Lesson 4-2 JSON Parser

In this video I'll talk about JSON and how WICED makes it easy for you to read data from a JSON document.

First, what is JSON? It stands for JavaScript Object Notation. It's an open-standard format that uses human-readable text to transmit data. It is by far the most commonly used format for exchanging data in the cloud so it's very important for you to know how to use it.

JSON supports double precision floating point values, strings, booleans, arrays, and key/value pairs. As an example, the following JSON document shows information about me.

First is my name which is a string, then whether or not I am an IoT expert (which I am!), which is a boolean set to true. Then I have an array with the names of my two children, Anna and Nicholas. Finally, I have my address which contains a number that's represented as a floating-point value, and the rest of the address is represented as strings.

Note that the carriage returns and spaces (except for within the strings themselves) don't matter. JSON is space agnostic.

There are 2 JSON parsers built into WICED – cJSON and JSON_parser.

The first parser, cJSON, is an open source project that was created by a very clever programmer named Dave Gamble. He described "cJSON aims to be the dumbest possible parser that you can get your job done with". It's a single file of C and a single header file. The bottom line is that it is a super lightweight and really easy to use parser. Don't be confused – he called it dumb but it's not dumb – it's a really excellent parser.

cJSON reads the entire document at once and then lets you access the data with various API functions. This is the simpler of the two parsers and it's the one that I will focus on here.

JSON_parser does not read the entire document at once. It's more complicated to use than cJSON but it's useful in situations where you might have a very large document that can't practically be read all at once. I find that to be a pretty unlikely case in an IoT application.

In this case I'm are going to use JSON to specify the LEDs that I want to turn ON and OFF on our shield board. Two of the LEDs are connected directly to GPIOs while 4 of the LEDs are controlled by writing over the I2C bus to the PSoC on the shield.

We will use one keymap for the I2C LEDs and one keymap for the GPIO LEDs. In this example, we'll turn ON 4 LEDs and we'll leave 2 of them off.

You can look at the README file inside of libraries/utilities/cJSON/README for a description of the APIs and some other useful examples. You can find all of the original source code on GitHub at https://github.com/davegamble.

We'll start with a new project called 04/04_cjson. Remember that I keep copying all of the chapter 4 projects into the 04 directory.

To use the library, we need to add one line to the make file as shown here that will include the library functions that I'm using:

```
$(NAME)_COMPONENTS := utilities/cJSON
```

In the C file, we need to include cJSON.h:

```
#include <cJSON.h>
```

This will bring all of the APIs into our project so that I can use them.

Next, we will add a constant char array to hold the JSON data. Normally you'd be receiving this data from the cloud, but for now I will just hard-code the JSON that I'm using into my project.

Note that we need to escape the quotes within the JSON with a backslash since we don't want the C compiler to interpret them for us. Unfortunately, that makes it a little bit messy to look at, but that's the way it is.

In the main application, we write to the I2C to allow us to control the LEDs on the shield, then we read the JSON data. Once we have the JSON data read in, there are functions that parse out the different sections and the the individual entries for the GPIO controlled LEDs and the I2C controlled LEDs. Finally, the corresponding LEDs are turned ON or OFF based on the requested states from the original JSON document.

The key library functions are:

cJSON_Parse which reads in the file, or in our chase the character string, and **cJSON_GetObjectItem** which traverses through the JSON hierarchy one level at a time to get the value that you are looking for. The structures used by these functions are pointers to structures of type cJSON.

These structures contain pointers to the next item and the previous item, the type of data, and finally the data itself.

Now let me program the project and see what it does. Look – 2 of the I2C LEDs turn ON and both of the GPIO LEDs turn ON. That's just what we expected because we wrote that into the JSON string.

If you want to, you can try changing the JSON yourself to turn ON different combinations of the LEDs.

If you don’t have the shield you can still run this project, but it will only control the 2 LEDs on the baseboard. Note that one of the LEDs on the baseboard is active high while the other is active low so the JSON above will turn ON one LED and turn OFF the other one.

As always, you can post your comments and questions in our Wi-Fi developer community or you are welcome to email me at alan_hawse@cypress.com or tweet me at @askioexpert. Thank you!