BEAM

Beamlib API Manual

Project	Beamlib
Date	2022-12-01
Version	3.0.0
Reference	BeamlibApiManual
Author	Dr Terry Barnaby

Table of Contents

1. Introduction

The Beam Beamlib class library provides system portable low level classes for the manipulation of strings, lists arrays as well as system independent network sockets etc. It is documented in: BeamlibApi reference documentation.

It is based on Beam's internal Beam-lib library that was initially started in the late 1980's to add Smalltalk like constructions/components to the emerging C++ language for use within Beam for real-time data processing and embedded system uses. Over the years it has been extended as needed to support wildly differing projects.

This new Beamlib library is based on the Beam-lib library but has been reduced to the core generally used classes, with many proprietary classes and functions removed. Also the documentation system has been updated together with more documentation on the library. The first iteration of the Beamlib library matches the API of the earlier library to simplify porting existing code. However we intend to clean up the API for a future version as there is a lot of functions in the library that can be removed and the API could do with a tidy with modern C++ compiler abilities and a cleaner API.

The Beamlib library has been ported to various Linux based targets as well as Microsoft Windows. There is also a variant of it called Armsys that is available for Microcontrollers.

Most 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. Positive error numbers indicate an API level error has occurred. Negative error numbers are from the underlying Linux system. Seet the section on errors for more detail.

The BString class is used extensively to efficiently 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 Beamlib library provides integer and floating point types such as BInt32, BUInt16, BFloat64 that are based on the appropriate lower level ‘C’ type. These have a fixed memory size to allow easy passing over communication interfaces and uses the CamelCase naming scheme so that basic type names can match this naming scheme.

The core Beamlib C++ classes are:

Class	***Description***
BString	A variable length string class used for storing, passing and manipulating ASCII text strings. This uses shared and referenced malloced data strings for efficiency.
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.

2. Installation

The Beamlib class library can be installed from the following RPM packages:

- beamlib-lib: Runtime shared libraries.
- beamlib-utils: Runtime tools.
- beamlib-devel: Development include files and shared and static libraries.
- beamlib-doc: Documentation.

If the BOAP name server is needed for applications that use the BOAP RPC system, then the Boap name server can be started using the command: “systemctl start beamlib-boapns” and set to start at boot with the command “systemctl enable beamlib-boapns”.

3. Naming Convention and Style

Beamlib uses a CamelCase naming scheme with the first character in uppercase when types/classes or constants are named and lower case for variables and function names. All of the type/class definitions are prefixed with the upper case letter “B”.

The Beamlib library uses only the basic C+_+ language features and although it supports operator overloading for some classes, where it obviously makes sense, most functionality is provided by class member funcions.

4. Simple Example

A simple example of using the library is below showing the Beamlib style:

```
#include <BError.h>

BError func1(BString name, BString& nameRet){
	BError	err;

	if(name[0] == ‘A’)
		err.set(ErrorMisc, “Cannot handle strings starting with A”);
	else
		nameRet = name.subString(1, -1);

	return err;
}

int main(){
	BError	err;
	BString	name = “Hi there”;

	if(err = func1(name))
		printf(“Error: %d, %s\n”, err.num(), err.str());

	return 0;
}
```

5. Fundamental Types

Beamlib defines some fundamental types that can be used in programs. In C== standard types such as an “int” can store a different number of bits depending on the architecture and compilers used. As Beamlib targets embedded systems as will as larger desktop systems having known size variables is more reliable. The Types also use a CamelCase naming scheme with the first character in uppercase when types/classes or constants are named.

The fundamental types are defined in BTypes.h and include:

BInt8	8 bit signed integer
BInt16	16 bit signed integer
BInt32	32 bit signed integer
BInt64	64 bit signed integer
BUInt8	8 bit unsigned integer
BUInt16	16 bit unsigned integer
BUInt32	32 bit unsigned integer

BUInt64	64 bit unsigned integer
BFloat32	32 bit floating point
BFloat64	64 bit floating point

5.1. BString

The BString class is one of the core Beamlib types that is used extensively in the library. The BString class is designed for the storage and manipulation of variable length ASCII strings. It uses the BRefData class to store the null terminated strings in shared and references counted shared memory areas on the heap. This efficient method reduces memory allocation, deallocation and copying. Copying or just passing a string in and out of a function becomes a simple pointer copy with the string data reference count incremented.

A BString is designed for simple ASCII text but can store UTF8 encoded characters as well with restrictions. The main restriction is that the character positional access with the get() and [] operator functions just return the 8 bit byte that may be part of a multi-byte UTF8 “character”.

You can convert a BStirng to a const char* using the BString::str() method. You can create a BString from a char* with its constructor BString(const char*) which is used implicitly for conversions. The BString also supports conversion to and from Qt QStrings if the Qt headers are included before the BString.h header. The BString also supports input and output from std::iostream objects.

As well as its constructor function, BString supports a number of explicity convert() functions that can convert basic types such as integers to a BString. There are also some ret*() functions to parse the BString for integer and floating point values.

The operators: =, +, +=, <, >, >=, <=, ==, != operate as you would expect with the +" appending strings.

There are a number of string manipulation functions that are provided.

There are also some overloaded conversion functions with the names: toBString() and fromBString() for use in conversion of data types to and from strings.

5.2. BError

Most 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 which makes it easy for software “if” statements to use. Positive error numbers indicate an API level error has occurred. Negative error numbers are from the underlying Linux system. There is a generic error ErrorMisc = 1 which is used for a generic error and ErrorWarning = 2 which is used for something that is often just a warning rather than an error. In all cases the ASCII string, that is part of the BError object, describes the error in more detail.

There are 4 sets of error number ranges:

- 0 – 63: Standard Beamlib error numbers.
- 64 – 127: User program errors
- < 0 Negative: Underlying OS errors

If you need to generate an error you can add your own error numbers starting at ErrorAppBase (64) or simply use the error number ErrorMisc (1) with an appropriate string describing the error.

The Beamlib library does not use C++ exceptions, preferring to always return errors up the function chain to be handled near the point of the error in the code. However if desired the BError can be raised as an

exception from user code.

Standard Beamlib API level error numbers are defined in the BError.h file and are as follows:

ErrorOk	0	The function ran fine.
ErrorMisc	1	A generic error occurred. Look at the string for more information
ErrorWarning	2	A generic warning occurred. Look at the string for more information
ErrorParam	3	A function parameter was incorrect, possibly out of range
ErrorTimeout	4	A time-out occurred
ErrorNotAvailable	5	A resource is not available
ErrorData	6	Some data related error like a data format or corruption error
ErrorChecksum	7	A checksum error
ErrorOverrun	8	A data buffer/queue has overrun, perhaps the system can't keep up with the data rate
ErrorUnderrun	9	A data buffer/queue has underrun, perhaps the data source is not sending data fast enough
ErrorInit	10	Initialisation error
ErrorConfig	11	Configuration error
ErrorNotImplemented	12	The function has not been implemented
ErrorResourceLimit	13	A resource limit has been reached
ErrorEndOfFile	14	The end of a file has been reached
ErrorFile	15	A file read or write error
ErrorFormat	16	A data format error
ErrorComms	17	A generic communications error
ErrorAccessDenied	18	Permissions for access deny this operation
ErrorNoData	19	There is no data available
ErrorEndOfData	20	The end of data has been reached in a file or stream
ErrorDataPresent	21	Data is already present when trying to add new data
ErrorDataTruncated	22	Data is truncated, ie. not of the expected length/size
ErrorApiVersion	23	The API version does not match

For the negative system errors, refer to the linux standard error documentation. The error numbers will match the standard errno.h values but as negative numbers.

5.3. BList

The BList class is a simple doubly linked list of objects. It is used to store an ordered list of any type/class of objects. The class provides a simple iteration system, using a BIter class, to allow easy navigation through the list. The BIter is really just a pointer to one of the lists entries and holds state on the current location. You can set a BIter's position using the start(BIter&), end(BIter&) and similar functions. You can access objects stored in the list with the get(BIter) function or the operator[BIter] operator. The BIter

can point off the beginning or end of the list. In this case the isEnd(BIter) function will return true.

The list entry access functions return a reference to the chosen item. If the list is a non const list this allows the item to be modified if needed.

The list supports stack functions push() and pop() and queueing functions such as queueAdd() and queueGet().

There is a macro BListLoop(list, iterator) that provides a concise way to iterate over a lists objects.

A example of its use is:

```
typedef BList<BUInt> UIntList;

UIntList funca(UIntList& list1){
      UIntList    list2;      // A list of unsigned integers
      BIter       i1;         // Iterator for the list

      printf("There are %u items in the list\n", list1.number());

      // Walk over the list
      for(list1.start(i1); !list1.isEnd(i1); list1.next(i)){
            list2.append(list1[i1] * list1[i1]);
      }

      // Alternative Walk over the list
      BListLoop(list1, i2){
            list2.append(list1[i2] * list1[i2]);
      }

      return list2;
}
```

It is also possible to use an integer as an iterator like:

```
// Walk over the list
for(int i = 0; i < list1.number(); i++){
      list2.append(list1[i] * list1[i]);
}
```

However this is inefficient as the BList will need to walk through the list to find entry. A BArray is more efficient if an integer iterator is needed.

The BList has a sort() function. By default this will use the stored objects operator>() function to compare the lists stored objects. However it is also possible to provide the BList with a function to be called to compare the stored objects.

```
int sort1(Person& p1, Person& p2){
      return p1.name > p2.name;
}
list1.sort(sort1);
```

5.4. BArray

The BArray class is a simple, contiguous in memory, list of objects. It is used to store an ordered list of any type/class of objects and you can use a simple integer as an iterator. The current implementation is based on the Standard C++ library vector class and has all of the functionality of that class.

```
typedef BArray<BUInt> UIntArray;
```

```
// Walk over the array
UIntArray   array1;

for(int i = 0; i < array1.number(); i++){
      array2.append(array1[i] * array1[i]);
}
```

5.5. BDict

This is a dictionary list class that uses a string as an index. For efficiency it uses a hashed string system to speed up access. It is derived from a BLIst xclass and so has all of the functionality of that class as well as the ability to access elements with a string key. An example is:

```
typedef BDict<BString>  StringDict;
StringDict  dict;
dict[“Item1”] = “This is item1”;
dict[“Item2”] = “This is item2”;

printf(“Items2 is: %s\n”, dict[“Item2”].str());
```

5.6. BDictMap

A mapped Dictionary class. This is based on the Standard C++ library map class and has all of the functionality of that class.

5.7. BComplex

The BComplex class along with the BComplex32 and BComplex64 classes are used to implement complex numbers. They are based on the Standard C++ library complex class and has all of the functionality of that class.

6. Multi Threaded Functions

The Beamlib library provides a set of classes to assist with multi-threaded programming. These include:

- BMutex: Provides a data section lock against other threads.
- BSemaphore: Provides boolean and counting semaphores for inter-thread communication.
- BSema: Provides an alternative style boolean and counting semaphores for inter-thread communication.
- BRWLock: Provides a read/write thread lock
- BAtomicCount: Provides an atomic integer counter
- BAtomic: Provides variable sized atomic integer counters
- BQueue: Provides a thread save queue of objects that can be used to communicate between threads.
- BThread: Implements a thread of execution.
- BTask: Alternative thread of execution class
- BStringLocked: Provides a basic thread locked string.
- BCond: Provides a thread safe conditional variable
- BCondInit: Provides boolean and integer thread safe variables

- BEvent: Event handling with an event description object and an implementation using OS pipes and BQueues.
- BEvent1: An alternative more generic event handling set of classes

7. File Access

- BDir: File system directory class.
- BFile: File system file access class.
- BConfig: Simple file based Configuration storage class.

8. Communications

- BSocket: Implements a network socket communications class and TCP/IP V4 addressing schemes
- BPoll: Poll a set of file descriptors for events
- Boap: A set of classes that implement an object orientated RPC communications system. Normally used in conjunction with the bidl IDL API compiler.
- BoapMc: A simpler Boap communication system for small microprocessors
- BoapMc1: A new but sill simple Boap communication system for small microprocessors
- BoapSimple: Old Boap communications class. Depreciated
- BComms: A base class for communications classes having a generic API.
- BEndian: A set of efficient inline functions to handle architecture endian conversion

9. Time

- BDate: The BDate class stores a calendar date wit a year and day in the year components. It provides functions to set this date from a string and convert the data to a string as well as BDate comparison functions.
- BDuration: Stores and manipulates a time to the nearest microsecond and a maximum of 24 hours
- BErrorTime: Error return class with time field. This provides an alternative to the standard BError return type for errors but includes BTimeStamp information.
- BTime: Implements a simple date/time class. Stores the date/time as a number of seconds since Unix epoch 1970-01-02T00:00:00.
- BTimeStamp: A date and time storage class with microsecond resolution.
- BTimeStampMs: A date and time storage class with milisecond resolution and an extra field to indicate a particular sampleNumber it refers to
- BTimeUs: Time storage as an unsigned 64bit value to TAI time standard.
- BTimer: Stopwatch style timer

10. Data

- BObj: A generic object base class that has runtime definable data fields
- BObjStringFormat: A set of functions to perform object to string and string to object for standard types and generic BObj classes.
- BFifo: A template first in first out data buffer to store any object types

- BFifoCirc: This class implements a thread safe FIFO buffer using a binary sized circular memory
- BFileCsv: A class to read and write CSV formatted files.
- BFileData: A class to implement a data storage file
- BMysql: A class to provide access to a MySQL database
- BNameValue: A simple, templated, name/value pair.
- BNameValueList: A simple, templated, name/value pair list.

11. System

- BFirmware: BEAM firmware file format structures
- BRtc: Access to the systems real time battery backed up time hardware
- BRtcThreaded: A thread safe class to access to the systems real time battery backed up time hardware
- BSpi: BSpi class for accessing SPI hardware devices
- BSys: Some Armsys system definitions to provide compatibility with Armsys

12. Debug

- BDebug: Software debug functions.

13. Misc

- BEntry: A simple name/value string pair
- BEntryList: A list of string name/value pairs
- BEntryFile: A file based list of string name/value pairs
- BBuffer: Create and manipulate a variable sized byte data buffer.
- BBufferStore: Create and manipulate a variable sized byte data buffer. Has functions to store and retrieve basic and extended types/classes in the binary buffer.
- BCrc16: A 16bit CRC generator
- BCrc32: A 32bit CRC generator
- BRefData: A pointer to a variable sized data area with reference counting so the data areas can be shared.
- BTable: A simple string based table structure
- BUrl: Access to a Url

14. BOAP Name Server

The Beamlib class library includes a BOAP object name server program, beamlib-boapns. This listens on the network socket named “boapns” which by default is set at 12000. BOAP server programs can advertise there RPC server objects via this name server and clients can find available RPC objects from it.

The beamlib-boapns will return the TCP/IP address and the socket number of the appropriate server program for the named object.

15. BIDL: Beam Interface Definition Language

Beamlib includes a interface definition language compiler named beamlib-bidl. This compiler accepts an interface definition file written using a C++ like syntax to define a RPC object API.

The bidl file allows classes to be defined from a set of standard Beamlib primitive types and provides a mechanism to define RPC server objects with member functions that can pass objects of these defined classes.

The beamlin-bidl compiler will generate the necessary C++ header and implementation files for both the server and clients to implement this RPC API.

As well as being able to generate a C++ implementation of the API, it can also generate implementations for other languages or provide helper code for these language implementations.

An example bidl file looks like:

```
module Api1 {
      apiVersion = 5;

      /// Priority levels
      enum Priority     { PriorityLow, PriorityNormal, PriorityHigh };

      class Person {
            String            name;
            UInt              age;
            List<String>      friends;
      };

      interface PeopleApi {
            Error       getPerson(in String name, out Person person);
            Error       addPerson(in Person person);
      };
};
```

16. Copyright and License