jWrite - a really simple JSON writer in C

tonywilk, 18 Mar 2015 CPOL

5.00 (1 vote)

Rate: vote 1vote 2vote 3vote 4vote 5

A simple set of functions to write C variables out to JSON with no fuss or errors

- **Download jWrite_1v2-noexe.zip - 18.1 KB**
- **Download jWrite_1v2.zip - 18.1 KB**

Introduction

jWrite is a simple way of writing JSON to a char buffer in C, directly from native variables. It manages the output buffer so you don't overrun, it handles all the fiddly quotes, brackets and commas and reports where you have tried to create invalid JSON.

You can, of course, write json to a string with sprintf()... but this is miles better.

Background

This is a companion set of functions to "jRead an in-place JSON element reader" (http://www.codeproject.com/Articles/885389/jRead-an-in-place-JSON-element-reader).

The same basic design principles apply: it should be in straight C, have little or no memory overhead, execute fast and be simple to use. It is intended for use in embedded projects where using some nice and big C++ structured solution is just not appropriate.

Several approaches were considered; the 'most automatic' was to define a structure which described the JSON and contained pointers to external variables to get the data - this seemed a good idea since it would then be a single call to 'stringify' everything... the downside was the complexity of API required to define such a structure and keep it all in memory.

For the programmer, it is pointless to make configuration of a JSON writer more complicated than writing the stuff out by hand!

jWrite attempts a happy medium in that it is simple and doesn't *seem* to do much - you just tell it what to write and it does it.

Using the code

Jumping straight in:

Hide Copy Code

```
jwOpen( buffer, buflen, JW_OBJECT, JW_PRETTY );  // open root node as object
jwObj_string( "key", "value" );                   // writes "key":"value"
jwObj_int( "int", 1 );                           // writes "int":1
jwObj_array( "anArray");                          // start "anArray": [...]
    jwArr_int( 0 );                              // add a few integers to the array
    jwArr_int( 1 );
    jwArr_int( 2 );
jwEnd();                                        // end the array
err= jwClose();                                  // close root object - done
```

which results in:

Hide Copy Code

```
{
    "key": "value",
    "int": 1,
    "anArray": [
        0,
        1,
        2
    ]
}
```

The output is prettyfied (it's an option) and has all the `{ } [ ] , : "` characters in the right place.

Although this looks very straightforward, not a lot different than a load of sprintf()'s you may say, but jWrite does a few really useful things: it helps you make the output *valid JSON*.

You can easily call a sequence of jWrite functions which are invalid, like:

Hide Copy Code

```
jwOpen( buffer, buflen, JW_OBJECT, JW_PRETTY );  // open root node as object
```

```
jwObj_string( "key", "value" );                 // writes "key":"value"
jwObj_int( "int", 1 );                          // writes "int":1
    jwArr_int( 0 );                             // add a few integers to the array
...
```

Since the JSON root is an object, we must insert `"key":"value"` pairs, so the call to `jwArr_int( 0 )` is not valid at this point... this sets an internal error flag to "*tried to write Array value into Object*" and *ignores subsequent function calls* until the ending `jwClose()` when it reports the error.

When writing a large JSON file it may be difficult to figure out where you went wrong... in this case jWrite helps out by giving you the number of the function which caused the error and leaving the partially-constructed JSON in your buffer (with '\0' termianator). In the above case `jwErrorPos()` would return `4` because the 4th function in this sequence caused the error (the `jwOpen()` call is number `1`)

Any valid JSON sequence can be created with value types of **Object, Array, int, double, bool, null and string**. You can also add your own stringified values by inserting them raw e.g. `jwObj_raw( "key", rawtext )`.

There are longer examples in `main.c` and some more info in `jWrite.h`

Points of Interest

The internal control structure

Internally the jWrite functions keep a stack of the Object/Array depth and at every call checks if that would result in an error or not. An almost minor point is that it manages your output buffer - once you pass a buffer and length to jwOpen() it will not overrun it (it will return you an "output buffer full" error) and it will keep it '\0' terminated.

You may have realised that these functions must store some state information (and the node stack) somewhere...

...yes, there is a `struct jWriteControl` which keeps track of the internals and is used by all of the functions.

Some of you may say "*Oh, ok, fine*" and others may say "*Wait a minute... that's not GLOBAL is it ?*"

Well, yes and no...

To be, or not to be, GLOBAL

For many applications it is a lot simpler to have one global (static) instance of a structure which can be used for jWrite, it makes the API calls easy to type in - you don't have to supply a reference every time.

However, that is not very flexible and does not allow for multiple uses of jWrite functions at the same time, so jWrite allows you to turn off the global by undefining `JW_GLOBAL_CONTROL_STRUCT`. This causes all the API functions to require a pointer to an application-supplied instance of struct jWriteControl.

The above example with `#define JW_GLOBAL_CONTROL_STRUCT` commented out looks like:

Hide Copy Code

```
struct jWriteControl jwc;
jwOpen( &jwc, buffer, buflen, JW_OBJECT, JW_PRETTY );  // open root node as object
jwObj_string( &jwc, "key", "value" );                  // writes "key":"value"
jwObj_int( &jwc, "int", 1 );                           // writes "int":1
jwObj_array( &jwc, "anArray");                         // start "anArray": [...]
    jwArr_int( &jwc, 0 );                             // add a few integers to the array
    jwArr_int( &jwc, 1 );
    jwArr_int( &jwc, 2 );
jwEnd( &jwc );                                        // end the array
err= jwClose( &jwc );                                  // close root object - done
```

Which is a lot more to type in and begs to be a C++ class really.

Conclusion

The jWrite and jRead are simple to use and make handling JSON in C manageable without overhead nor a complicated API to learn, especially in embedded projects where you still need to be careful of memory and processor usage.

The download jWrite_1v2.zip contains the source for jWrite.c/jWrite.h and a Windows commandline main.c which runs a couple of examples. Compiled in VS2010 with the included project files.

jWrite is written in C86 and has no dependancies.

License

About the Author

tonywilk

Founder

United Kingdom

Developer in just about anything from 6502 machine code thru C, C++ and now things like PHP and javascript. Used to develop hardware and still dabble with electronics and ham radio when I'm not letting off pyrotechnics, shooting or flying my VPM M16 gyroplane.