Json Library Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview 3
Purpose and Characteristics 3
Main Features 3
Resources Used 3
Embedding into a Program 3
Sample Programs 3
Reference Materials 4
2 Main Specifications 5
Handled JSON Document Specifications 5
Partitioned Parsing Feature 5
Partitioned Serializing Feature 6
NULL Value Objects 7
NULL Data Handling 7
Character String Handling 8
3 Using the Library 9
Preparations 9
Parsing JSON Documents 10
Generating JSON Documents 12
Post-processing 12

1 Library Overview

Purpose and Characteristics

The Json library is a lightweight library for parsing/generating JSON documents (JSON-formatted text data). By using this library, applications can obtain the information in a JSON document as tree structured data, and in contrast, data in applications can be output as JSON documents.

Sequential input/output of JSON documents is possible, allowing for reduced memory consumption. In addition, arrays/objects in JSON documents can be referenced using subscripts for intuitive referencing and setting of values.

Main Features

The main features provided by the Json library are as follows.

- Feature for parsing JSON documents and obtaining tree structured data
- Feature for obtaining each element of a JSON document from tree structured data
- Feature for formulating/editing tree structured data
- Feature for outputting (serializing) tree structured data as JSON documents

Resources Used

The Json library uses the following system resources.

Resource	Description
Footprint	Approx. 32 KiB
Work memory	32 bytes for each `Value` object + memory for maintaining the actual value

Embedding into a Program

Include json.h in the source program.

Load the PRX module in the program, as follows.

```
if ( sceSysmoduleLoadModule(SCE_SYSMODULE_JSON) != SCE_OK ) {
    // Error handling
}
```

Upon building the program, link libSceJson_stub.a.

Sample Programs

Sample programs using the Json library are as follows.

sample_code/system/api_json/parse

This is a sample of the basic usage of the parsing feature.

sample_code/system/api_json/serialize

This is a sample of the basic usage of the serializing feature.

Reference Materials

Regarding the JSON format, refer to the following.

- The application/json Media Type for JavaScript Object Notation (JSON) (http://www.rfc-editor.org/rfc/rfc4627.txt, The Internet Society, updated June 2006)
- http://www.json.org
- http://en.wikipedia.org/wiki/JSON

(The above reference destinations have been confirmed as of February 27, 2014. Note that pages may have been subsequently moved or the contents modified.)

2 Main Specifications

This chapter explains the handled JSON document specifications, the partitioned parsing feature, the partitioned serializing feature, and other features as the specifications of the Json library.

Handled JSON Document Specifications

Text Encoding

Only UTF-8 is complied with. In some situations, make sure to not partition multi-byte character expressions.

Partitioned Parsing Feature

JSON documents can be partitioned then parsed. A small buffer for data input will suffice, allowing for reduced memory usage.

To perform partitioned parsing, implement a `DataProvideFunction` type callback function in the application. This function must be created so that 1-byte data is returned every time it is called.

In the following example, the `UserDataProvider` class is defined with a read pointer managing feature added in the `DataProvideFunction` type callback function `cbProvide()` (since this is a simplified version, processing that collects compiles multi-byte characters is omitted). For a usage example, refer to the "Parsing JSON Documents" section.

```
struct UserDataProvider
{
    const char *m_cur;
    const char *m_last;

    UserDataProvider(const char *src, size_t siz_src)
         : m_cur(src)
         , m_last(src + siz_src)
    {
    }
    static int32_t cbProvide(char& data, void *userdata)
    {
         return reinterpret_cast<UserDataProvider*>
                          (userdata)->onProvide(data);
    }

    int32_t onProvide(char& data)
    {
         if (m_cur == m_last) {
              return -1;
         }
         data = (*m_cur++);
         return 0;
    }
};
```

Partitioned Serializing Feature

The serialization processing of JSON documents can be partitioned. A small buffer for the output results will suffice, allowing for reduced memory usage.

To perform partitioned serializing, implement a `DataReceiveFunction` type callback function for the application. This function is called every time serialization of a single element is completed and one element's worth of serialization results are passed. If processing is performed where the passed character string is output to a file each time or sent to a network server, a small buffer in the memory will suffice.

In the following example, the `UserDataReceiver` class is defined with an output destination managing feature added in the `DataReceiveFunction` type callback function `cbReceive()`. For a usage example, refer to the "Generating JSON Documents" section.

```
struct UserDataReceiver
{
  FILE *m_fp;

  UserDataReceiver(const char *path)
      : m_fp(0)
  {
      m_fp = fopen(path, "w");
  }
  ~UserDataReceiver()
  {
      if( m_fp ){
          fclose(m_fp);
      }
  }
  static int32_t cbReceive(Json::String& str, void* user_data)
  {
      return reinterpret_cast<UserDataReceiver*>(user_data)->onReceive(str);
  }
  int32_t onReceive(Json::String& str)
  {
      fwrite(str.c_str(), 1, str.length(), m_fp);
      str.clear();

      return 0;
  }
};
```

NULL Value Objects

In order to improve the robustness of the Json library, the specifications are designed so that a NULL pointer will not be returned no matter the data input so long as it is const referencing. This is implemented by NULL `Value` objects which are special instances of the `Value` class. Specifically, NULL `Value` objects are returned in the following situations.

- When "null" appears in a JSON document
- When an attempt is made to obtain a value for an array in a JSON document and the value exceeds the length of the array
- When a specified element name does not exist in a JSON document and an attempt to obtain the value is made

The values returned by each API of NULL `Value` objects are as follows.

API	Returned Value
`getType()`	`kValueTypeNull`
`getString()`	"" (character string with a length of 0)
`getArray()`	Empty array
`getObject()`	Empty object
`getInteger()`	0
`getUInteger()`	0
`getReal()`	0.0f
`getBoolean()`	false
`getValue()`	NULL `Value` object
`count()`	0

NULL Data Handling

It is possible to customize the behavior for cases where a NULL `Value` object is returned. For example, if you want processing that causes an error, interrupts parse processing, and returns an application-defined default value instead of a NULL `Value` object, implement a `NullAccessFunction` type callback function.

In the following example, the `NullAccess` class is defined with the callback function `NullAccessCB()` set to return a default value. For a usage example, refer to the "Parsing JSON Documents" section.

```
class NullAccess
{
public:
        Json::Value vboolean;
        Json::Value vinteger;
        Json::Value vuinteger;
        Json::Value vreal;
        Json::Value vstring;
        Json::Value vnull;

public:
   NullAccess()
   {
        vboolean.set(false);
        vinteger.set((int64_t)-99);
        vuinteger.set((uint64_t)99);
        vreal.set((double)-0.99);
        vstring.set(Json::String("invalid string"));
        vnull.set(Json::kValueTypeNull);
   }
   ~NullAccess(){};
```

```
        static const Json::Value& NullAccessCB(
            Json::ValueType accesstype,
            const Json::Value* parent,
            void* context)
        {
            printf("CB(%d)[parent:%d]\n", accesstype, parent->getType());

            switch(accesstype){
            case Json::kValueTypeBoolean:
              return reinterpret_cast<NullAccess*>(context)->vboolean;
            case Json::kValueTypeInteger:
              return reinterpret_cast<NullAccess*>(context)->vinteger;
            case Json::kValueTypeUInteger:
              return reinterpret_cast<NullAccess*>(context)->vuinteger;
            case Json::kValueTypeReal:
              return reinterpret_cast<NullAccess*>(context)->vreal;
            case Json::kValueTypeString:
              return reinterpret_cast<NullAccess*>(context)->vstring;
            default:
              return (reinterpret_cast<NullAccess*>(context))->vnull;
            }

            return (reinterpret_cast<NullAccess*>(context))->vnull;
        }

    };
```

Character String Handling

The Json library handles character strings as `String` objects.

Although these objects provide `std::string` subset APIs, there is no compatibility in terms of type. Because of this, when a value is exchanged with `std::string`, the `c_str()` function must be used to replace the value via a string of the `const char*` type.

```
Json::String jsonstr = "sample string";
std::string stlstr = jsonstr.c_str();
. . .
jsonstr = stlstr.c_str();
```

3 Using the Library

Preparations

(1) Prepare a memory allocator

A memory allocator (`MemAllocator`) that has `allocate()` and `deallocate()` as in the following must be prepared when using this library.

- `allocate()`: Receives the size, allocates a memory area of that size and returns the beginning pointer (When no memory area can be allocated, NULL is returned)
- `deallocate()`: Releases the memory area allocated with `allocate()`

Internal to the library, `allocate()` is called when memory allocation is required and `deallocate()` is called when memory release is required.

The `allocate()` and `deallocate()` specifications have user-defined arguments added to the standard libc functions `malloc()` and `free()`, but the expected behavior is similar to `malloc()` and `free()`. Allocation requests must be fulfilled each time, but deallocation requests can also be optimized so that memory areas are freed together after library usage, for example.

When memory cannot be allocated with `allocate()` and NULL returns, newly call the `notifyError()` virtual function.

The base class is implemented so that this message is output as a standard error output; however, override `notifyError()` and implement appropriate processing in your application.

A memory allocator code example is as follows.

```
class MyAllocator : public Json::MemAllocator
{
public:
    MyAllocator(){};
    ~MyAllocator(){};
    virtual void *allocate(size_t size, void *user_data)
    {
        void *p = malloc(size);
        return p;
    }

    virtual void deallocate(void *ptr, void *user_data)
    {
        free(ptr);
    }
    virtual void notifyError (int32_t error, size_t size, void *userData )
    {
        switch(error){
        case SCE_JSON_ERROR_NOMEM:
            printf("allocate Fail. size = %ld\n", size);
            abort();
        default:
            printf("unknown[%#x]\n", error);
            break;
        }
    }
};
```

(2) Create an InitParameter object

Create an `InitParameter` object and set the memory allocator to it. Set the library behavior as required, such as the size of the buffer used internally.

```
MyAllocator allc;
Json::InitParameter initparam((Json::MemAllocator*)&allc,0,512);
```

(3) Generate/initialize the Initializer object

Generate an `Initializer` object and initialize it with the `InitParameter` object.

One `Initializer` object can be initialized at a time in a process. If an initialized `Initializer` object already exists and another `Initializer` object is attempted to be initialized, the `SCE_JSON_ERROR_MULTIPLEINIT` error will return. If re-initialization is required, delete all objects being used and then carry out initialization again.

```
Json::Initializer initializer;
ret = initializer.initialize(&initparam);
```

Parsing JSON Documents

The procedure for parsing a JSON document, obtaining the `Value` object tree, and referencing the content is as follows.

(1) Generate a root Value object

Generate a `Value` object to be the root. If it is to be retained during the utilization period, it can be generated either on a stack or on a heap.

```
Value rootval;
```

(2) Implement NullAccessFunction

Implement the behavior when a NULL `Value` object is accessed during JSON document parsing as a `Parser::NullAccessFunction` type callback function and set it to the root `Value` object.

```
NullAccess na;
rootval.setNullAccessCallBack(NullAccess::NullAccessCB, &na);
```

(3) Parse processing

The parse processing can be performed with the following three methods.

- Specifying the path of the JSON document file
- Batch parsing: Place the entire JSON document in memory and specify its address
- Partitioned parsing: Pass the JSON document to the parser 1 byte at a time, create a data provide callback function, and specify it

For any of these methods, specify the root `Value` object that will store the parse results as an argument. Refer to each of the following samples.

```
// parse from file
ret = Json::Parser::parse(rootval, "/hostapp/sample.json");

// batch parse
const char jsonstr[] =
   "[ 1, -1, 0.1, \"sample\", { \"name\":\"sample.json\" }, true]";
ret =  Json::Parser::parse(rootval, jsonstr, strlen(jsonstr));

// partitioned parse (refer to the "Partitioned Parsing Feature" section)
const char jsonstr[] =
    "[ 11, -11, 0.11, \"sample func\", { \"name\":\"sample.json\" }, false]";
UserDataProvider provider(jsonstr, strlen(jsonstr) );
ret = Json::Parser::parse(rootval, UserDataProvider::cbProvide, &provider);
```

(4) Refer to the parsing results

If the structure of a parsed JSON document is known in advance, by applying subscript operators to arrays and objects, it is possible to refer to content as if it were arrays and associated arrays.

For example, consider the parsing of a JSON document with the following structure:

```
[
  {
    "code":10001,
    "name":"Tarou Yamada"
  },
  {
    "code":20015,
    "name":"John Smith"
  },
  (omitted: similar objects)
]
```

In such cases, it is possible to read the entire document with code such as the following:

```
int cnt = rootval.count();
for (int i=0; i < cnt ; i++) {
    printf("(%d)code: %ld, name: %s\n",
              i,
              rootval[i]["code"].getUInteger(),
              rootval[i]["name"].getString().c_str()
    );
}
```

If the JSON document structure is not known, it is possible to read the content following the tree structure from the root `Value` object. For data other than arrays/objects, first determine the type as in the following, then obtain the values using a method according to the type.

```
switch(dspval.getType()){
case Json::kValueTypeInteger:
    printf( ":[Integer]%ld\n", dspval.getInteger() );
    break;
case Json::kValueTypeUInteger:
    printf( ":[UInteger]%ld\n", dspval.getUInteger() );
    break;
case Json::kValueTypeReal:
    printf( ":[Real]%f\n", dspval.getReal() );
    break;
default:
    printf( ":[toString]%s\n", dspval.toString().c_str() );
}
```

Generating JSON Documents

The procedure for formulating a new `Value` object tree structure and outputting it as a JSON document is as follows.

(1) Generate a root Value object

Generate a `Value` object to be the root. If it is to be retained during the utilization period, it can be generated either on a stack or on a heap.

```
Json::Value rootval;
```

(2) Formulate a tree structure

Include the data contained in the JSON document into respective `Value` objects, and formulate their tree structure. `set()` is used to include the data in the `Value` objects, but objects that contain data other than arrays or objects can be created with just constructors. For arrays, each element can be included by preparing an `Array` type variable and using an API such as `push_back()`. For objects, prepare an `Object` type variable, then a key can be substituted as a subscript. Refer to the following code.

```
Json::Array ary;
ary.push_back(Json::Value( true ));
ary.push_back(Json::Value( (uint64_t)1 ));
ary.push_back(Json::Value( (int64_t)-1 ));
ary.push_back(Json::Value( (double)0.1 ));
ary.push_back(Json::Value( Json::kValueTypeNull ) );
ary.push_back(Json::Value( Json::String("serialize sample") ));

Json::Object obj;
obj["version"] = (Json::String)"sample1.00";
ary.push_back(Json::Value( obj ));

rootval.set(ary);
```

(3) Output (serialize) the tree structure

Call `serialize()` for the root `Value` object and convert it to text data. In the following code example, a data receive callback function is set and the serialized results will be partitioned then output (refer to the "Partitioned Serializing Feature" section).

```
Json::String buf;
UserDataReceiver receiver("/hostapp/output.json");
int32_t ret = rootval.serialize(buf, UserDataReceiver::cbReceive, &receiver);
```

Post-processing

(1) Destroy objects other than Initializer

Objects other than `Initializer` generated by this library such as `Parser` and `Value` should be destroyed by calling a destructor before calling `terminate()`.

When processing for the Json library in one function completes, caution is required for destroying objects allocated in the stack. If `terminate()` is called before the objects are destroyed, memory leaks or segmentation faults may be caused, therefore code as follows:

```
void usefunc()
{
  Json::Initializer initializer;
  int32_t ret = initializer.initialize(&initparam);
  {
    // {} used because the rootval destructor will be called
    Json::Value rootval;
    ret = Parser::parse(rootval, "/hostapp/sample.json");
    ...
    // Explicitly calling the destructors of the used objects is possible
    // rootval.~Value();
  }
  initializer.terminate(); // Does not need to be explicitly called
}
```

(2) Destroy the Initializer object

After destroying all other objects generated with this library, finally destroy the `Initializer` object.

`terminate()` is called in destructors, therefore it does not need to be explicitly called.