MyProject

Generated by Doxygen 1.9.1

Chapter 1

Test List

Global TEST (parse_SFLV_init_test, returns_correct_format)

parse_SFLV_init_test_returns_correct_format

Global TEST (parse_SFLV_integer_node, returns_correct_format)

parse_SFLV_integer_node_returns_correct_format

Global TEST (append_string_test, multiple_appends)

append_string_test_multiple_appends

Global TEST (read_str_test, simple_string)

read_str_test_simple_string

Global TEST (read_str_test, empty_string)

read_str_test_empty_string

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 arguments Struct Reference

Represents the command line arguments.

Data Fields

- bool verbose

  Verbose flag for extra output.
- bool silent

  Silent flag for minimal output.
- char * operation

  Operation mode, such as "encode".
- char * schema_dictionary

  Path to the schema dictionary file.
- char * annotation_dictionary

  Path to the annotation dictionary file.
- char * json_file

  Path to the JSON file.
- char * bej_output_file

  Path to the BEJ output file.
- char * pdr_map_file_encode

  Path to the PDR map file for encoding.
- char * bej_encoded_file

  Path to the BEJ encoded file.
- char * pdr_map_file_decode

  Path to the PDR map file for decoding.

4.1.1 Detailed Description

Represents the command line arguments.

The arguments structure holds all command line arguments which can be provided by the user.

4.1.2 Field Documentation

4.1.2.1 annotation_dictionary

```
char* arguments::annotation_dictionary
```

Path to the annotation dictionary file.

4.1.2.2 bej_encoded_file

```
char* arguments::bej_encoded_file
```

Path to the BEJ encoded file.

4.1.2.3 bej_output_file

```
char* arguments::bej_output_file
```

Path to the BEJ output file.

4.1.2.4 json_file

```
char* arguments::json_file
```

Path to the JSON file.

4.1.2.5 operation

```
char* arguments::operation
```

Operation mode, such as "encode".

4.1.2.6 pdr_map_file_decode

```
char* arguments::pdr_map_file_decode
```

Path to the PDR map file for decoding.

4.1.2.7 pdr_map_file_encode

```
char* arguments::pdr_map_file_encode
```

Path to the PDR map file for encoding.

4.1.2.8 schema_dictionary

```
char* arguments::schema_dictionary
```

Path to the schema dictionary file.

4.1.2.9 silent

```
bool arguments::silent
```

Silent flag for minimal output.

4.1.2.10 verbose

```
bool arguments::verbose
```

Verbose flag for extra output.

The documentation for this struct was generated from the following file:

- src/encoder_decoder.c

4.2 bej_node Struct Reference

Represents a single node in the BEJ data.

```
#include <bej_decoder.h>
```

Data Fields

- unsigned char dictionary_type

  The type of dictionary (schema or annotation).
- unsigned char sequence

  The sequence number of the node.
- unsigned char format

  The format of the data (e.g., INTEGER, ENUM, STRING, ARRAY).
- unsigned int length

  The length of the data value.
- unsigned int count

  The count of elements in case of ARRAY format.
- void ** value

  A pointer to the data value.

4.2.1 Detailed Description

Represents a single node in the BEJ data.

The bej_node structure holds the information required for a single BEJ data node.

4.2.2 Field Documentation

4.2.2.1 count

```
unsigned int bej_node::count
```

The count of elements in case of ARRAY format.

4.2.2.2 dictionary_type

```
unsigned char bej_node::dictionary_type
```

The type of dictionary (schema or annotation).

4.2.2.3 format

```
unsigned char bej_node::format
```

The format of the data (e.g., INTEGER, ENUM, STRING, ARRAY).

4.2.2.4 length

```
unsigned int bej_node::length
```

The length of the data value.

4.2.2.5 sequence

```
unsigned char bej_node::sequence
```

The sequence number of the node.

4.2.2.6 value

```
void** bej_node::value
```

A pointer to the data value.

The documentation for this struct was generated from the following file:

- src/bej_decoder.h

4.3 dynamic_string Struct Reference

Represents a dynamic string used for BEJ data parsing.

```
#include <bej_decoder.h>
```

Data Fields

- char * data
- size_t length
- size_t capacity

4.3.1 Detailed Description

Represents a dynamic string used for BEJ data parsing.

The dynamic_string structure is used to build a string dynamically during the BEJ data parsing.

4.3.2 Field Documentation

4.3.2.1 capacity

```
size_t dynamic_string::capacity
```

4.3.2.2 data

```
char* dynamic_string::data
```

4.3.2.3 length

```
size_t dynamic_string::length
```

The documentation for this struct was generated from the following file:

- src/bej_decoder.h

Chapter 5

File Documentation

5.1 src/bej_decoder.c File Reference

```
#include "bej_decoder.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
```

Include dependency graph for bej_decoder.c:

5.2 src/bej_decoder.h File Reference

```
#include <stddef.h>
#include <string.h>
```

Include dependency graph for bej_decoder.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct bej_node

  Represents a single node in the BEJ data.
- struct dynamic_string

  Represents a dynamic string used for BEJ data parsing.

Macros

- #define INITIAL_CAPACITY 20
- #define CAPACITY_INCREASE_STEP 50

Enumerations

- enum data_types { INTEGER = 0x30 , ENUM = 0x40 , STRING = 0x50 , ARRAY = 0x10 }

  Defines the types of data in the BEJ format.
- enum bej_node_to_str_types { WITH_KEY , WITHOUT_KEY }

  Defines the types of parsing for BEJ nodes to strings.

Functions

- struct dynamic_string * decode_bej (unsigned char *data, size_t data_len, unsigned char *schema_↩ dictionary, size_t schema_dictionary_len, const unsigned char *annotation_dictionary, size_t annotation↩ _dictionary_len)

  Decodes BEJ data into a string.

5.2.1 Macro Definition Documentation

5.2.1.1 CAPACITY_INCREASE_STEP

```
#define CAPACITY_INCREASE_STEP 50
```

5.2.1.2 INITIAL_CAPACITY

```
#define INITIAL_CAPACITY 20
```

5.2.2 Enumeration Type Documentation

5.2.2.1 bej_node_to_str_types

```
enum bej_node_to_str_types
```

Defines the types of parsing for BEJ nodes to strings.

This enum lists the two modes of parsing for BEJ nodes into strings. It can either include the key (WITH_KEY) or exclude it (WITHOUT_KEY).

Enumerator

WITH_KEY	
WITHOUT_KEY	

5.2.2.2 data_types

```
enum data_types
```

Defines the types of data in the BEJ format.

This enum lists the different types of data that can be present in BEJ. Currently, it includes INTEGER, ENUM, STRING, and ARRAY.

Enumerator

INTEGER	
ENUM	
STRING	
ARRAY	

5.2.3 Function Documentation

5.2.3.1 decode_bej()

```
struct dynamic_string* decode_bej (
            unsigned char * data,
            size_t data_len,
            unsigned char * schema_dictionary,
            size_t schema_dictionary_len,
            const unsigned char * annotation_dictionary,
            size_t annotation_dictionary_len )
```

Decodes BEJ data into a string.

This function is the key function in the file, that decodes the given BEJ data using the provided dictionaries. It first parses the BEJ data into a BEJ tree, and then converts this tree into a string.

Parameters

data	A pointer to the BEJ data.
data_len	The length of the BEJ data.
schema_dictionary	A pointer to the schema dictionary.
schema_dictionary_len	The length of the schema dictionary.
annotation_dictionary	A pointer to the annotation dictionary.
annotation_dictionary_len	The length of the annotation dictionary.

Returns

A pointer to the dynamic string holding the decoded data.

5.3 src/encoder_decoder.c File Reference

```
#include "bej_decoder.h"
#include <stdio.h>
```

```
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
```
Include dependency graph for encoder_decoder.c:

Data Structures

- struct arguments

  Represents the command line arguments.

Functions

- struct arguments parse_arguments (int argc, char ∗argv[ ])

  Parses the command line arguments.
- size_t read_file (const char ∗file_path, unsigned char ∗∗buffer)

  Reads a file into a buffer.
- char ∗ append_to_filename (const char ∗filename, const char ∗suffix)

  Appends a suffix to the filename.
- void write_to_file (struct dynamic_string ∗str, const char ∗filename)

  Writes the content of a dynamic string to a file.
- int main (int argc, char ∗argv[ ])

  Main function.

5.3.1 Function Documentation

5.3.1.1 append_to_filename()

```
char* append_to_filename (
            const char * filename,
            const char * suffix )
```

Appends a suffix to the filename.

This function appends a given suffix to the filename by replacing the ".bin" extension. If the filename does not have a ".bin" extension, it returns NULL.

Parameters

filename	The original filename.
suffix	The suffix to be appended.

Returns

The new filename with the appended suffix. NULL if the filename does not have a ".bin" extension.

5.3.1.2 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function.

This function is the entry point of the program. It contains the main algorithm of the program: parses the command line arguments, reads binary files, performs BEJ decoding, and writes the decoded message to a file.

Parameters

argc	The argument count.
argv	The argument array.

Returns

The exit status of the program.

5.3.1.3 parse_arguments()

```
struct arguments parse_arguments (
            int argc,
            char * argv[] )
```

Parses the command line arguments.

This function goes through the given command line arguments and extracts them into a structured form for easier use. If the provided arguments are invalid, the function will print an error message and exit.

Returns

The parsed arguments.

5.3.1.4 read_file()

```
size_t read_file (
            const char * file_path,
            unsigned char ** buffer )
```

Reads a file into a buffer.

This function reads a file and places its content into a buffer. It handles file opening, size determination, memory allocation for the buffer, and file reading. If there is any error during these operations, it will print an error message and exit.

Parameters

file_path	The path to the file.
buffer	The pointer to the buffer to fill.

Returns

The size of the file.

5.3.1.5 write_to_file()

```
void write_to_file (
            struct dynamic_string * str,
            const char * filename )
```

Writes the content of a dynamic string to a file.

This function writes the content of a dynamic string to a file with the given filename. If the file cannot be opened, an error message will be printed.

Parameters

str	The dynamic string to be written to a file.
filename	The name of the file.

5.4 src/test/decoder_test.cpp File Reference

```
#include "gtest/gtest.h"
#include "../bej_decoder.h"
```

Include dependency graph for decoder_test.cpp:

Macros

- #define TESTING

Functions

- unsigned char * hex_to_bytes (const char *hex)

  Converts a string of hexadecimal digits into an array of bytes. This function iterates through pairs of characters in the input string, converting each pair from a hexadecimal digit to a byte value using sscanf, and storing the resulting bytes in a dynamically allocated array.
- TEST (parse_SFLV_init_test, returns_correct_format)

  This test case checks the function 'parse_sflv_init' for the correct decoding of format, count, sequence, and dictionary type fields.

- TEST (parse_SFLV_integer_node, returns_correct_format)

  This test case checks the function 'parse_sflv_init' for the correct decoding of format, count, sequence, and dictionary type fields.
- TEST (append_string_test, multiple_appends)

  This test case checks the 'append_string' function for multiple string appends.
- TEST (read_str_test, simple_string)

  This test case checks the function 'read_str_const_ptr' for a simple string.
- TEST (read_str_test, empty_string)

  This test case checks the function 'read_str_const_ptr' for an empty string.
- int main (int argc, char **argv)

5.4.1 Macro Definition Documentation

5.4.1.1 TESTING

```
#define TESTING
```

5.4.2 Function Documentation

5.4.2.1 hex_to_bytes()

```
unsigned char* hex_to_bytes (
            const char * hex )
```

Converts a string of hexadecimal digits into an array of bytes. This function iterates through pairs of characters in the input string, converting each pair from a hexadecimal digit to a byte value using sscanf, and storing the resulting bytes in a dynamically allocated array.

Parameters

hex	A pointer to a null-terminated string containing hexadecimal digits. It is assumed that the string length is even and the string contains only valid hexadecimal digits (0-9, A-F, a-f).

Returns

A pointer to the newly allocated array of bytes. The length of the array is half the length of the input string. The caller is responsible for freeing this memory when it is no longer needed.

5.4.2.2 main()

```
int main (
            int argc,
            char ** argv )
```

5.4.2.3 TEST() [1/5]

```
TEST (
            append_string_test ,
            multiple_appends  )
```

This test case checks the 'append_string' function for multiple string appends.

Test append_string_test_multiple_appends

An initial dynamic string is created with 'create_dynamic_string' and then appended with multiple strings using the 'append_string' function. After each append operation, the test checks if the string has been correctly appended and its length has been updated correctly. The test also checks if the overall string is as expected after each append operation. If the appended string and its length match the expected string and length, the test passes.

5.4.2.4 TEST() [2/5]

```
TEST (
            parse_SFLV_init_test ,
            returns_correct_format  )
```

This test case checks the function 'parse_sflv_init' for the correct decoding of format, count, sequence, and dictionary type fields.

Test parse_SFLV_init_test_returns_correct_format

The input data is a hexadecimal string representing an array of bytes. The 'hex_to_bytes' function is used to convert this string into an array of bytes. The function 'parse_sflv_init' is then expected to parse this byte array into a 'bej_node' structure. The test checks the 'format', 'count', 'sequence', and 'dictionary_type' fields of the returned structure, comparing them to expected values. If they match, the test passes.

5.4.2.5 TEST() [3/5]

```
TEST (
            parse_SFLV_integer_node ,
            returns_correct_format  )
```

This test case checks the function 'parse_sflv_init' for the correct decoding of format, count, sequence, and dictionary type fields.

Test parse_SFLV_integer_node_returns_correct_format

The input data is a hexadecimal string representing an array of bytes. The function 'parse_sflv_init' is expected to parse this byte array into a 'bej_node' structure. The test checks the 'format', 'count', 'sequence', and 'dictionary_↩ type' fields of the second element of the returned structure, comparing them to expected values. If they match, the test passes.

5.4.2.6 TEST() [4/5]

```
TEST (
            read_str_test ,
            empty_string  )
```

This test case checks the function 'read_str_const_ptr' for an empty string.

Test read_str_test_empty_string

The input data is an empty string, encoded as a one-element array of unsigned char containing a null character. The expected output is an empty string. The function is expected to correctly decode the input data and return an empty string. The returned string is compared to the expected output using the 'ASSERT_STREQ' macro. If they match, the test passes.

5.4.2.7 TEST() [5/5]

```
TEST (
            read_str_test ,
            simple_string  )
```

This test case checks the function 'read_str_const_ptr' for a simple string.

Test read_str_test_simple_string

The input data is the string "Hello" encoded as an array of unsigned char. The expected output is the original string "Hello". The function is expected to correctly decode the input data and return the original string. The returned string is compared to the expected output using the 'ASSERT_STREQ' macro. If they match, the test passes.