eProsima Dynamic Fast Buffers

Users Manual
Version 0.1

Experts in networking middleware

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 2 – 1ºG
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Technical Support

- Phone: +34 91 804 34 48
- Email: support@eProsima.com

Table of Contents

1 Introduction

eProsima Dynamic Fast Buffers is a high-performance library that allows users to describe and serialize or deserialize data dynamically in run-time. Its functionality is based on the creation of a typecode for data definition, and generate afterwards a bytecode to do data serialization. There is no need for users to know any internal data of the typecode or the bytecode, neither of the serialization procedure.

This library uses eProsima Fast Buffers for data serialization and deserialization, for in it are defined the functions that perform this operations.

1.1 Data Describing and Serializing

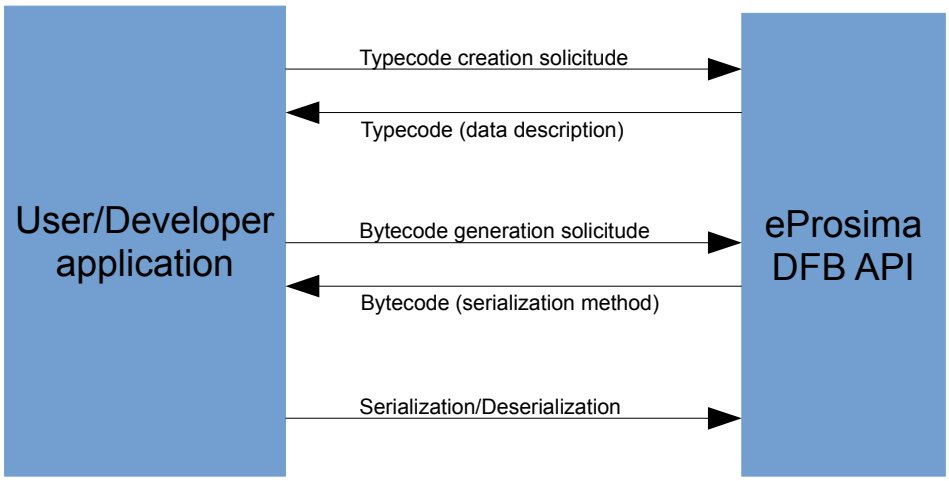
In computer science, in the context of data storage and transmission, serialization is the process of translating data structures or object state into a format that can be stored (for example in a file or a memory buffer) and recovered later in the same or another computer environment.

When it comes to dynamic serialization, the data that will be serialized has to be described somehow. It is in this moment when eProsima Dynamic Fast Buffers provides the functionality for doing it.

The description of data can be done by using a typecode. This is just an entity that stores information in order to proportion full knowledge of the data which is described through it, taking care only of the data type and not of its content or value. This avoids developers the task of defining their data types inside an IDL file that odd to be parsed later in run-time.

Once the data is described, the application will get full knowledge of it and how to manage its contents. At this moment, a bytecode can be generated in order to define how this data types must be serialized. This bytecode is an internal representation of how the data is going to be serialized.

The image below shows the library functionality:



1.2 Main Features

eProsima Dynamic Fast Buffers (DFB) provides an easy way to describe and serialize or deserialize data defined by library's users.

eProsima Dynamic Fast Buffers exposes this features:

- **Data description through a typecode:**
  - The typecode is a way to describe how is the data that any user wants to use in its application.
  - Through this typecode, eProsima DFB knows how is the user's data and how to move along it.
  - Avoids the developer to describe data statically inside an IDL file.
- **Bytecode generation for data serialization/deserialization:**
  - This is just a way for DFB of knowing how to serialize the data defined by user.
  - There are two kinds of bytecode that can be generated, one specific for data serialization, and the other for data deserialization.
- **Data serialization/deserialization**:
  - eProsima DBF provides users a way to serialize or deserialize data by using a FastCDR object provided by eProsima Fast Buffers library.
  - Serialized data will be stored inside a buffer defined by user.
  - Deserialized data will be stored in the user's data type previously defined.

2 Building an application

eProsima Dynamic Fast Buffers allows a developer to easily describe its own data types, providing functions to serialize them into a buffer previously defined and the deserialize them back.

How the library defines or works with the user's data types must be transparent, so there is no need for mentioned users to know any of that information. From the point of view of the developer, a Typecode object that represents the data could be created in his application and this object could be used for creating a Bytecode object.

This Bytecode is the object that will be used later by eProsima DBF for knowing how to serialize user's datum. In the same way, how eProsima DFB uses this Bytecode object should be transparent for the developer. Only the creation of the Typecode concerns to the developer.

eProsima DFB offers this transparency to the developer and facilitates the development. The general steps to build an application are:

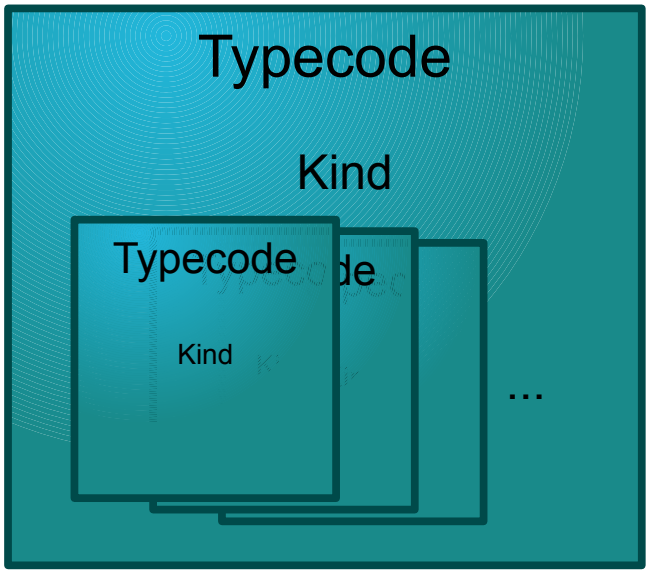
- Create a Typecode which represents the data.
- Generation of a bytecode so the library knows how to serialize or deserialize mentioned data.
- Serialize data into a FastBuffer object using provided eProsima DFB's API for that task.
- Deserialize data previously serialized in the buffer to the user's structures (simple or complex).

This section will describe the basic concepts of these four steps that a developer has to follow to use eProsima DFB.

2.1 Describing data through Typecode

A Typecode object is used by eProsima DFB to define the data according to what any user wants to use. That is why the data Typecode must be created by user according to the data that will be serialized later. If this definition is made wrong, eProsima DBF does nor warranty a correct ending of the execution.

eProsima DFB's typecode is defined by using a kind (which specifies data type) and it could contain other Typecode objects inside it. The reason for that representation is to provide users a way to define complex data types such as structures, inserting other simple or complex data types inside them.



2.1.1 Typecode creation syntax

2.1.1.1 Simple types supported

eProsima DFB supports a variety of simple types that the developer can create via the API functions provided. The following table shows the supported simple types and the function that odd to be called for creating them.

TABLE : SPECIFYING SIMPLE TYPES IN IDL FOR C++

User Type	C++ Sample Data Type	Function for creation
char	`char charSample;`	`createCharacter()`
short	`int16_t shortSample;`	`createShort()`
unsigned short	`uint16_t shortSample;`	`createShort()`
int	`int32_t intSample;`	`createInteger()`
unsigned int	`uint32_t intSample;`	`createInteger()`
long	`int32_t longSample;`	`createLong()`
unsigned long	`uint32_t longSample;`	`createLong()`
float	`Float floatSample;`	`createFloat()`
double	`Double doubleSample;`	`createDouble()`
string	`std::string stringSample;`	`createString()`
bool	`Bool boolSample;`	`createBoolean()`

2.1.1.2 Complex types supported

Complex types can be created by the developer using simple types or other complex types if supported. These complex types can be used as containers for other simple types or just to define complex data structures such as arrays. The following table shows the supported complex types, how they are defined in C++ language, and wich is the function for their creation.

TABLE : SPECIFYING COMPLEX TYPES IN IDL FOR C++

User Type	C++ Sample Data Type	Function for creation
Struct (see note below)	`typedef struct structSample{};`	`createStruct()`
array	`std::array<type, size> arraySample1();` `type arraySample2[][]...[];`	`createArray(type, nDims, dim1, dim2, ..., dimN)`
bounded sequence	`std::vector<type> sequenceSample(size);`	`createSequence(type, size)`

Note: Structures may contain any kind of data type, but the other complex types cannot contain complex data types.

2.1.1.3 Simple data types definition

In this section will be described all functions used for simple data definition through the creation of a typecode associated to it. These functions are used always the same way, and have been designed in order to provide user a simple API for data description.

The functions of this API can be accessed by scoping into the existing namespace called "DynamicFastBuffers", and scoping then into "TypecodeAPI" class.

Character data definition:
To describe a character data type, the function "createCharacter" will be used. By calling the mentioned Typecode API function, users will be able to obtain a Typecode object which describes this kind of data.

This function is executed as is shown below:

```
DynamicFastBuffers::Typecode *characterTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createCharacter();
```

By using this function, in the previously declared pointer to a Typecode object will be arranged a new Typecode object that defines a character data type.

Short data definition:
In order to describe a short data type that odd to be serialized, the function "createShort" defined in TypecodeAPI must be used.

This function is shown next:

```
DynamicFastBuffers::Typecode *shortTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createShort();
```

In the example, inside the object pointed by "shortTypecode" pointer, there will be an instance of "Typecode" whose kind represents a short data type.

Integer data definition:
To define a typecode which represents an integer value defined by user, the function "createInteger" inside "TypecodeAPI" must be used.

```
DynamicFastBuffers::Typecode *integerTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createInteger();
```

Long data definition:
In order to describe a long data type, the function "createInteger" defined in "TypecodeAPI" has to be used. An example of how to use the mentioned function is shown below:

```
DynamicFastBuffers::Typecode *longTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createLong();
```

Float data definition:
To describe a simple precision floating point number, a function named “createFloat” must be used. This function is also defined in “TypecodeAPI” class, that can be scoped through namespace “DynamicFastBuffers”.

In next example is shown how to create a Typecode object associated to a float data type defined by user:

```
DynamicFastBuffers::Typecode *floatTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createFloat();
```

Double data definition:
The function used for describing a double precision floating point number is named “createDouble”, and as the other functions it can be accessed via “TypecodeAPI” class.

```
DynamicFastBuffers::Typecode *doubleTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createDouble();
```

String data definition:
By using eProsima DFB, user can also describe objects defined using std::string data type. The function for describing this kind of objects is “createString”, and it is defined in “TypecodeAPI” class too.

```
DynamicFastBuffers::Typecode *stringTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createString();
```

In this example, it can be seen the way to describe an std::string object by using eProsima DFB Typecode API.

Boolean data definition:
If any user wants to describe a boolean data type for serializing or deserializing it, the “TypecodeAPI” class provides a function called “createBoolean” to do it. This function is used as shown below:

```
DynamicFastBuffers::Typecode *boolTypecode;
characterTypecode = DynamicFastBuffers::TypecodeAPI::createBoolean();
```

2.1.1.4 Complex data types definition

On the other hand, if the user wants to describe complex data types, other kind of functions have to be used. This other functions have parameters that the mentioned user must know in order to use them, which is clearly a difference versus the functions shown in the previous section.

There are three kinds of data that can be described though their typecode by using eProsima DFB. These kinds are:

- Structures

- Arrays
- Sequences

Struct data definition:
When it comes to describing struct data types, user must have in mind that this kind of data types are composed by more simple data types. This means that there will be "Typecode" objects inside the outer "Typecode" object used to describe the structure.

In this case, the function used to create the typecode for the structure is a little bit different than the previously mentioned functions. Nevertheless, the name is way similar, and it is "createStruct".

There are two main ways of describing a structure typecode. The first one is by creating first the object and then using "addMembers" function to specify one or more "Typecode" objects that will be inserted into it. The second one is based on adding the inner typecode definitions as parameters of the "createStruct" function.

Now an example of how to use these two ways is shown:

- **Creation using "createStruct" and then "addMembers":**
  In the previous image a structure containing an integer inside is described. First, an object instance of "Typecode" is created, and then the function "addMembers" is used to add an integer data type into it. If the user wants to add more data definitions to the structure, they can be added later by calling the same function (having in mind this means adding them after the ones that are already added).

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct();

DynamicFastBuffers::TypecodeAPI::addMembers(
       structTypecode,
       DynamicFastBuffers::TypecodeAPI::createInteger(),
       NULL
);
```

  In case a wrong call to this function is done by not specifying the destination "Typecode" object of the structure, a "WrongParamException" exception object will be thrown. On the other hand, if the user does not provide any "Typecode" objects to insert, an exception object instance of "NotEnoughParamsException" will be thrown. Finally, if the destination "Typecode" object is not a structure type description, an exception "WrongTypeException" will be thrown.

- **Creation using only "createStruct":**
  The other way of describing a structure is by adding the inner data types as parameters when executing the function provided for creating the object. An example of how to do this can be seen in next image:

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
       DynamicFastBuffers::TypecodeAPI::createInteger(),
       DynamicFastBuffers::TypecodeAPI::createString(),
       DynamicFastBuffers::TypecodeAPI::createStruct(
              DynamicFastBuffers::TypecodeAPI::createShort(),
              DynamicFastBuffers::TypecodeAPI::createDouble(),
              NULL
       ),
       NULL
);
```

As it is shown in previous image, users can add when creating the typecode for describing the struct data type, other “Typecode” objects (that can be created earlier and inserted in any order). In this example, a structure containing an integer, a string and another struct has been described. Note that struct creations must have as last parameter a NULL value, for there is no way of knowing how many objects the user wants to insert.

Whether if the user chooses to use function “addMembers” or the default function “createStruct” with parameters to insert more data types into a structure definition, it is important to know that the order of the insertions is determinant for the “Typecode” creation to be successful. This means that the order must be the same as the order of the data created by user, for in other case serialization may fail.

The other concerning feature that users have to keep in mind is that a structure can be done by adding any type of data definition provided by eProsima DFB, but this is not possible in the opposite way, not only in simple data types but also in arrays and sequences.

Array data definition:
Other complex datum types that can be described by using eProsima DFB library are arrays. An array is described by the kind of data that it holds inside, and the length of the dimensions.

For example, an integer matrix with two rows and three columns will be described using typecode as an object by specifying that it has two dimensions, having the first one a length of two and the second one a length of three.

This concrete “Typecode” object can be created using the function named “createArray”, defined in “TypecodeAPI” class. This function receives as parameters a typecode indicating the kind of data that will be stored inside it, followed by an integer which specifies the number of dimensions. After that, the length of each and everyone of the dimensions have to be provided.

In the nextimage, a creation of a “Typecode” object which represents an array can be seen. The first parameter defines the type of data that the array will hold, in this case long values. Afterwards, the first integer (four in this case) specifies how many dimensions will be provided, being the next four integers the respective lengths of each

dimension. Finally, a zero value must be inserted in order to know that no more dimensions will be specified.

```
DynamicFastBuffers::Typecode *arrayTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createArray(
       DynamicFastBuffers::TypecodeAPI::createLong(),
       4, 10, 10, 3, 5,
       0
);
```

In this function, if a number of dimensions (first integer) lower than one is specified, an object instance of “NotEnoughParamsException” will be thrown. On the other hand, if any of the dimension's length is lower than one (same case as before), a “WrongParamException” exception object will be thrown. Finally, “NotEnoughParamsException” object will be thrown if the number of dimensions

```
DynamicFastBuffers::Typecode *sequenceTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createSequence(
       DynamicFastBuffers::TypecodeAPI::createShort(),
       20
);

DynamicFastBuffers::TypecodeAPI::deleteTypecode(sequenceTypecode);
```

defined is not equal to the number of dimensions really inserted by user.

Sequence data definition:

The last complex data that eProsima DFB permits to describe is called sequence. This kind of type is represented in C++ as a vector of objects. These objects have to be simple data types (excluding std:string), and cannot be under any circumstance complex types.

A sequence is defined by using a typecode object for describing the kind of data that would be stored inside, and an integer greater than zero which specifies the maximum length of the vector that will hold the datum. The function used to describe this kind of data type is named “createSequence”, and it is also defined in “TypecodeAPI” class.

In this example, a sequence of twenty (at the most) short values is described for serialization. The first parameter of the “createSequence” function is used to determine which kind of data type is stored in the sequence, and the second one is the maximum number of data that can be inserted in it.

2.1.1.5 Data definition destruction

Once the user has created the typecode for definition of the types used by the application, the “Typecode” objects must be destroyed in order to not waste memory used and reserved.

```
DynamicFastBuffers::Typecode *sequenceTypecode;
sequenceTypecode = DynamicFastBuffers::TypecodeAPI::createSequence(
       DynamicFastBuffers::TypecodeAPI::createShort(),
       20
);
```

For this action, a function named “deleteTypecode” is provided, which eliminates all data reserved inside the “Typecode” object.

In case of complex data types (structures, arrays and sequences), a single call to this function giving as parameter the upper typecode will be enough to erase all reserved data.

2.1.1.6 Calculating serialized data size

For calculating the size of the data when serialized, so that the buffer could be initialized properly, a function named “checkSerializedDataSize” can be called to perform this functionality.

This function receives a pointer to a Typecode object, and from this object the needed buffer size will be calculated. If more than one Typecode has been created, the size of all of them must be added and stored into a variable, which will be used for the buffer creation.

2.1.1.7 Example

Later in this document there will be added different examples for bytecode generation and data serialization and deserialization. Due to this, a “Typecode” example object is now defined, using it from now on.

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
       DynamicFastBuffers::TypecodeAPI::createInteger(),
       DynamicFastBuffers::TypecodeAPI::createString(),
       DynamicFastBuffers::TypecodeAPI::createStruct(
              DynamicFastBuffers::TypecodeAPI::createShort(),
              DynamicFastBuffers::TypecodeAPI::createDouble(),
              NULL
       ),
       NULL
);
```

As it can be seen in the previous image, the typecode describes a structure with three kinds of data inside it, an integer, an std::string object and another structure. This second structure has a short and a double data type inside.

2.1.1.8 Limitations

The limitations for the typecode creation that must be deeply considered are the following:

- No data types can be added into simple data types
- While structure data types can have inside any other types, arrays and sequences cannot have complex data types, and no string or boolean types neither.
- Not union or enum data types can be described using this version of eProsima Dynamic Fast Buffers.

2.2 Generating specific Bytecode given a Typecode object

Once the typecode for describing a concrete data type is defined, the generation of a bytecode associated to it is necessary in order to do the data serialization or deserialization.

There are to kinds of bytecode that could be generated, one for doing data serialization and another for data deserialization. This is so because the functions to perform the operations are not the same, so a specific bytecode must be solicited for both of them.

```
DynamicFastBuffers::Bytecode *bytecode;
bytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
       structTypecode,
       DynamicFastBuffers::flag::FLAG_TRUE
);
```

The API class that holds the functionality for generating “Bytecode” objects can be accessed by scoping into “DynamicFastBuffers” namespace, and looking for a class named “BytecodeAPI”.

2.2.1 Bytecode for data serialization:

As it has been mentioned before, a specific bytecode for data serialization must be created, different than the bytecode for data deserialization.

eProsima DFB provides a simple API for performing this task, by executing a mere function in which users must tell via parameter they want to serialize some data. Once the “Typecode” object has been created, the generation of a “Bytecode” object must be done the following way:
For the example shown in last image, the typecode which describes the data defined in previous chapter has been used. The code above shows a call to a function named “generateBytecode”, whose parameters are:

- structTypecode: The “Typecode” object previously created.
- DynamicFastBuffers::flag::FLAG_TRUE: Constant value used to specify the library that the generated bytecode must be for data serialization.

By executing this function, user receives an object instance of “Bytecode” class which contains an internal structure. This structure is a list of pointers to the functions of eProsima Fast Buffers API that must be executed for serializing this kind of data.

```
bytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
       structTypecode,
       DynamicFastBuffers::flag::FLAG_FALSE
);
```

If a NULL value is inserted as first parameter, en exception will occur. This exception is an object defined in this library, belonging to “WrongParamException” class.

2.2.2 Bytecode for data deserialization:

The same way a bytecode is generated for serializing data must be done for deserializing it. The function of the “BytecodeAPI” class that odd to be executed is the very one executed in the last example, but specifying a different flag.

An example of how to do this will be shown now:

In the last example, the same “Bytecode” object has been used. This can be done, but keeping in mind that its internal data will be overwritten, so it will not be valid for doing data serialization, but only deserialization.

The main difference between this two function calls is only the flag specified. In this case parameter are the following:

- structTypecode: The “Typecode” object previously created.
- DynamicFastBuffers::flag::FLAG_FALSE: Constant value used to specify the library that the generated bytecode must be for data deserialization.

By executing this function, user receives an object instance of “Bytecode” class which contains an internal structure. This structure is a list of pointers to the functions of eProsima Fast Buffers API that must be executed for deserializing this kind of data.

If a NULL value is inserted as first parameter, en exception will occur. This exception is an object defined by a class named “WrongParamException”.

2.3 Data serialization and deserialization

eProsima DFB provides an API that can be easily used for data serialization and deserialization. There are two functions defined in “SerializerAPI” class. The users of the library can have access to this class by scoping into “DynamicFastBuffers” namespace.

In order to do the mapping of data types into a buffer (where serialized data will be stored), this types must be previously user-defined by coding them. For example, if any user defines an integer value (int), it can be then serialized by creating a typecode description for this data, generating a bytecode then and using it for doing the mentioned serialization. The same way happens with other data types, such as floating point numbers (float), structures, sequences, etc.

2.3.1 Data serialization:

When it comes to data serialization, a function named “serialize” defined in “SerializerAPI” class must be executed. This function receives as parameters a void pointer to the data defined by user, a “Bytecode” object previously generated using

“BytecodeAPI” functions, and a “FastCdr” object created using eProsima Fast Buffers library.

An example of how to perform this action is shown in the next image:

```
DynamicFastBuffers::SerializerAPI::serialize(
       (void*) &data,
       bytecode,
       &cdr
);
```

As it can be seen in the image below, “serialize” function parameters are:

- (void*) &data: Void pointer to a variable named “data”, which is the data type defined by user.
- bytecode: “Bytecode” object previously generated using “generateBytecode” function from “BytecodeAPI” class.
- &cdr: “FastCdr” object created using eProsima Fast Buffers library.

Once this operation has been performed, inside the buffer existent in “FastCdr” object all user data must have been serialized. To recover this data, a deserialization operation must be done, which is explained in next section.

2.3.2 Data deserialization

Concerning data deserialization, another function provided by eProsima DFB must be executed. This function to perform this operation is named “serialize” and it is allocated inside a class named “SerializerAPI”, which can be accessed by scoping into “DynamicFastBuffers” namespace.

Next image shows an example of how to do data deserialization:

```
DynamicFastBuffers::SerializerAPI::deserialize(
       (void*) &data,
       bytecode,
       &cdr
);
```

As it can be seen in the image, the parameters of the function are the same than in “serialize” function, but the procedure is different. In this case, data already serialized is located inside a “FastCdr” object, and the deserialization will be done into the object pointed by “data” variable. It does not matter which is the data type of the variable pointed by the void pointer, as long as the typecode specified at the beginning of the execution is well defined according do the data.

3 eProsima Dynamic Fast Buffers API

The API for accessing eProsima DFB is define within three main classes. This classes names are “TypecodeAPI”, “BytecodeAPI” and “SerializerAPI”, and they can all be accessed via “DynamicFastBuffers” namespace.

The functions that conform the library's API are listed in next subsections. The behavior for each function will not be described due to the fact that it has already been described in previous chapters of this document.

3.1 Typecode creation API

Public functions located in “DynamicFastBuffers::TypecodeAPI::”:

- static Typecode* createCharacter()
- static Typecode* createShort()
- static Typecode* createInteger()
- static Typecode* createLong()
- static Typecode* createFloat()
- static Typecode* createDouble()
- static Typecode* createString()
- static Typecode* createBoolean()
- static Typecode* createStruct(Typecode* init, …)
- static Typecode* createArray(Typecode *type, int nDims, int dim1, …)
- static Typecode* createSequence(Typecode *type, int maxLength)
- static void addMembers(Typecode *dest, …)
- static void deleteTypecode(Typecode *tc)
- static int checkSerializedDataSize(Typecode *tc)

3.2 Bytecode generation API

Public functions located in “DynamicFastBuffers::BytecodeAPI::”:

- static Bytecode* generateBytecode(Typecode *typecode, flag flag);

3.3 Data serialization/deserialization API

Public functions located in “DynamicFastBuffers::SerializerAPI::”:

- static void serialize(void *data, Bytecode *bytecode, eProsima::FastCdr *cdr)
- static void deserialize(void *data, Bytecode *bytecode, eProsima::FastCdr *cdr)

4 Known Issues

The only issue that has been found while designing this library is the following:

- eProsima Dynamic Fast Buffers does not warranty a correct serialization or deserialization of any data type if there is no match between typecode created for describing the mentioned data and the data itself.

5 HelloWorld example in Visual Studio 2010

In this section an example will be explained step by step on how to create a new project on Visual Studio 2010 which user eProsima DFB library. A complex structure data type will be created and represented via typecode. This data will be serialized and deserialized in order to confirm the correct execution of the functions.

The example will be made and compiled for a 64-bit windows 7 OS. And for that reason a 64-bit architecture compilation of the library will be selected.

5.1 Setting up environment

First thing that odd to be done is to create a new project named “HelloWorldDFB” and have sure that all references to external libraries (in this case eProsima Fast Buffers and eProsima Dynamic Fast Buffers) are correctly set.

The linked library files will be in a directory which has to be included as additional directory. The library version will be automatically linked. Having done this, all classes and headers from this projects can be accessed.

5.2 Including headers

In the class containing the main function, which will be the entry point to the application, some headers must be included.

```
#include “FastCdr.h”
#include “TypecodeAPI.h”
#include “BytecodeAPI.h”
#include “SerializerAPI.h”
#include “CommonData.h”

int main()
{
	…
}
```

Note that a header exists for each API available for users. Furthermore an extra inclusion must be done in order to create new “Typecode” and “Bytecode” entities, and it is named “CommonData.h”.

5.3 Data declaration and “FastCdr” object creation

Open the Visual Studio 2010 solution “HelloWorldDFB.sln”. In the file where function “main” is located, two tasks odd to be done. First one is to define the data that will be serialized, in this case a structure (outside any function, in global scope):

```
Struct HelloWorldStruct{
	short att1;
	int att2;
	string att3;
};
```

This structure is formed by three inner variables, first one is of type short, second one of type int and the third one is a std::string object.
The second task is to declare and instantiate, inside “main” function, a “FastCdr” object with a “FastBuffer” object inside it.

```
Char buffer[500];
eProsima::FastBuffer cdrBuffer(buffer, 500);
eProsima::FastCdr cdr(cdrBuffer);
```

By coding this three lines, a buffer with five hundred free positions has been declared and inserted into a “FastCdr” object. There will be in this buffer where all data is going to be serialized and from which all data will also be deserialized.

Obviously it is not enough with the structure definition, if a user wants to serialize that kind of data type two objects have to be created, one for reading data to do the serialization procedure, and another in which data will be recovered.

```
HelloWorldStruct inputStruct, outputStruct;
inputStruct.att1 = 10;
inputStruct.att2 = 2;
inputStruct.att3 = “Hello World!”;
```

As it can be seen in the previous image, two structures have been declared. The one named “inputStruct” is where all data will be initialized and from which will be serialized. The other one, named “outputStruct” is where data will be deserialized in the end.

5.4 Typecode creation

Once data is defined it must be described via creation of an object instance of “Typecode” class. The following code shows how to do it:

```
DynamicFastBuffers::Typecode *structTypecode;
structTypecode = DynamicFastBuffers::TypecodeAPI::createStruct(
       DynamicFastBuffers::TypecodeAPI::createShort(),
       DynamicFastBuffers::TypecodeAPI::createInteger(),
       DynamicFastBuffers::TypecodeAPI::createString(),
       NULL
);
```

Note that the internal structure of the typecode is created with exactly the same data types and exactly in the same order. This must be done to ensure the serialization process finishes successfully.

5.5 Bytecode generation

Now that the typecode that represents the data defined by user has been created, it is time for the generation ob a bytecode associated to it. In this case, two bytecodes will be generated, one for data serialization and another for data deserialization. Keep in mind that the same object can be used for both operations, but it has to be redefined for each purpose.

```
DynamicFastBuffers::Bytecode *serializationBytecode;
DynamicFastBuffers::Bytecode *deserializationBytecode;

serializationBytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
       structTypecode,
       DynamicFastBuffers::flag::FLAG_TRUE
);
deserializationBytecode = DynamicFastBuffers::BytecodeAPI::generateBytecode(
       structTypecode,
       DynamicFastBuffers::flag::FLAG_FALSE
);
```

As it can be seen in the image above, two objects have been created. The one named “serializationBytecode” contains pointers to functions used for data serialization, and the one named “deserializationBytecode” pointers to functions for data deserialization.

5.6 Data serialization

For data serialization, function named “serialize” defined within class “serializerAPI” will be used. The code for serializing data is shown below:

```
DynamicFastBuffers::SerializerAPI::serialize(
       (void*) &inputStruct,
       serializationBytecode,
       &cdr
);
```

At it has been mentioned in previous sections, note that the data defined by user is passed as parameter by making a casting to a void pointer. So it is really the bytecode generated from the typecode description of the data where the memory structure of the mentioned data is defined.

5.7 Buffer reset

If the “FastCdr” object used for deserializing data is the same that have been used for serializing it, the inner buffer of this object must be reset by executing the following function:

```
cdr.reset()
```

5.8 Data deserialization and Typecode destruction

Finally, in order to do data deserialization to a previously declared variable, a function named “deserialize” have to be called. The code for this call is next:

```
DynamicFastBuffers::SerializerAPI::deserialize(
       (void*) &outputStruct,
       serializationBytecode,
       &cdr
);
DynamicFastBuffers::TypecodeAPI::deleteTypecode(structTypecode);
```

In the image above, after the execution of function “deserialize”, all data that has been serialized before will be recovered into the object pointed by “outputStruct” void pointer.

To not waste memory, a final call to “deleteTypecode” function defined in “TypecodeAPI” class will be done.