

GitHub

This repository ▾

Search or type a command 🔍

Explore

Features

Enterprise

Blog

Sign up

Sign in

.IC  interactive-matter / **aJson**

★ Star272

🍴 Fork56

aJson is an Arduino library to enable JSON processing with Arduino. It easily enables you to decode, create, manipulate and encode JSON directly from and to data structures.  
<http://interactive-matter.org/2010/08/ajson-handle-json-with-arduino/>

📦 144 commits


🌿 2 branches


📦 8 releases

👤 7 contributors

 branch: master ▾

**aJson** / 

Merge pull request #42 from sudar/bugfix ...		
👤 <b>interactive-matter</b> authored 6 months ago <span>latest commit d8d517c3f4 </span>		
📁 Examples	Convert the aJson library to use Wiring-style streams	a year ago
📁 utility	Convert the aJson library to use Wiring-style streams	a year ago
📄 .gitignore	updating gitignor	3 years ago
📄 README.md	Update README.md	11 months ago
📄 aJSON.cpp	Added '/' to the list of characters that are escaped. Fixes #41	6 months ago
📄 aJSON.h	Make aJson parser more robust to come with malformed input	9 months ago
📄 keywords.txt	added more classifiers	11 months ago

 **README.md**

# aJson v1.0

Copyright (c) 2010, Interactive Matter, Marcus Nowotny

Based on the cJSON Library, Copyright (C) 2009 Dave Gamble

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Welcome to aJson.

<> Code

🔔 Issues23


🔗 Pull Requests3

📊 Pulse


📈 Graphs


🌐 Network

HTTPS clone URL

https://github.com 

You can clone with HTTPS, or Subversion. ©

 Clone in Desktop

 Download ZIP

https://github.com/interactive-matter/aJson

1/5

aJson is the attempt to port a complete JSON implementation to Arduino. It is based on the cJSON implementation, reduced in size and removing one or two features:

- The code has very limited support on ATmega168 - there is just not enough memory and memory fragmentation is a serious problem
- Arrays and Lists are max 255 elements big
- There is no proper Unicode handling in this code
- There is an internal buffer eating up 256 bytes of ram

Most of the limitation will be gone in one of the future releases.

JSON is described best here: <http://www.json.org/> It's like XML, but fat-free. You use it to move data around, store things, or just generally represent your program's state. JSON is especially useful to exchange data efficiently with e.g. JavaScript, Java, C++, Processing or anything else

aJson is a library to receive, understand, create or modify JSON strings directly in the Arduino. JSON is quite a standard, so that is perfect for exchanging data with other applications. In combination with HTTP it is suitable to implement REST Web Services.

aJson provides functions to parse JSON strings to object models. Handle, search and create and modify JSON Object structures.

This is some JSON from this page: <http://www.json.org/fatfree.html>

```
{
  "name": "Jack (\\"Bee\\" Nimble",
  "format": {
    "type": "rect",
    "width": 1920,
    "height": 1080,
    "interlace": false,
    "frame rate": 24
  }
}
```

## Parsing JSON

To parse such a structure with aJson you simply convert it to a object tree:

```
aJsonObject* jsonObject = aJson.parse(json_string);
```

(assuming you got the JSON string in the variable json\_string - as a char\*)

This is an object. We're in C. We don't have objects. But we do have structs. Therefore the objects are translated into structs, with all the drawbacks it brings.

Now we can e.g. retrieve the value for name:

```
aJsonObject* name = aJson.getObjectItem(root, "name");
```

The value of name can be retrieved via:

```
Serial.println(name->valuestring);
```

Note that the aJsonObject has a union which holds all possible value types as overlays - you can get only useful data for the type which you have at hand. You can get the type as

```
name->type
```

which can be either `aJson_False`, `aJson_True`, `aJson_NULL`, `aJson_Number`, `aJson_String`, `aJson_Array` or `aJson_Object`. For `aJson_Number` you can use `value.number.valueint` or `value.number.valuedouble`, for `aJson_String` you can use `value.valuestring`, for `True` or `False`, you can use `value.valuebool`.

To render the object back to a string you can simply call

```
char *json_String=aJson.print(jsonObject);
```

Finished? Delete the root (this takes care of everything else).

```
aJson.deleteItem(root);
```

This deletes the objects and all values referenced by it.

## Parsing streams

As you can see this will eat up lots of memory. Storing the original string and the JSON object is a bit too much for your Arduino - it will most likely use up all the memory. Therefore it is better to parse streams instead of strings. A stream in C is a `FILE*` - on Arduino there are some special streams, but later adapters will be provided. So if you for example read from a `FILE*` stream you can simply call

```
aJsonObject* jsonObject = aJson.parse(file);
```

By that you will not have to store the JSON string in memory.

## Filtering while parsing

Any JSON respond can have object name/value pairs your code either does not understand or is not interested in. To avoid those values to go into your memory you can simply add filters to your parsing request. A set of filter is just a list of names you are interested in, ended by a null value. If you are only interested in "name", "format", "height" and "width" in the above example you can do it like:

```
char** jsonFilter = {"name","format","height","width",NULL};
aJsonObject* jsonObject = aJson.parse(json_string,json_filter);
```

(assuming you got the JSON string in the variable `json_string` - as a `char*`)

By that only the following structure is parsed - the rest will be ignored:

```
{
  "name": "Jack (\\"Bee\\" ) Nimble",
  "format": {
    "width": 1920,
    "height": 1080,
  }
}
```

It is good practice to always use the filtering feature to parse JSON answers, to avoid unknown objects swamping your memory.

## Creating JSON Objects from code

If you want to see how you'd build this struct in code?

```
aJsonObject *root,*fmt;
root=aJson.createObject();
```

```

aJson.addItemToObject(root, "name", aJson.createItem("Jack (\\"Bee\\" Nimble));
aJson.addItemToObject(root, "format", fmt = aJson.createObject());
aJson.addStringToObject(fmt, "type", "rect");
aJson.addNumberToObject(fmt, "width", 1920);
aJson.addNumberToObject(fmt, "height", 1080);
aJson.addFalseToObject (fmt, "interlace");
aJson.addNumberToObject(fmt, "frame rate", 24);

```

The root object has: Object Type and a Child The Child has name "name", with value "Jack ("Bee") Nimble", and a sibling: Sibling has type Object, name "format", and a child. That child has type String, name "type", value "rect", and a sibling: Sibling has type Number, name "width", value 1920, and a sibling: Sibling has type Number, name "height", value 1080, and a sibling: Sibling has type False, name "interlace", and a sibling: Sibling has type Number, name "frame rate", value 24

If you want to create an array it works nearly the same way:

```

aJsonObject* root = aJson.createArray();

aJsonObject* day;
day=aJson.createItem("Monday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Tuesday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Wednesday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Thursday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Friday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Saturday");
aJson.addItemToArray(root, day);
day=aJson.createItem("Sunday");
aJson.addItemToArray(root, day);

```

The whole library (nicely provided by cJSON) is optimized for easy usage. You can create and modify the object as easy as possible.

## aJson Data Structures

aJson stores JSON objects in struct objects:

```

// The aJson structure:
typedef struct aJsonObject {
    char *name; // The item's name string, if this item is the child of, or is in the list of subitems of an object.
    struct aJsonObject *next, *prev; // next/prev allow you to walk array/object chains. Alternatively, use GetArraySize/GetArrayItem/GetObjectItem
    struct aJsonObject *child; // An array or object item will have a child pointer pointing to a chain of the items in the array/object.

    char type; // The type of the item, as above.

    union {
        char *valuestring; // The item's string, if type==aJson_String
        char valuebool; //the item's value for true & false
        int valueint; // The item's number, if type==aJson_Number
        float valuefloat; // The item's number, if type==aJson_Number
    };
} aJsonObject;

```

By default all values are 0 unless set by virtue of being meaningful.

Note that the aJsonObject has a union 'value' which holds all possible value types as overlays - you can get only

useful data for the type which you have at hand. You can get the type as

```
name->type
```

which can be either `aJson_False`, `aJson_True`, `aJson_NULL`, `aJson_Number`, `aJson_String`, `aJson_Array` or `aJson_Object`. For `aJson_Number` you can use `value.number.valueint` or `value.number.valuedouble`. If you're expecting an int, read `valueint`, if not read `valuedouble`. For `aJson_String` you can use `value.valuestring`, for `True` or `False`, you can use `value.valuebool`.

`next/prev` is a doubly linked list of siblings. `next` takes you to your sibling, `prev` takes you back from your sibling to you. Only objects and arrays have a "child", and it's the head of the doubly linked list. A "child" entry will have `prev=0`, but `next` potentially points on. The last sibling has `next=0`. The type expresses `Null/True/False/Number/String/Array/Object`, all of which are `#defined` in `aJson.h`

Any entry which is in the linked list which is the child of an object will have a "string" which is the "name" of the entry. When I said "name" in the above example, that's "string". "string" is the JSON name for the 'variable name' if you will.

Now you can trivially walk the lists, recursively, and parse as you please. You can invoke `aJson.parse` to get `aJson` to parse for you, and then you can take the root object, and traverse the structure (which is, formally, an N-tree), and tokenise as you please.

## Lists in aJson

Lists are easily handled in `aJson`, to create a list you can simply use the provided API functions:

```
aJson.create<TYPE>Array(objects,24);
```

You simply pass a array of the respective type: `char*`, `int*` and so on.

`aJSON` doesn't make any assumptions about what order you create things in. You can attach the objects, as above, and later add children to each of those objects with

```
aJson.addItemToArray()
```

or remove them with

```
aJson.deleteItemFromArray() - which also deletes the objects, or  
aJson.detachItemFromArray() - which does not free the memory
```

As soon as you call `aJson.print()`, it renders the structure to text.

Have Fun!

