**Binary message Functions Overview**

addSizeBytes(std::shared\_ptr<std::list<uint8\_t>>

bytes, uint64\_t size)

**Main purpose:**

Writes the passed in information to the bytes list.

Typically used by other functions to write the size of a message to the bytes list.

- Note: Decoder needs the size of a message to know when the message ends

**Used By:**

encodeLableBytes()

encodeBytes() // Both Element and Object functions

**Relies On:**

N/A

**Inputs:**

**bytes:**

**Type:**

Shared pointer to a list

**Use:**

Contains the binary message information as it is built

**Format:**

[info,info,info] // Typically in hex format for the info

**size:**

Type: uint64\_t

Use: contains size information of data

**Outputs:**

bytes->push\_back((uint8\_t)size);

Use: Writes the information in the size variable to the bytes list

Condition: This occurs only when the size <= 127 bits or 0x7F in hex

bytes->push\_back(byteCount | 0x80);

Use: Writes the number of bytes needed to store the size of the information in the size variable.

Also changes MSB (Most significant Byte) to 1s, indicating to the decoder this size variable needed multiple bytes to represent it.

bytes->push\_back(byte)

Use: Writes the information in the size variable to the byte list.

**Operation:**

The two main cases of this function are if the 64 bit number is greater or less than 127

Case 1: number <= 127

If the number is <= 127 or 0x7F, it can be properly represented by one byte

Since the number is <= 127 it is cast to a uin8\_t or const char which is 8 bits and the information in the size variable is written to the bytes list

Case 2: number > 127

There are < 1 bytes in the message

The function creates a 64-bit mask 0xFF00000000000000L this is used to locate the first non-zero byte (see Example below).

Counts (after most significant byte) how many of the top bytes are zero

For every byte that is zero it incraments the "zeroCount" variable.

When a non-zero byte is found, the loop ends

Determines the number of bytes the message requires

byteCount = 8 - zeroCount (Total number of bytes in 64 bit minus the zero bytes gives the number of bytes with data)

Creates a indicator flag for future decoding

byteCount | 0x80 which masks the MSB (most significant byte) to 1s, indicating this is a multi-byte message, and appends the bytes required to the end.

ex byteCount = 2, (byteCount | 0x80) = 0x82

Extract each byte and send each byte to output.

ByteCount and subtract 1 (byteCount - 1 or 6 - 1 = 5) this orders the bytes from 0 (LSB) to x (MSB) where x is the max number of bytes,

in our case 5.

Shift the MSB to the LSB position by doing number >> (index \* 8) where index is the byteCount - 1

Read the LSB in the size variable after shifting and stores it in the temporary byte variable

The byte variable is then read and its information is pushed onto the bytes list

Loop repeats until index reaches 0

**Example:**

0x123456789ABC or 20015998343868

**64bit representation:**

Byte[7] (MSB): 0x00

Byte[6]: 0x00

Byte[5]: 0x12

Byte[4]: 0x34

Byte[3]: 0x56

Byte[2]: 0x78

Byte[1]: 0x9A

Byte[0] (LSB): 0xBC

**Determine Case:**

(size !< 127) therfore use Case 2

**Determine Bytes to represent number:**

**Determining the number of 0 bytes:**

Mask Value (Hex) |mask & number | Zero? | zeroCount

0xFF00000000000000|0x0000123456789ABC| Y | 1

0x00FF000000000000|0x0000123456789ABC| Y | 2

0x0000FF0000000000|0x0000123456789ABC| N | Break

zeroCount = 2

**Deterniming number of bytes to represent size:**

byteCount = 8 - zeroCount = 8-2 = 6

6 Bytes are required to represent the size

**Sending size Byte Count:**

bytes->push\_back(byteCount | 0x80);

Since byteCount is 6 then this becomes 0x86 or 10000110

The MSB is set to indicate this is a multi-byte transmission and the last 7 bits indicate the number of bytes in the message

**Writing bytes to output:**

Set index to byteCount - 1 to go from 0 to x instead of 1 to x bytes.

Start at MSB

**Loop 1:**

index = 5

number = 0x123456789ABC

number >> (index \* 8) & 0xFF

number >> (40) & 0xFF

shifting number right 40 bits results in number = 0x12

byte = 0x12

**Loop 2:**

index = 4

number 0x123456789ABC

number >> (32) & 0xFF

number = 0x1234

byte = 0x34 since 0x1234 is & with 0xFF masking LSB off

Loop repeats until all bytes have been read and sent to output.

void BinaryMessage::encodeLabelBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, std::string label)

**Main purpose:**

Serializes a binary string message and writes it to output

**Used By:**

encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Object object)

Note: Object function

encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Element element)

Note: Element function

**Relies On:**

addSizeBytes()

**Operation:**

1. Pushes the type of message onto the stream (STRING or 13) defined in BinaryMessage.hpp

2. Calls addSizeBytes(bytes, label.size() ); to write the size of the string to the bytes list

3. Pushes the information in the label onto the bytes list

**Inputs:**

**- label:**

Type: string

Use: Contains the String message to be Encoded

- bytes

Type: Shared pointer to a list

Use: Contains the binary message information as it is built

Format: [info,info,info] // Typically in hex format for the info

**Outputs(In-Order):**

- bytes->push\_back(TYPE::STRING);

- Writes STRING or 13 to the bytes list for future decoding

- addSizeBytes(bytes, label.size() );

- Writes the size of the label to the bytes list for future decoding

- bytes->push\_back(label[index]);

- Writes the data inside the label to the bytes list

**Example:**

label = 200 repeating A's

bytes->push\_back(TYPE::STRING)

Pushes 13 onto the Bytes list

[13]

addSizeBytes(bytes,label.size() );

Pushes the multibyte flag with the byte size representation and then the size of the label

[0x0D, 0x81, 0xC8]

13 flag 200

for(int index=0; index < label.size(); index++){

bytes->push\_back(label[index]);

}

Pushes the 200 repeating A's onto the Byte list

[0x0D, 0x81, 0xC8, 0x41, 0x41, 0x41, ....., 0x41]

void BinaryMessage::encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Object object)

**Main purpose:**

Takes in a Object struct and converts its entire strucutre (label, elements, and child objects) into binary format

**Used By:**

getBytes()

**Relies on:**

encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Element element)

Note: This is the function that encodes Element variables

addSizeBytes()

**Operation:**

Two parameters passed in are a bytes list and a object variable

Encode any data in the Label of the Object struct utalizing the encodeLabelBytes(bytes, object.label) function

Note: encodeLabelBytes() pushes its own indicator (TYPE::STRING) onto the bytes list

Push another indicator for a TYPE::OBJECT, indicating the data that follows is not another string.

Note: Elements and child objects are next, and are handeled differently (by the decoder) than a string.

Place the size of the element list onto the byte list, with the addSizeBytes() function

Encode each element using the void BinaryMessage::encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Element element) function

Note: The encodeBytes() function used is for Element encoding not Object encoding. Elements consist of Boolean, Char,...ect.

The second loop encodes all the children elements

**Inputs:**

**object:**

**Type:**

Object Instantiation

**Use:**

Source of all the structured data that will be converted into a binary format

**Format:**

string label

uint8\_t type //8 bit integer

vector<Element> elementList

vector containing ordered collection of Element objects

vector<Object> children

vector containing sub-Objects or children. Creates Tree like representation

Ex. Top-level object contains an overall configuration, it has children objects containing "Sensors", "Motors", ect

Each of those Children can also have children objects.

bytes

**Type:**

Shared pointer to a list

**Use:**

Contains the binary message information as it is built

**Format:**

[info,info,info] // Typically in hex format for the info

**Example:**

**For Reference:**

TYPE::STRING = 13 or 0x0D

TYPE::OBJECT = 0 or 0x00

TYPE::INT8 = 3 or 0x03

These are defined in BinaryMessage.hpp TYPE enum

Object Sample Data: (See Object struct in BinaryMessage.hpp for structure details)

Object Label = "Test"

Elements (See Elements struct in BinaryMessage.hpp for details)

For single value elements they contain a Label, Data Type, and the actual Data

Element Label: "Value"

Type: INT8 (from BinaryMessage.hpp enum defined as 3)

Data: 42

**Encoding Object Label:**

object.label is encoded with encodeLabelBytes(bytes, object.label);

TYPE::STRING (0x0D) is pushed onto the bytes list

bytes = [0x0D]

addSizeBytes(bytes, label.size()) pushes the object.label size onto the bytes list

string "Test" is length 4, which is < 127 so 0x04 is written to the bytes list

bytes = [0x0D, 0x04]

For loop calls bytes->push\_back(label[index]); which pushes each character onto the bytes list

'T' = 0x54

'e' = 0x65

's' = 0x73

't' = 0x74

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74]

type size data ------------------

**Encoding Object Elements:**

bytes->push\_back(TYPE::OBJECT) pushes OBJECT (0x00) onto the bytes list

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00]

type size data ------------------ type

addSizeBytes(bytes, object.elementList.size()) Pushes the size of the elementList onto the bytes list

elementList.size() = 1 or 0x01 which is pushed onto the bytes list

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01]

type size data ------------------ type size

for (auto iterator = object.elementList.begin(); iterator != object.elementList.end(); iterator++)

Itterates through the start of the elementlist to the end

Note:auto is used to let the compiler automatically deduce the iterator type

Here it will be somthing like std::vector<Element>::iterator

encodeBytes(bytes, \*iterator) encodes and pushes each element in the list onto the bytes list

NOTE: This function shares the same name, but it accepts inputs of type Element not Object.

Pushing element label onto byte list

Note: When it dereferences the itterator the first thing to be encoded is the Element Label

encodeLabelBytes(bytes, element.label)

element.label = "value"

Pushes TYPE::STRING (0x0D) onto the bytes list

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D]

type size data ------------------ type size type

Pushes size of input onto the bytes list

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D, 0x05]

type size data ------------------ type size type size

Pushes string data onto bytes list "value"

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D, 0x05, 0x76, 0x61, 0x6C, 0x75, 0x65]

type size data ------------------ type size type size data ------------------------

Pushing Element Type onto byte list

encodeLabelBytes(bytes, element.type)

element.type = 0x03

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D, 0x05, 0x76, 0x61, 0x6C, 0x75, 0x65, 0x03]

type size data ------------------ type size type size data ------------------------ type

Pushing Element Data onto byte list

encodeLabelBytes(bytes, element.data)

element.data = 42 or 0x2A

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D, 0x05, 0x76, 0x61, 0x6C, 0x75, 0x65, 0x03, 0x2A]

type size data ------------------ type size type size data ------------------------ type data

**Encoding Childrent Objects:**

There are not Children Objects, so addSizeBytes(bytes, object.children.size()) returns 0x00

**Final Bytes List:**

bytes = [0x0D, 0x04, 0x54, 0x65, 0x73, 0x74, 0x00, 0x01, 0x0D, 0x05, 0x76, 0x61, 0x6C, 0x75, 0x65, 0x03, 0x2A, 0x00]

type size data ------------------ type size type size data ------------------------ type data children

void BinaryMessage::encodeBytes(std::shared\_ptr<std::list<uint8\_t>> bytes, Element element)

**Main Purpose:**

**Used By:**

**Relies On:**

**Operation:**

**Inputs:**

**Outputs(In-Order):**

**Example:**

**Function Template**

**Function Name**

**Main purpose:**

**Used By:**

**Relies On:**

**Operation:**

**Inputs:**

**Outputs(In-Order):**

**Example:**