Using The JSON C API

Superseded by:
https://realtimelogic.com/ba/doc/en/C/reference/html/md_en_C_md_JSON.html

Download source code: https://realtimelogic.com/products/json/

JSON has become a popular inter-process communication (IPC) data interchange format for a variety of computer languages. JSON is particularly easy to use in scripting languages since JSON maps directly to data structures in most scripting languages, including JavaScript and Lua. However, more work is involved when using JSON from compiled languages that does not support introspection, such as C/C++.

The Barracuda JSON serializer and parser (deserializer) provides a number of methods that can be used by C/C++ code when designing a system that uses JSON for IPC. This tutorial gives an introduction to this API and how one can use the API to serialize and de-serialize C data structures.

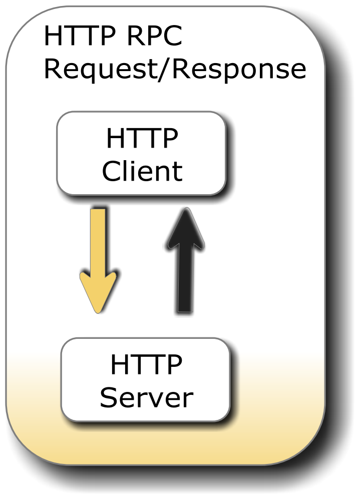
The following examples use the C API's. The C++ API is somewhat easier to use. The JSON parser is designed in an object oriented way and the header files contain C++ class wrappers for the C++.

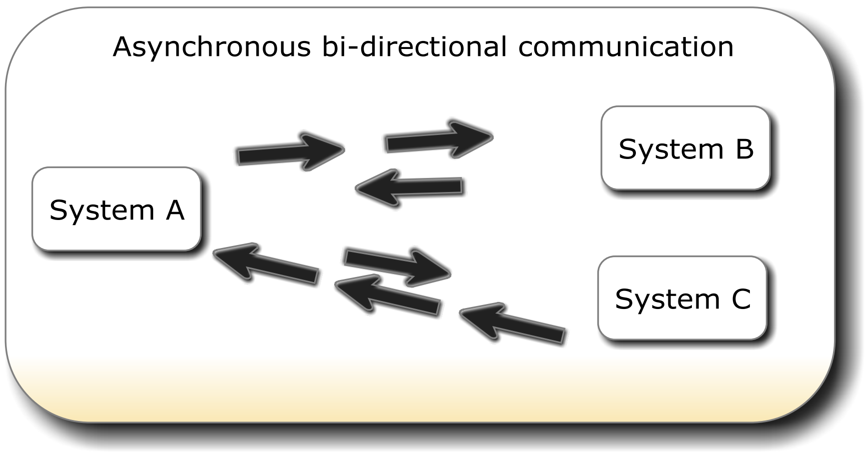
Keep it simple

The JSON data interchange format maps directly to objects in Scripting languages. Data structures are represented as tables in, for example, Lua and JavaScript and can be nested indefinitely. One can also represent JSON as C structures, but the complexity involved in working with this data in C code grows as the data structures become more complex. We suggest that you avoid nesting data structures when communicating with C code. In other words, keep the data structures simple so you can directly map JSON to one C structure. The C API provided by the JSON parser and serializer provides many functions, but only a few are needed when using simple data structures. In particular, this tutorial focuses on two methods that resembles function printf (serialize) and function scanf (de-serialize).

JSON Use Cases

Many consider JSON a format that can only be used together with HTTP. A HTTP message is a Remote Procedure Call with a request and a response. The reason for this limitation is related to the typical JSON serializer and parser. However, the Barracuda JSON library is designed such that the serializer and the parser can be used for asynchronous bi-directional communication between multiple distributed systems. Any type of communication channel can be used, such as raw TCP and serial communication.





In addition, the Barracuda Server and the Barracuda HTTPS client library enable a HTTPS request to morph into a secure raw TCP connection that can be used for asynchronous bi-directional communication. A client can initiate an asynchronous communication channel by requesting a HTTPS request that bypasses firewalls and proxies. The channel can then be converted to raw TCP suitable for asynchronous bi-directional communication.

Serializing JSON data

The serializer requires a few objects: an output buffer and an error container.

The error container is constructed as follows:

```
JErr err;
JErr_constructor(&err);
```

The output buffer must be based on the Barracuda BufPrint class. This class can be used "as is" or you can use a derived class such as the DynBuffer (Dynamic grow Buffer).

The following example shows how to directly use the BufPrint class.

```
static int
BufPrint_sockWrite(BufPrint* o, int sizeRequired)
{
 // Write data to socket or whatever method that is used for
 // transmitting the serialized data.
 // Send: o->buf with length o->cursor
 o->cursor=0; /* Data flushed */
}


/* Initialize a basic BufPrint that writes data to socket */
char outBuffer[256];
BufPrint out; /* Buffer needed by JSerializer */
BufPrint_constructor(&out, NULL, BufPrint_sockWrite);
out.buf = outBuffer;
out.bufSize = sizeof(outBuffer);
```

We can create the JSON serializer when we have created a JErr object and a BufPrint object.

```
JSerializer_constructor(&js, &err, &out); // out is the BufPrint obj
```

Any type of C data structure can be serialized, but we mentioned above that it is better to keep it simple in C code.

Assume we have the following C structure:

```
#define ALARM1 4
typedef struct
{
      int signo;
      char* msg;
      int level;
} Alarm1;
```

The signo and the associated signal number definition above is a convenient method for sending message numbers when using a full duplex channel such as raw sockets. The signal number is not needed if a message is sent over HTTP since you would typically embed the message type in the HTTP header or URL.

```
Alarm1 a1;
// Initialize a1
JSerializer_set(s,"{dsd}",
                "signo",a1.signo,
                "msg",a1.msg,
                "level",a1.level);
JSerializer_commit(s); /* Commit JSON data i.e. send to peer. */
```

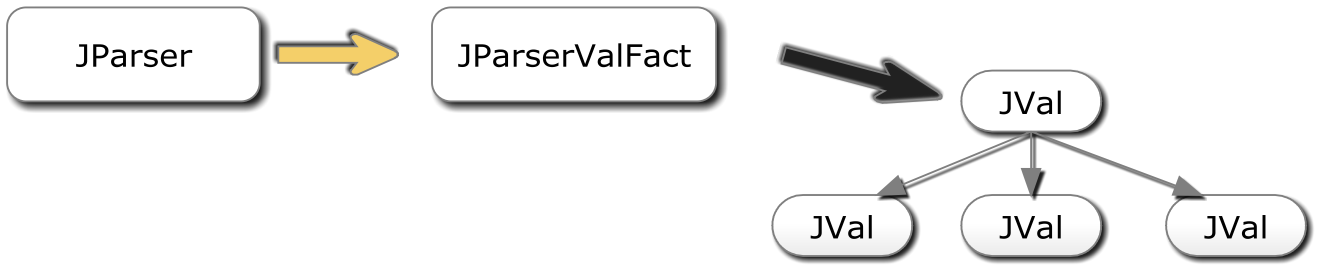
The above code snippet uses JSerializer_set(), which is a function similar to printf in that it takes formatting parameters as argument. The above formatting "{dsd}" instructs the serializer to emit a JSON table with an integer(d), a string(s), and integer(d). The curly brackets indicate start and end of the table. You can nest structures also, such as "{dsd{d}}". See the JSON documentation for more information on the format flags. The JSerializer_set() function is under the hood parsing the format string and using the lower level JSerializer_XX functions. For example, the formatting "{dsd}" instructs the function to do the following in sequence on the Serializer object: beginObject(),setInt(),setString(),setInt(),endObject().

If you are using the JSON serializer in a fully bi-directional, asynchronous design, a new object can be sent as soon as JSerializer_commit() is called. If you are using the JSON serializer with HTTP, you would typically destroy the JSerializer object at this point by calling the destructor.

Deserializing JSON data

De-serializing JSON encoded data is managed by the JSON parser. The Barracuda JSON parser is capable of serializing data on a data stream such as raw TCP. The parser gives the calling code information on the parsing state and indicates when a complete JSON object is parsed. The state functionality is only needed when using JSON for bi-directional asynchronous message transfer. When using JSON with HTTP, you use the parser to parse one object only. HTTP is a remote procedure based protocol, thus a complete JSON object will be in the request and response.

The Barracuda JSON parser parses JSON and feeds the parsed JSON to an JParserIntf instance. The JParserIntf is an abstract class and cannot be used directly. We provide one implementation for this class, the JParserValFact. The JParserValFact builds a tree structure of JVal objects. A JVal instance represents one JSON value.



When the parser has completed parsing one JSON object, the JParserValFact contains the tree structure of the parsed JSON object. The top JVal node can be fetched from the JParserValFact container and the JSON data can be extracted by iterating the JVal tree. A function similar to scanf simplifies extraction of JSON data structures that closely resembles C structures. We will focus on using the JVal_get() function for extracting data by providing a format string and arguments that copies the JSON data directly to a C structure.

Create a parser and the support objects as follows:

```
JParser p; /* JSON parser */
JParserValFact pv; /* JSON parser value factory. Used by the parser */
JParserValFact_constructor(&pv, 0, 0);
/* Cast JParserValFact to base class JParserIntf */
JParser_constructor(&p, (JParserIntf*)&pv, 0);
```

When using the parser with bi-directional asynchronous message transfer, a forever loop is typically created that reads data from the stream as it trickles in. This data is fed to the parser, and the parser gives us state information on the parse state such as: need more data, parsing failed, and a complete object is parsed.

```
int status=0;
for(;;) // forever
{
   int status,n;
   char* buf[1024];
   JErr err;
   JErr_constructor(&err);
   if(status==0) /* if not in "parsed complete object" state (R1) */
     n = socketread(buf,sizeof(buf)); // Read from any type of data channel
   if(n <0 ) break; // Stop: No more data or data channel err.
   status = JParser_parse(&p, buf, n);
   if(status < 0)
      break; // Stop: Parse error.
   else if(status > 0)
   { // A Complete JSON object
      int signo;
      JVal* v;
      v=JParserValFact_getFirstVal(&vp); // Top tree node

      // Extract the signal number that is common for all structures.
      // Typically used with bi-directional asynchronous message transfer,
      // but not with JSON over HTTP.
      JVal_get(v, &err, "{d}", "signo", &signo);

      if(signo == ALARM1)
      {
         Alarm1 a1;
         // Extract data from JSON tree
         JVal_get(v, &err,"{dsd}",
                  "signo",&a1.signo,
                  "msg",&a1.msg,
                  "level",&a1.level);
         if(JErr_noError(&e))
         {
            // Use data
         }
      }

      /* Delete the JSON value tree, thus making the
       * JParserValFact ready for the next object in the stream. */
      JParserValFact_termFirstVal(&pv);
   }
   // Else: need more data. Continue loop.
}
```

When using the parser with HTTP, you typically read the complete HTTP input, create a parser, parse all data at once, and then terminate the parser. The parser can also be used when receiving HTTP chunk encoded data with unknown length. You would use the parser similar to above; i.e. parse data as it streams in on a HTTP chunk encoded stream.

Notice how we skip reading from the data stream if we parsed a complete object (**R1**). The reason for this construction is that a chunk read from the stream may contain parts of a new JSON object or contain multiple objects. We must call the parser again, without reading more data since the parser is at this point still using our input buffer.

In the above code, the JSON data is extracted from the JSON tree by the following code:

```
JVal_get(v, &err,"{dsd}",
         "signo",&a1.signo,
         "msg",&a1.msg,
         "level",&a1.level);
```

The function is the opposite of JSerializer_set(). The function is similar to scanf. A format flag specifies the data structure to extract from the JSON tree. The function is under the hood parsing the format string and using the lower level JVal_XX functions. For example, the formatting "{dsd}" instructs the function to do the following in sequence on the JVal tree:

1. getObject(): The curly bracket { instructs JVal_get to go one level down in the JVal tree.
2. getInt(): Copy data into the pointer to integer &a1.signo
3. v=JVal_getNextElem(v): Advance to next element
4. getString():Copy data into the pointer to string &a1.msg
5. v=JVal_getNextElem(v): Advance to next element
6. getInt():Copy data into the pointer to integer &a1.level