particular record in a database table. This allows particular object instances to be manipulated in a generic way.

BEAM

3.1. The Beam-lib class library

The BDS API is heavily object orientated and based on the beam-lib C++ class library. This library provides low level classes for the manipulation of strings, lists arrays as well as system independent network sockets etc. It is documented in: libBeamApi and the bdsApi reference documentation links to it.

The core C++ classes used are:

Class	Description
BString	A variable length string class used for storing, passing and manipulating ASCII text strings.
BError	Most functions return a BError object to provide the status of the functions operation. A BError object has a number and a string. The error numbers are listed in the API with 0 indicating Ok. The string is a human readable error message.
BList<Type>	This is a generic doubly linked list object that can be typed to store any other C++ object.
BArray<Type>	This is a generic contiguous memory array object that can be typed to store any other C++ object.
BDict<Type>	This is a dictionary object that can be typed to store any object indexed by a string.
BTimeStamp	A date/time to microsecond resolution
BUInt32, BInt32 ...	General low level integer and floating point types.

3.2. BDS Dates and Times

The BDS uses the beam-lib BTimeStamp class to store datetime information. This stores the datetime to the nearest microsecond based on the UTC time standard. It normally returns an ISO 8601 string representation. There are some special datetime's that are used:

- **0000-01-01T00:00:00:**   Means the datetime has not been set.
- **0001-01-01T00:00:00:**   Means the beginning of time.
- **9999-01-01T00:00:00:**   Means the end of time.

The beginning of time and end of time values are often used to represent the period for which Metadata is valid when no changes are made it it.

The BTimeStamp class has functions to set and get it in Unix epoch time as well as set, return in various ASCII string formats and other functionality. See the doxygen documentation for more details.

3.3. Errors and BError

As mentioned, most functions return a BError object to provide the status of the functions operation. A BError object has a number and a string. The error numbers are listed in the API with 0 indicating Ok. The string is a human readable error message.

BEAM

Most of the possible error numbers are defined in BError.h. There are 4 sets of error number ranges:

- 0 – 63: Standard Beam-lib error numbers.
- 64 – 95: BDS special system errors
- 96 – 127: User program errors
- Negative: Underlying OS errors

If you need to generate an error you can add your own error numbers starting at ErrorUserBase (96) or simply use the error number ErrorMisc (1).
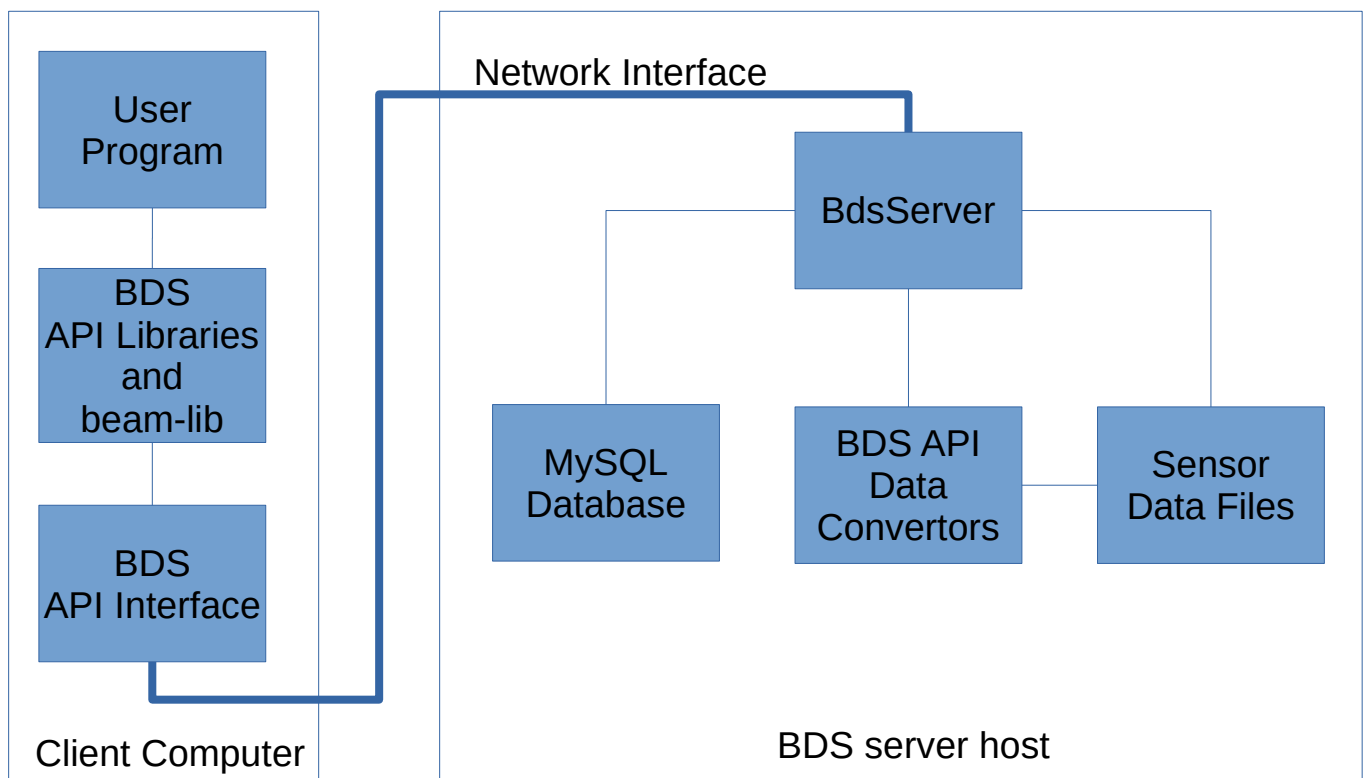
4. API Overview

The Blacknest Data System (BDS) is designed to provide storage and access to seismic data sets. The original seismic data is available in many different formats and the BDS unifies these formats into a generalised system for standard user access. The BDS provides a generalised data access API that allows access to any data format and allows conversion between data formats.

There are two types of data stored in the BDS:

- **Seismic sensor data:** This consists of a set of timestamped data samples from a measuring instrument. Typically this is earth movement measurements but can be other quantities such as infra-sound, temperature etc. Seismic data is stored in timestamped blocks of samples which is stored in files on a computer server system. This file data may have some low level Metadata included in the binary files which is generally not used but available for special uses and validation of the overall Metadata.

- **Seismic Metadata:** This provides extra information on the sensor data. It includes information such as physical location of the instrument, sampling rate, calibration factors and frequency response of the signal chain. It is stored in a MySQL database on the main BDS computer server. Most of this information rarely changes, perhaps only when a sensor is re-aligned or an instrument is updated or a new calibration performed.

All of this sensor data and Metadata is stored on the main BDS server and accessed through the, gate keeper, BdsServer daemon program via a set of network API's.

BEAM

The BdsServer program provides access to all functionality. The functionality is accessed via function calls on one of three access objects. Various BDS client programs use this API to manipulate the Data and Metadata of the BDS system as well as user programs. As well as providing API functions for sensor data and Metadata access there are functions for login authorisation and overall system management.

The BdsApi has been developed using the BOAP (BEAM Object Access Protocol). This provides a simple but powerful Object Orientated RPC (remote procedure call) mechanism. The BdsApi is written in a high level interface definition language (IDL). The bidl tool generates the client and server side 'C++' and Python interface and implementation files for the API. These are then provided as a set of 'C++' header files and a binary library file for the clients to link to. This C++ library is then converted by the SWIG system into API libraries for other software languages including Python and PHP. The BOAP system employs a simple network BOAP name server process that provides a translation between object names and network IPAddress/Socket numbers. The BOAP name server runs on the BDS server host. More information on the BOAP system can be found in the beam-lib documentation.

The BDS API provides a number of data storage classes and three interface objects. The interface objects are:

1. **Bds::DataAccess** BDS Data API: This will provide read only access to the data and meta data. It will be used by the AutoDRM email and Web systems as well as for program access to the data.
2. **Bds::DataAddAccess** BDS DataAdd API: This will provide read and restricted write access to enable the adding of data to the system. It will not allow deletions of data to be performed. It is designed to be used by manual and automatic data adding programs.
3. **Bds::AdminAccess** BDS Admin API: This will provide full read/write access to the data and meta

BEAM

data as well as administrative configuration information.

These API's share the same functions with the Bds::DataAccess API have the minimal set of read-only functions and the BDS:AdminAccess API having all of the possible functions.

All BDS functions are accessible via these API's, given a suitable user/password is available, to any client program.

The Python version of this library available, that sits on top of the C++ API library allows Python programs to directly access the BDS system in the same way as a 'C++' program.

4.1. Terminology

Some special terminology used in the API includes:

Term	Description
Array	Name of an array of Stations having seismometers and other sensors. This has a location. A single name is given for both Arrays and Stations.
Station	A single measuring site that can have a number of instruments and provide a number of channels of data. A station has a location which is time dependent. It is also sometimes referred to as a Site. The Stations name is designated by a three to five letter code in international catalogues. The location may vary within 1 km of the originally registered position, because the instruments are moved or the position is re-surveyed.
Instrument	A single measuring instrument that can provide a number of channels of data.
Channel	A single measuring channel that will return a single set of data. It has an associated Digitiser and Sensor. Each channel has a calibration value and sample rate.
Digitiser	A digitiser that digitises the analogue signals from a Sensor. A digitiser can support multiple sensors and also have multiple output channels at differing sample rates. It has an associated set of filter frequency responses for the Anti-aliasing and other processing functions.
Sensor	A per channel, seismometer, hydrophone or other sensor measuring time-dependent data, e.g. temperature, wind speed. It has a frequency response and other associated parameters such as installation angles.
Network	This defines an organisation that is responsible for a number of seismic Arrays/Stations or provides data for a number of Arrays. Each Network organistaion may have its own database with different settings for various parameters such as CalibrationFactor.
PAZ	Poles and Zeros table for frequency response
FAP	Frequency Amplitude Phase table for frequency response
FIR	Finite Impulse Response coefficient table for frequency response
CalibrationFactor	A measured or set value that defines the gain of a channel from physical Earth movement to numeric value at a particular frequency. Normally in nanometers/count.

BEAM

CalibrationFrequency	The frequency at which the CalibrationFactor is valid. Sometimes given as a period rather than frequency.
Event	A seismic disturbance, e.g. an earthquake or explosion.
Arrival	An "arrival" is a signal from an earthquake or explosion recorded at a station.
Synchronous	When multiple channels of data are returned they are considered to be synchronous if their sampling clock is shared. This allows the data to be returned in a sample multiplexed fashion (all the channels sample data in one row per time stamp)
Sample Multiplexed	This is a format of data when multiple channels of data are multiplexed into a single set by sample. That is all of the channels sample data are stored in one row per time sample. The channels have to be syncronously or near synchronously sampled for this storage method. Older seismic data such as BDRS are stored this way.
Channel Multiplexed	This is a format of data where each channels data is independently stored with respect to other channels of data. The data may have differing sample rates or sample clocks, but could also be synchronously sampled. All data can be stored using this method.
Segment	A Segment of data is a contiguous set of data with now time gaps. There are often missing blocks of data so gaps in time. A set of requested data would be segmented at each time gap. The system will also, by default, segment data at file boundaries unless suitable dataOpen() options are provided. Metadata can also be segmented. The segment boundaries would be at the point in time where some aspect of the Metadata changed.

4.2. Seismic Data

The BDS API is based on the following common sensor data model.

- Seismic sensor data is split into individual channels each containing a single data stream.
- Data files or streams may contain multiple channels of data.
- Each channel has a set of meta data associated with it. This defines things like the StartTime, EndTime, Network, Station, Channel, SampleRate, CalibrationFactor etc.
- The data for each channel is split into segments. Each segment may be for a different period of time and/or a different network or source. Segments are split where there is a time discontinuity between individual blocks of data ie a gap in the available data.
- Each channels segment data is split into blocks. These may be of variable or fixed length. Each block has start and end time stamps associated with it. The BDS keeps the original sources blocks timestamps and sizes intact to preserve data integrity.
- Each data block may have special meta data information associated with it depending on the original data format. This is generally for user consumption only. It contains things like tape quality data from the TapeDigitiser etc.
- If there are multiple channels, these may be synchronously sampled or independently sampled.
- Data can be multiplexed by sample or by channel. Sample multiplexed means it stores a set of samples for each sample time. Channel multiplexed means each channels data is separate from the

BEAM

others. Only synchronously sample data can be multiplexed by sample.
- Raw data has an original sample format of Int16, Int32 or Float32. The BDS stores the samples with the same format as the original data. The number or different sample formats can be extended in the future.
- The BDS stores all data in its own DataFileBds format. It is able to store the data in a compressed format in these files using the Canada compress method. However this is not currently used the intention being to make it easier to recover data in the event of data storage errors.
- Imported data formats can be compressed using various schemes such as: Canadian, STEIM, GCF, CM6 etc. The BDS Data converts support reading and writing to the various seismic file formats using the appropriate compression methods.
-
- The BDS system allows a set of channel data, from the same time period, to be returned as a single set of sample multiplexed data if the channels are synchronously sampled.

4.3. Seismic Metadata

The BDS seismic Metadata consists of information like Station's and Channel names, Station locations, calibration factors, frequency responses and instrument information etc. This information comes from various sources including other databases and paper sheets. It is stored in the MySQL database on the BDS server host.

For each entity of information, such as a Calibration, there exists a BDS API object describing the data associated with it. For example a Calibration has a scaling factor, the frequency at which the scaling factor is valid, the sample rate etc. In most cases these Metadata objects are stored in individual MySQL data tables but some are effectively stored across many tables.

The core Metadata items include:

Object	Description
Network	This defines a Seismic Network organisation. It provides a selector for data and Metadata from a particular seismic group. For Blackenst sourced and managed data this is "BN". A Network object has a list of the Stations it has. Its name is used as part of the BDS data selection target.
Station	This defines a seismic station. A Station could be an individual station or an array of stations. Each Station has a set of Channels associated with it. Its name is used as part of the BDS data selection target.
Location	The physical location of a Station. This defines the physical location of the station using, typically, WGS84 longitude and latitude parameters. It also defines the stations elevation and if part of a seismic array, the offset with respect to the arrays centre location.
Channel	This defines a seismic data Channel. It is one of the core Metadata elements. It defines a particular channel of data that will have associated sample data. Its name is used as part of the BDS data selection target.

BEAM

Source	This defines a seismic data Source. A Seismic data source allows different sources of data for a channel to be described and allows different Metadata sets to be used with the different data sources. It might be that there were two different digitisers in use or one data set was received real-time though a particular data processing chain while the other was via CD medium with a different processing chain.
Calibration	This class defines a calibration setting. Each channel has a samplingFrequency and a calibrationFactor (scaling factor) associated with it at a particular calibrationFrequency. There may be additional calibration information such as the depth of the sensor and its positional angles.
Response	This defines a seismic frequency response characteristic for a channel. There can be multiple stages in a channels data processing path, this response data describes one of those stages frequencies responses. The stage parameter defines which stage it is for (1, 2, 3, …). Stage 1 is reserved to store an overall channel response. A response can be in the form of an array of poles and zeros, a FAP array, or a set of FIR coefficients.
ChannelInstrument	This describes an Instrument that measures the channels samples. It consists of a Sensor and Digitiser each with there own parameters.

4.4. Selecting Sensor Data or Metadata

In order to access Sensor Data or Metadata from the BDS system, the set of channels required needs to be selected. The BDS system allows for the selection of multiple channels of data from various sources over a given time period. To achieve this the BDS system is passed a Selection object. The Selection object has the following main fields:

Item	Description
startTime	The start time to the nearest micro-second. 0001-01-01 is used to define the start of time of the Sensor or Meta data required. The first sample of sensor data will have a timestamp equal to or after the given startTime. This first sample may be some time past the given startTime if no data is present. The returned Datainfo and DataBlock objects will return the actual startTime of the data.
endTime	The end time to the nearest micro-second. The last sample of Sensor or Metadata will be before this time. This time value is thus just past the period wanted. 9999-01-01 is used to define the end of time. If the actual data finishes before the given endTime then the actal data endtime may be well before the endTime requested. Note that the endTime field in a DataBlock is actually time time of what would be the first sample in the next block ie the endTime is after the data contained within the block.
channels	The List of SelectionChannel Objects

Each SelectionChannel has the following fields:

Item	Description

BEAM

network	The network the data is from
station	The Array or Station name
channel	The Channel name
source	The data source (Master, Tape, Processed etc)

Any number of SelectionChannel entries can be included. Each attribute can be set to a null string, which is used as a synonym for any, or a suitable value for selection. Standard regular expression characters, such as the wild card character "*" and "^[a-z]*?.*" etc. can be used in any of the fields. The BDS system will expand the given set of SelectionChannel objects to a set defining unique channel Sensor data or Metadata segments in the system. Note that the pattern matching is case sensitive.

When selecting Sensor data, the data selection scheme will return a DataInfo object describing the data selected. This will contain a number of separate channels of data. Each channel can have multiple segments of data. Segments of data are based on time periods where the data is split into multiple files or where there are multiple sets of data from different data sources (for example Digital and/or Tape).

When selecting Metadata, the selection scheme will return a set of suitable Metadata matching the selection criteria. This may be stations, channels, responses etc. There could be multiple items of each in time periods bounded by Metdata changes.

5. The BDS Python API

The BDS Python API is built on top of the standard BDS 'C++' API using the SWIG API generator. Thus all of the standard BDS C++ API functions and documentation applies. However there are some differences due to the language facility and syntax differences.

The Python language is interpreted rather than compiled and does not require strict types like 'C++'. This can help speed up the development of simple tools and programs, but it can result in less robust and less maintainable code.

To use the BDS API library in Python we provide two alternative modules: the "bdslib" and the "bdslibe" modules. The "bdslibe" provides an exception based API as noted below.

The SWIG system wraps the BDS C++ objects in a Python object layer. You can then interact with the C++ BDS objects from Python in the same way as you would have done in C++ apart from a few differences including:

1. Reference returns: 'C++' allows references/pointers to be passed as function arguments which allows functions to return values into arguments passed. Python does not always support this. Instead Python provides the ability for functions to return multiple items on the left-hand side of the function call. When using a BDS API call that in 'C++' returns items by reference, the Python equivalent will have these returned on the left hand side. For example:

   err = bds.channelGetList(selection, channels);   // C++

   (err, channels) = bds.channelGetList(selection);   # Python

2. Testing for error returns. All BDS API calls return a BError object. This provides information as to if the function completed successfully or if there was an error. The BError object contains both an error number and an error string. 'C++' allows an "if" statement to have an assignment operator.

BEAM

This makes returning and checking errors quite concise. Python (pre 3.8) does not allow this and requires a separate assignment and if statement. For example:

```
if(err = bds.channelGetList(selection, channels)){
        return err;
}
```

In Python:

```
(err, channels) =  bds.channelGetList(selection);
if(err):
        return err;
```

3. Exceptions: The BDS API does not use 'C++' exceptions. As the BDS functions all return a BError object we have provided an alternative Python API library that uses exceptions for all BDS API calls instead of returning a BError object. With this calls can be written thus:

```
try:
        channels = bds.channelgetList(selection);
except ExceptionBError as e:
            print("Execption", e.number, e.string);
```

Where possible the BDS Python API wraps the underlying C++ objects so they can be manipulated in a Python like way. So for example the BList and BArray types can be manipulated as if they were internal Python lists, at least for basic list functionality. Also BStrings can be set and returned as regular Python strings.

There are also a few "gotchas":

1. C++ provides the concept of template classes that are uses for things like generic lists, arrays and dictionaries that can store any object type. In order to use these Python needs to know the specify type of each used template. So the BDS SWIG Python interface provides Python classes for the normally used templates. For example there is a BListResponse type that is a C++ BList<Response> template class.

2. In C++ parameters can be passed too and from functions by reference/pointer or by value (a copy). In Python everything is passed by reference but a reference counter is provided on each object to make sure they are kept around as long as there is an active reference to them. The SWIG Python interface wraps the C++ objects in a Python type object and manages reference counters at this level as Python would normally do. However if you return a part of a C++ object from a function, perhaps a member that is a BList, then the underlying C++ object could disappear when the top level C++ object goes out of scope. So if returning a portion of a C++ object you will need to make a copy of the object you are returning. For example, if returning a list of Responses from a ChannelInfo object you would have to do: "return BListResponse(ci.responses); rather than "return ci.responses;". This creates a copy of the BList<Response> list that is returned from the function. Another option is to set the "thisown" parameter on the toplevel object to 0. This keeps that object in memory. That will likely create a memory leak however.

6. Using the BDS API

We will use the Bds::AdminAccess interface to access the BDS system. The other interfaces are identical although with a reduced number of functions to limit their operation.

All of the BDS client programs including the user orientated bdsAdminGui, bdsDataAccess etc. programs, the data importers bdsImportData, bdsImporStreamCd etc., admin programs bdsControl etc. all

BEAM

use these API's to carry out the operations wanted. So all of the BDS functionality is accessible via these interfaces.

As stated the API make use of the Beam-lib class library. One of the classes provided by the beam-lib library is the BError class. This is used to return the status of a function request. It contains both an error number and a string. The error numbers include generic basic error numbers, BDS specific error numbers and Linux system error numbers (negative values). The error number ErrorOk (0) means no error occurred. The error number ErrorMisc (1) is a generic non specific error and ErrorWarning (2) is for warnings. The string gives a more user friendly textual representation of the error. Most of the BDS API functions return an error object to indicate the status of the function request. The BError::num() function will return the error number and the BError::str() function will return a C (const char*) format string pointer from the BError object.

The BString object stores a simple ASCII variable length string. It has the BString::str() function to return a C (char*) pointer to this string.

When the BDS stores an item in the MySQL database it allocates a unique ID for it within the database table. This is used to reference this particular item when updating its contents or deleting it. All Metadata objects have this "id" field.

Here will will provide a generic description of using the Bds::AdminAccess interface vai the C++ language. Using Python is more or less identical apart form the syntactical differences as described in the section on Python in this document.

6.1. Building 'C++' programs

In order to use the BDS API library you should have the bds-devel and beam-lib RPM packages installed. These will pull in the necessary dependancies. When compiling 'C++' code you should use the following include path flags:

```
-I /usr/bds/include/Bds -I/usr/include/Beam
```

When linking C++ code you should use the following flags in addition ot other libraries you are using:

```
-L /usr/bds/lib64 -lBdsData -lBds –lmseed-beam -lBeam
```

6.2. Building Python programs

Python programs should import either the "bdslib" or "bdslibe" modules depending on if exception error returns are wanted.

```
import bdslib
or
from bdslib import *
```

Note as of BDS 2.2.x only Python3 is supported.

6.3. Access to MetaData

In general the BDS API provides the following generic functions for Metadata access. We have listed these below for the Channel type but all other types are similar with appropriate name changes:

BEAM

- **channelGetList(Selection sel, BList<Channel>& channels):** This gets a list of the Channels based on the selection criteria typically by reading them from the MySQL database.

- **channelUpdate(Bool append, Channel channel, UInt32& id):** Either appends or updates an existing Channel entry based on the append boolean value. The MySQL database item ID is returned.

- **channelDelete(UInt32 id):** Deletes an existing channel object given its ID.

Please see the BdsExamples for the details of the programs. Here, for brevity and simplicity, we list the important aspects but ignore the necessary #include's and name space declarations etc. The following gets a list of stations and prints them out.

Create a Bds::Admin access object that provides the functional interface to the BdsServer:

```
AdminAccess        bds;
```
Now connect this to the appropriate BdsServer across the network:

```
if(err = bds.connectService("//bdsServer/bdsAdminAccess")){
        cerr << "Error: " << err.str() << "\n";
        return 1;
}
```
The connectService() function is passes a string providing a URL like parameter that defines the host name (or IP address) and the name of the BOAP services to connect to (bdsAdminAccess in this case). This will connect to the BOAP name server to determine the host and socket to connect to and create a network socket based connection to this service.

Now you need to provide a username and password to access the functionality:

```
err = bds.connect("test", "beam00");
```
The BdsServer maintains a list of users and passwords that are allowed to access it.

The function of the "bds" object can now be called which result in RPC requests to the BdsServer. For example to get a list of stations matching a selection criteria:

```
BError            err;
Selection         selection;
BIter             i;
BUInt             n;
BList<Station>    stations;

// Set up selection
selection.startTime.setString("2008-01-03T00:00:00.000000");
selection.endTime.setString("2008-01-03T00:01:00.000000");
selection.channels.append(SelectionChannel("BN", "EKA", "", ""));

// Get list of stations available
if(err = bds.stationGetList(selection, stations)){
        return err.set(1, "Error: Getting stations: ");
}

// This displays some of the information available
for(stations.start(i); !stations.isEnd(i); stations.next(i)){
        Station&   s = stations[i];

        printf("Station: %s\n", s.name.str());
```

BEAM

```
                printf("Type: %s\n", s.type.str());
                printf("Description: %s\n", s.description.str());
        }
```

A Python equivalent of the above is:

```
        # Create DataAccess object to connect to BDS Server
        bds = DataAccess();

        # Connect to the DataAccess service
        err = bds.connectService("//bdsServer//bdsAdminAccess");
        if(err):
                print("Error: " + str(err));
                return 1;

        # Connect to service
        err = bds.connect("test", "beam00");
        if(err):
                print("Error: " + str(err));
                return 1;

        selection = Selection();

        # Set up selection
        selection.startTime.setString("2008-01-01T00:00:00.000000");
        selection.endTime.setString("2008-01-01T00:01:00.000000");
        selection.channels.append(SelectionChannel("TT", "TSA", "", ""));

        # Get list of stations available
        (err, stations) = bds.stationGetList(selection);
        if(err):
                return err.set(1, "Error: Getting stations: " + err.str());

        # This displays some of the information available
        for s in stations:
                print("Station: " + s.name);
                print("Type: " + s.type);
                print("Description: " + s.description);
```

6.3.1. Python Example To Read an IMS Response File

The BDS data converters are a part of the BDS API so that client programs can use these locally. An example of reading a set of instrument responses from an IMS response file is in the BDS example fbdsDataFile2.py the contents of which follows:

```
#!/usr/bin/python
###############################################################################
#       BdsDataFile2.py           BDS API example code for a BDS DataFile access
#                           T.Barnaby,  BEAM Ltd,   2012-12-19
###############################################################################
#
# This Example code opens a Response file, gets information from the file.
#
```

BEAM

```python
import sys
import getopt
import cmath
from bdslib import *

def main():
    filename = "response.resp";

    # Create the BDS data contertor object for the data file type.
    f = DataFileResponse();

    # Open the data file
    err = f.open(filename, "r");
    if(err):
        print("Unable to open data file:", filename);
        return err;

    # Set the subformat in the file. Actually the DataFileResponse converter
    #  will performan an automatic type of file check if this is not given
    f.setFormat("RESPONSE");

    # Get the Metadata from the file
    (err, channelInfos) = f.getMetaData();
    if(err):
        print("Failed to extract Ps and Zs from file ", filename, str(err));
        return err;

    print("Number of channels of info:", len(channelInfos.channels));
    for chan in channelInfos.channels:
        print("Number of segments of info:", len(chan));
        for seg in chan:
            print("Number of Responses:", len(seg.responses));
            for resp in seg.responses:
                print("Resp: Stage: %d Type: %s" %(resp.stage, resp.type));
                # Print out the PoleZero response if of this type. Could
                #  also be a fap or fir response
                if(resp.type == "PoleZero"):
                    for c in resp.poleZeros.poles:
                        print("Pole: %f %f" % (c.real, c.imag));
                    for c in resp.poleZeros.zeros:
                        print("Zero: %f %f" % (c.real, c.imag));


    return 0;

if __name__ == "__main__":
    main();
```

This program uses the DataFileResponse data converter to read a traditional IMS ASCII response file and return the BDS objects that hold the information. The resp object is of the Response class and is the BDS method of defining a single stage of an instruments frequency response. The Response object can contain the frequency response in PoleZero, FAP or FIR formats and has additional Metadata available. See the reference manual for more details.

BEAM

The BDS stores all instrument responses as Response objects within its MySQL database for ease of access and return s a list of these when information on a data set is asked for.

6.4. Access to Sensor Data

The BDS API provides a simple set of functions that allow access to the sensor data within the BDS system. The API allows the user to select a set of data channels and then stream them over either using the raw BDS data block API or formatted in one of the seismic data formats that the BDS system has data converters for.

The BDS raw data API returns DataBlock objects that have the sensor data in arrays of double (64bit) floating point values. It allows simple and efficient access for direct data processing needs.

The BDS Data Access API has the following core functions:

Function	Description
BError getSelectionInfo(SelectionGroup group, SelectionInfo selectionInfo)	Returns information on all the Networks, Stations, Channels and Sources that the BDS system knows about. Useful for GUI driven data selectors.
BError getSelections(SelectionGroup group,Selection selectionIn, Selection& selectionOut);	Expands the given selection to match the data contents of the BDS system
BError dataSearch(Selection selection,DataInfo& dataInfo);	Searches for data matching the given selection and returns information on the associated data channels in a DataInfo object.
BError dataGetChannelInfo(DataInfo dataInfo, ChannelInfos* channelInfos);	Returns the channel MetaData in structured form for the set of channels defined by the DataInfo object.
BError dataOpen(DataInfo dataInfo, String mode, String format, UInt32 flags, DataHandle& dataHandle);	Opens a data stream. This will open a data stream which will contain all of the channels listed in dataInfo. The BdsServer will open all of the data files that contain data for the channels asked for. The mode should be set to "w" for writing data, "a" when appending data and "r" for reading data. The format should be one of the formats that the BDS has converters for or the raw API format (API, BDS, BKNAS, IMS1.0, SEED etc.) API defines the raw BDS API access scheme. The flags argument gives options such as return fullblocks etc
void dataClose(DataHandle dataHandle)	Closes the stream
API Stream Write Interface	
BError dataSetInfo(DataHandle dataHandle, DataInfo& dataInfo)	Provides information on the data. This is used to create file headers etc.

BEAM

BError dataPutBlock(DataHandle dataHandle,DataBlock& data)	Writes a data block to the file. These have to be sequential.
API Stream Read Interface	
BError dataGetInfo(DataHandle dataHandle, UInt32 infoExtra, DataInfo& dataInfo)	Gets information on the data from the files data header and perhaps the data blocks. This will include the real sample rate and actual number of samples.
BError dataGetWarnings(DataHandle dataHandle, BStringList& warnings);	Return a list or warnings.
BError dataSeekBlock(DataHandle dataHandle, BUInt32 channel, UInt32 segment, BTimeStamp time, BUInt32& blockNumber)	Seeks to a particular data block given a time. This can operate on a single channel if a channel number is given or on multiple channels if channel number is 0. The segment parameter is the data segment number.
BError dataGetBlock(DataHandle dataHandle, BUInt32 channel, UInt32 segment, BUInt32 blockNumber, DataBlock& data);	Reads a data block from the file. If the channel number is given, then it reads data from a given channel. If the channel number is 0 it will read a set of data from all of the channels in sample multiplexed form if this is possible. The segment parameter is the data segment number.
Formatted Stream Read Interface	
BError dataFormatedRead(DataHandle dataHandle,UInt32 number, Array<UInt8>& data)	Returns the next set of bytes of pre-formatted data. For example BKNAS, IMS etc.
BError dataFormattedGetLength(DataHandle dataHandle,UInt64& length);	The total length in bytes of the formated data.

An example of a Python programs read of data (ignoring the connection to the BDS API code) is:

```
# Set up selection
selection = Selection();
selection.startTime.setString("2008-01-01T00:00:00.000000");
selection.endTime.setString("2008-01-01T00:01:00.000000");
selection.channels.append(SelectionChannel("TT", "TSB01", "", ""));

# Get list of all data available for the selection
(err, dataInfo) = bds.dataSearch(selection);
if(err):
      return err.set(1, Error: Searching for data: " + err.str());

# We should now choose which set of data we would like from the list.
# Here we choose all of the channels data
s = dataInfo.channels.size();
if(s == 0):
      return err.set(1, "No data found");
```

BEAM

```python
    # bdsDumpDataInfo(dataInfo);

    # Get all of the Metadata associated with this data set
    (err, channelInfos) = bds.dataGetChannelInfo(dataInfo);

    # Open the file to read the actual sensor data
    (err, dataHandle) = bds.dataOpen(dataInfo, "r", "API", 0);
    if(err):
        return err;

    blockNumber = 0;
    while(1):
        (err, data) = bds.dataGetBlock(dataHandle, 0, 1, blockNumber);
        if(err):
            return err;

        # print("DataChannels:", data.channelData.size());

        print("Data0:", data.channelData[0][0]);
        blockNumber += 1;
```

An example of a Python programs read of formatted data (ignoring the connection to the BDS API code) is:

```python
    # Set up selection
    selection  = Selection();
    selection.startTime.setString("2008-01-01T00:00:00.000000");
    selection.endTime.setString("2008-01-01T00:01:00.000000");
    selection.channels.append(SelectionChannel("TT", "TSB01", "", ""));

    # Get list of all data available for the selection
    (err, dataInfo) = bds.dataSearch(selection);
    if(err):
        return err.set(1, "Error: Searching for data: " + err.str());

    # We should now choose which set of data we would like from the list.
    # Here we choose all of the data.
    s = dataInfo.channels.size();
    if(s == 0):
        return err.set(1, "No data found");

    # A debug printout of the DataInfo object
    # bdsDumpDataInfo(dataInfo);

    # Open the data file for reading in IMS format
    (err, dataHandle) = bds.dataOpen(dataInfo, "r", "IMS", 0);
    if(err):
        return err;

    # Read the formatted data
    while(1):
        (err, data) = bds.dataFormattedRead(dataHandle, 1024);
        if(err):
            return err;
```

BEAM

```
        if(data.number() == 0):
                break;

        print "Data0:", data[0];
        # file.write(data);
```

When accessing sensor data it can be returned in either sample multiplexed or channel multiplexed formats. Sample multiplexed is where there are a set of samples per time instant across all of the channels. This requires the original data to have been in sample multiplexed format or marked as synchronously sampled data. Channel multiplexed is where each channels data is separate and may or may not be synchronously sampled with other channels.

6.4.1. BDS DataInfo

When sensor data is selected, the dataSearch() function returns a DataInfo object to describe the selected sensor data. A selection can result in many sensor data files being accessed, possibly one for each channel and multiple files per channel. A DataInfo object has the following fields:

Field	Description
startTime	The start time for the data
endTime	The end time for the data (just past data)
array	If this is a set of data channels from a seismic array Station, this gives the array stations name.
description	A generic description of the dataset. Comes from Metadata in the primary sensor data file if present.
synchronous	A flag indicating that the set of channels are synchronously sampled.
channels	A two dimensional array of DataChannel channel information. The first dimension has one entry per channel. Then for each channel there can be one or more entries. There would be multiple entries if there are multiple segments of data. See below for more details.
info	Provides extra Metadata on the files from database Metadata.
infoExtra	Extra Info on the set of channels from in-depth sensor data file Metadata. Used for extended error/logging information and is specific to the original data file/stream imported.
warnings	A list of warnings for the data either system generated or user added.

BEAM

The channels information consists of an array of entries per selected sensor data channel. There is one DataChannel entry per data segment. Segments are introduced at each time gap or file change. The DataChannel entry has the following fields:

Field	Description
id	Unique ID when stored in a database or for other uses.
startTime	The start time for the data
endTime	The end time for the data (just past data)
network	The Network Name
station	The Station name
channel	The Channels name
source	The Data Source
numBlocks	The total number of blocks per channel if known, 0 otherwise
numSamples	The total number of samples per channel if known, 0 otherwise
sampleRate	The data's sample rate
sampleFormat	The data sample format
dataFileId	The Data File Id. This links to the particular DataFileInfo where the data is stored.
dataFileChannel	The Data File Channel number. The channel number within the data file. (1, 2, 3 ...)
importFormat	The original data format
importFilename	The original data file name
importStartTime	The original import files start time
info	Extra info on the channel

When an initial dataSearch() is performed the DataInfo object is primarily filled in with information form the MySQL databases DataChannel table information. The channels will be segmented at file boundaries if multiple files are used to service the requested data's time span.

When later the actual seismic data set is opened using the dataOpen() command, a following dataGetInfo() command will return a more highly detailed DataInfo object based on the actual seismic data files's contents. In this case there may be more segments of DataChannel entries due to time gaps between blocks of data. The numBlocks and numSamples fields will be updated to match the actual files contents and the sampleRate field will be calculated based on the actual DataBlock time stamps and actual number of samples. The info field will also have extended information from the data files internal MetaData.

6.4.2. BDS DataBlock

When reading/writing data from/to a BDS stream with the raw API format, there is a single object called a

BEAM

DataBlock that stores the data. This object has the following attributes:

Member	Description
startTime	The Start time
endTime	The End Time
channelNumber	The channel number. 0 if all channels
segmentNumber	The segment number. 0 if all segments
channelData	Two dimensional array of data in floating point format. Firs dimension is each channel. This may be just one channel or multiple channels depending on how you opened the data stream. Each channel has an array of data values and each channel can have a different number of samples to others.
info	Extra meta data information. This may be available with particular data formats such as the TapeDigitiser

There is an array of data samples per channel if the data is in sample multiplexed format. If in the simpler channel multiplexed format there is just one channels data in an array. The channelData array contains samples in double (64 bit) floating point format.

The info field contains a dictionary of extra Metadata information for the block specific to the original data format imported. This may be tape quality information in the case of TapeDigitiser data for example.

7. BDS Code Examples

There are a number of BDS C++ and Python API examples in the /usr/bds/bdsExamples directory.

8. Notes on Specific BDS API Aspects

8.1. BDS Sensor data storage and access

The original Seismic sensor data comes from many different sources and in many different formats. To simplify the storage of data the BDS uses its own internal BDS data format that is used to store the Sensor data. This is a generalised format that is able to store data from the various different formats without loss of information including some Metadata. It is described in more detail in the BdsDataFile.pdf document.

The sensor data is stores with one of two primary structures: Sample multiplexed and Channel multiplexed. Sample multiplexed is where there are a set of samples per time instant across all of the channels. This requires the original data to have been in sample multiplexed format or marked as synchronously sampled data. Channel multiplexed is where each channels data is separate and may or may not be synchronously sampled with other channels.

Data is blocked into time period blocks of data with a variable number of samples per block. The original data quite often has issues such as missing blocks or periods of data and data corruptions. A BDS data file can store one or more channels of data as a set of these blocks. The blocks can be stored in time sequence order or more randomly. Data from real-time data sources are generally in time order except when backfill operations are carried out.

When the BdsServer opens a BDS data file it orders the blocks in a time sequenced list of blocks in the

BEAM

file. When reading the data the data will be segmented into time periods of time consecutive data blocks based on the start and end times of each block. The data will also always be segmented across file boundaries. The API provides the DataFlagMergeSegments option to dataOpen() which asks the BdsServer to merge all data segments, even across file boundaries, assuming their start/end times match.

Now some sources of data such as hydro-acoustic, often have clocks that are synchronised with GPS time infrequently. This can result in small time discontinuities between blocks when the clock jumps. The BDS has a TIMESTAMP_JITTER facility that allows the timestamps of consecutive blocks to be slightly miss-aligned (normally by 1% of a blocks time period). When exporting data the BdsServer calculates an overall sampleRate based on the start and end times of the data requested and the number of samples of data. For data sources with variable clock sources, the sample rate returned can thus vary and be different to the generalised sample rate in the Metadata for the channel.

When using the BDS dataOpen() function the format can be set to any of the formats that the BDS's data converters support. The API function: dataFormatGetList() returns a list of the formats supported and if they are available for creating and/or reading data in the given format.

The dataOpen() function's options parameter is a bit list of options that can include the following or'ed together:

Name	Description
DataFlagClipDataToTime	Clip the data to the time period requested so that data begins and ends with the sample at the requested time. Normally the BDS will return data beginning at the startTime of the data block in which the user startTime occurred and the endTime of the block that the user supplied endTime occurs so that complete original data blocks are returned.
DataFlagClipDataToChannels	When requesting data from a number of channels the start and end times per channel may be different due to missing blocks or other reasons. This option asks the BDS to truncate the data so that all channels start and end with the sample timed sample.
DataFlagMergeSegments	Data will normally be segmented at file boundaries. This option merges these segments assuming the start/end times match.
DataFlagNoMetadata	Don't include non-essential Metadata in export data files. Only Metadata needed to get the actual samples out such as the channel names is provided. There is no sampling rate, response or calibration information provided unless it is required in the data format in question. When export a SEEd format file this would be equivalent or what is often called a mini-seed file.

When reading data using the dataGetBlock() function there are channel and segment parameters. The use of these depend on whether the data stream has been opened in sample multiplexed or channel multiplexed mode. In sample multiplexed mode the channel parameters should be set to 0 which means all channels as all channels data will be returned in each data block. The segment parameter can now be set to either 0, to return all of the data blocks across all segments, or to the appropriate segment number required.

I channel multiplexed mode the channel parameter would be set to the channel number requested (1, 2,

BEAM

…) and the segment parameter can be either 0, to return all of the data blocks across all segments, or to the appropriate segment number required.

8.2. BDS Sensor data access with MetaData

Once you have opened a BDS data stream, which may have opened several data files, you can use the **dataGetChannelInfo()** function to return all of the Metadata associated with the Sensor data.

This returns a ChannelIinfos object which is essentially a two-dimensional array of ChannelInfo objects. The first dimension is for the channels. For each channel there is then one, or (rarely) more than one, ChannelInfo object containing a Metadata segment over a period of time for the channel. Usually there is a single ChannelInfo object that contains the Metadata for the channel for the entire timespan of the requested data, but if the Metadata changes (eg. there is a change of seismometer orientation) during the requested timespan then further ChannelInfo objects, with appropriate time spans matching the Metadata changes, are present.

Each ChannelInfo object has the following core fields:

Field	Description
startTime	The start time
endTime	The end time
station	The seismic Station
location	The Station's location
channel	The seismic data channel
source	The source of the data
dataType	The type of data including: seismic, seismicUnknown, data, log, unknown, empty.
digitiser	The digitiser that was used
sensor	The Sensor that was used
calibration	Calibration information for the channel
responses	The list of frequency responses for the channels data processing chain.

8.3. BDS Data Converters

The BDS has a library of data format converters to handle the various seismic file and stream data formats. These data converters, supplied in the bdsDataLib library are a set of classes based on the generic DataFile class which defines the generic API for all converters. The converters are used within the BdsServer to return data in a user requested format or in client programs directly when importing or exporting data.

There is an overall data converter management class, DataFormats, that can provide a list of all of the data converters available and the formats they can handle. A converter bay handle reading or writing a file or both of these.

BEAM

Each data converter has the following standardised API functions available:

Function	Description
BError open(BString fileName, BString mode)	Opens the file for read or write access
BError close()	Closes the file
BError setFormat(BString format)	Set the sub-format to use if needed
DataOrder getDataOrder()	Get the expected multiplexing order of writing data, by sample or by channel
int getFeatures()	Get bitmask of supported features
File writing	
BError setInfo(...)	Set information on data for write
BError start(BUInt channel, BUInt segment)	Start writing next segment of data
BError writeData(const DataBlock& data)	Write a block of data
BError end();	End write segment
File Reading	
BError getInfo(...)	Get info on data
BError seekBlock(...)	Seek to a specific block based on a time
BError readData(...)	Read a data block
BError getMetaData(ChannelInfos& channelInfos)	Return all known MetaData in the file

New data converters can be added or the existing ones updated. The system will then have support for these data formats wherever the data converters are used.

See the separate BdsDataFormats.pdf document for more details on the data formats supported.

8.4. BDS Metadata updating

When updating Metadata often a lot of different associated Metata data needs to be updated at once. For example adding a new set of channels together with the instruments, calibrations and responses. The BDS API provides the transactionStart() and transactionEnd(Bool abort) functions to carry out a grouped set of Metadata updates. The database changes are only actually committed when transactionEnd() is called with the abort parameter set to 0. If the abort parameter is 1 then the changes are ignored. This is especially useful if one of the database changes results in an error.

It is also possible to use the databaseBackup() function to backup the entire database to a compressed file prior to carrying out extensive changes. Recovering the database can be tricky as other modifications, including new data imports from the real-time importers will likely need to be preserved. The

BEAM

databaseRestore() function provides the ability to restore certain parts from the database. As well as a reference for the backup it has a type field. This can be set to: "admin", "metadata", or "data". Normally a restore of all of the "metadata" would be used when recovering from Metadata change issues.

8.5. BDS MySQL Database Accessing

The BDS API provides the sqlQuery() API function to do low level MySQL database queries. You can also use standard MySQL external tools for this.

However, you are discouraged from using this mechanism or standard MySQL tools for updating the BDS database entries. The BdsServer provides validation of the data input and may modify more that one MySQL data table at a time and makes sure that the database is kept consistant. If using raw MySQL access you will need to make sure you set fields to the correct values and update all data tables to match as necessary. Also note that the BDS's functions for system backup snaphots and database backups cane be messed up if you perform a direct MySQL database change. Reading the database entries directly is no problem however.

8.6. Users and groups

Access to the BDS system is protected by a username/password system. The BDS API's connect() call provides this information. The BDS maintains a user list in its MySQL database that can be manipulated using the BDS API in a similar way to general Metadata access.

Each user can belong to one or more security groups. The security groups allows the user to perform certain functions and access certain data. The standard security groups include:

Group	Description
admin	Administrator. Can access all functionality
control	Used for control program access to control the operation of the BdsServer. This group cannot change data but can set the BdsServer into different modes for backup and other such purposes.
dataAdd	Can add data to the system. The import daemons would have a user account and belong to this group
dataDelete	The ability to delete sensor data and Metadata
dataModify	The ability to modify MetaDataq
userAdmin	The ability to changes user details and add/delete users.
userSet	The ability for a program to change its user to another. Used by daemons such as the BdsWeb system to change its user to somone logged into the web system so they can access particular data sets etc.
dataAccessAll	Ability to access all data
data,*	Particulat datasets can be allocated to a security group that prevents normal users accessing it. If a user belongs to that group they will be able to access it.

BEAM

8.7. Data Availability

The BDS API provides the dataAvailability() function that can be used to obtain information on the availability of data in a number of channels. The dataAvailability() function accepts the normal Selection object to select the channels to check the availability on. It returns an array of DataAvailChan objects, one for each channel that lists the sensor data segments and their individual availability.

Note that this system uses the MySQL DataChannels information only, it does not open the seismic data files to get detailed information. Thus it doesn't know about missing blocks within the data and for real-time data it assumes the entire days data is present (even though a portion of it will be present).

8.8. Changes, Notes and Logs

Whenever a change is made to the database tables, the system will create a Change record in the Changes database table. This provides information on the table modified and the record id modified. Change records normally are made within a ChangeGroup which groups together a set of changes. The ChangeGroup also lists the user than made the changes.

Notes can be added by any user and BDS programs. It provides a network:channel:source with time period indexed note that can be added to the system. It is also possible to add a note with a linked file uploaded, such as a PDF document. This can be used to provide additional information on a set of data.

The Log's system allows general system log entries to be added by programs or users for information and system status messages.

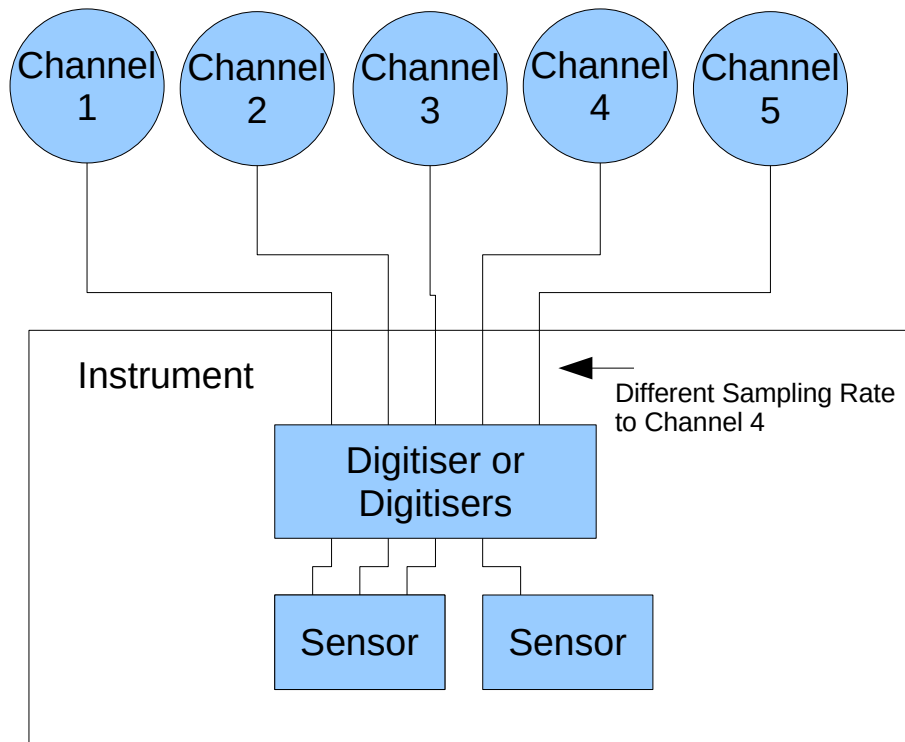## 8.9. Instruments: Sensors and Digitisers

The main Metadata information for a Channel is provided by the Station, Location, Calibration and Response objects. However the system is also able to store detailed information on the channels instrument measuring system split into a Sensor object and a Digitiser object.

The system has a database table of ChannelInstrument objects that are connected with an individual Channel by storing the Channel's id. The ChannelInstrument objects have a start and end time to handle instrument changes as well as a source if there are multiple instruments on the same channel. The ChannelInstrument object has an ID for the Sensor object and Digitiser object for this channel.

There are two methods of using this system:

1. Have generic Sensor and Digitiser objects defining generic instrument types and share these amongst multiple channels. Note this could be for a three component sensor for example.

2. Have an individual Sensor and Digitiser object per channel. This allows individual instrument information such as serial numbers to be stored.

It depends on how much information there is on the individual Channels as to which of these methods are chosen for a particular stations channels.

BEAM

General Channels/Instruments

A typical Metadata procedure would be adding a new Channel to the system. In order to do this you would carry out the following actions using the BDS API:

1. Use or create a new Station if needed with its physical location defined.

2. Create a Channel for the given station with an appropriate name ("<ChanType>_<ChanAux>") and over an appropriate timespan (could be over all of time from 0001-01-01 → 9999-01-01).

3. Add a suitable Calibration object with appropriate Network:Station:Channel:Source name over the time period required. Its fields would be filled in with the necessary data.

4. Add suitable Response objects, one for each stage of processing in the instruments measurement chain. Stage1 is typically an overall instrument chains response which may not be as accurate as applying all of the various stages responses.

5. Create, or find the ID of an existing, Sensor. Creating a new Sensor with sensorUpdate() will return the new ID of this sensor. Its fields can be set as needed.

6. Create, or find the ID of an existing, Digitiser. Creating a new Digitise with digitiserUpdate() will return the new ID of this digitiser. Its fields can be set as needed.

7. Create the ChannelInstrument using the Channel, Sensor and Digitiser object ID's returned by these three processes and over the time period required. There can be multiple ChannelInstrument objects, each over a different time spans, to handle instrument changes. These time spans should

BEAM

not overlap.

The BDS API's <item>GetList(…) function returns a set of objects matching a selection criteria. The methods sensorGetList() and digitiserGetList() can be used to find existing sensors and digitisers if they are already assigned to channels. The channel ID must first be found by feeding network channel:source channel identification names and a timespan to a channelGetList() method (unless the ID is known because the channel was created earlier in the program).

The BDS API's <item>Update(…) function is used to modify existing objects or create new ones and returns the ID of the original or new object.

9. General API Notes

9.1. Updating from Python2 to Python3

Older BDS systems supported the Python 2 language. BDS release 2.2.x onwards only supports Python3. The majority of Python2 code should run with Python3 as long as the function style of the print() function was/is used.

- To update existing Python2 code the systems "**2to3 -w"** program can be used.