Hello YouTubers, here is the guy with the Swiss accent. Every ESP chip has a built-in SSD disk drive. In this episode, you will learn how to use it. As an example, I store some configuration parameters on this drive. So, you will also learn how to deal with json files. Both skills can be used for many other occasions.

Let’s get started.

You discover this SSD drive on your ESP as soon as you select your board. Here, you can choose from different selections. If you select the flash size, you get some options. For a nodeMCU board, for example, you get the two selections 4M/1MSPIFFS or 4M/3M SPIFFS. First, what does SPIFFS mean? It means “SPI Flash File System” and it was created for small flash memories (which are often connected by the SPI bus). Guys who watched my video # 34 know, that each ESP board has such a flash memory. The first modules had 512kB, and the newest Wemos Pro has 16 Mbytes of such memory. Most of the current ESP-12, NodeMCU and Wemos boards have 4MB flash size. This is the first number in your selection. If you would choose a “Generic ESP8266 board, you would get more choices, for example the choice for our Sonoff boards: 1M/64k SPIFFS. This can be used to program modules with smaller Flash memories.

So, the meaning of the first number is clear. What is the meaning of the second? It is the part of this flash memory which is reserved for your SSD drive. And because our ESPs do not have the resources of a full-blown computer, this drive only has limited functionality. For example, the drive has no formal directory structure. But it is sufficient for a lot of fun.

Fortunately, the Arduino IDE for ESP8266 has a library which can deal with this disk drive. It can store and read files, just as you are used from a normal drive.

Today, I want to store a file with our configuration parameters on this drive and read it by a sketch. This is quite handy, because you can also upload the content of this disk drive independently from the sketch. So, you can change your configuration without changing your sketch. Cool. And in the next version you will even be able to manage and upload this disk over the air from IOTappstory.com. Even cooler.

So, let’s start with an example. The library comes with the boards package. So, you should find the example file if you select “examples” “ESP8266” and “ConfigFile”. If you open the example, you see, that it uses two libraries: FS.h is used to deal with our disk drive and Arduino.json will be used to deal with the json files and our constants.

So, let’s look for one moment at the expression “JSON”. Another abbreviation. If we look it up, we see, that it means “JavaScript Object Notation”. Fortunately, it has nothing to do with java script. This just where it came from.

JSON is used to store and transport simple data structures. It is human readable and can be created and edited by a normal text editor.

Let’s assume, our sketch needs two constants, serverName and accessToken. In the old days, we would have defined these two constants like that. With json, they would look like that:

Very simple. Of course, this text could be extended if you need. Here you see an example from my SonoffSC sketch. I found a nice online editor where you can write and test the syntax of your json texts. You find the link in the description.

The ArduinoJson library helps you to decode or to encode such texts without any hassle.

Let’s summarize what we have now to deal with our constants:

1. Each ESP8266 is equipped with a small SSD disk, whether we use it, or not
2. The FS.h library can read and write this disk
3. We found a simple and standardized format to store our constants. By the way: This format is also used to transfer data between applications on the internet and is very common.
4. The json files can be created and read by humans
5. The disk can be created by our sketch, but we can also upload it manually, or even over the air
6. We found an example file to play with these things

For all of you who think now: This cannot be so simple. Of course, you are right. The guys from ESP development want to separate the cowards from the heros. And this is, why they did not include the function to upload the content of the disk right into the distribution. No, the brave ones have to install it now. You have to copy the file esp8266fs.jar to a sub-directory of the Arduino directory. On my Windows10 PC I find the “tools” directory on drive C and in the program files (X86)/ Arduino. In this tools directory you have to create the two directories ESP8266FS and tool and copy the jar file in this newly created directory. Not a big deal and you find the link in the description. As soon as you restart your Arduino IDE, you find a new topic in “tools”.

Now we are ready to rumble. Save your example file to a directory and press this newly created “esp8266 sketch data upload” button and you are asked to create a directory. After pressing “yes”, you find a new “data” directory in your sketch folder. All files written into this directory will be transferred to the disk by this newly created button. Just close the serial monitor before that, otherwise you get an error message. So, lets now create our first json configuration file. Its name is config.json and we can open it with our editor. We copy the newly created content into this file and save it. If we press now the button “esp8266 sketch data upload”, we see, that the IDE uploads a file to the ESP, just as it usually does with the sketch. Make sure, that your ESP is in flashing mode. Now, the disk drive on your ESP contains our json file with the configuration. This data survives also, if you power your ESP off, just as your sketch. It resides the at the end of your flash memory. The Sketch resides at the beginning of this memory. So, they never interfere. You can change the sketch or the disk independently.

But how we can read the disk and use the constant stored in our config file? As usual, we have to begin the SPIFFS with the command SPIFFS.begin(). Further down in the setup, we find two functions : Save Config” and “loadConfig”. Because we already loaded our configuration to the disk, we comment the “saveConfig”.and add yield command into the loop. Otherwise, the ESP will crash.

Let’s have now a closer look at the loadConfig. Here, we open the config.json file in “read” mode and check, if the file size is not bigger that 1024 bytes. Then, we create a buffer and read the content of the file into this buffer. Just to check, I print its content. Next, with the help of the json library, we decode and store the content of the file into our variables. Make sure, that the StaticJsonBuffer is bigger than your file size.

After parsing the file, we can assign its content to our variables. Here, it is extremely important, that we use exactly the same names as in our file. These names are case sensitive!

This feature can be used for trick: If we extend our structure, or even mix the sequence, no problem. The json parser deals with that. So, you could have much longer config files and your sketch only reads the fields it needs. Great!

Just two small tips: If you change the configuration of your board from let’s say 1M SPIFFS to 4M, you have to upload the sketch and the disk, because the disk starts at a different location. And if you define 4M SPIFFS, the upload takes 4 times longer, because the whole content of the disk has to be uploaded, even if our file is only a few bytes long. By the way, this is, why I usually use 921600 upload speed. I am too old to wait…

Summarized,

1. We updated our Arduino IDE to deal with SPIFFS
2. We created a little configuration file in the famous json format
3. We uploaded it to the disk on the ESP
4. We read this file with our sketch
5. And parsed the json file to find the content of our configuration variables
6. At the end, we even extended and mixed the sequence of our file and we still got the same results.

As a “goodie”, I uncomment the saveConfig now and run the sketch again. So, we see, we could also change the file from within our sketch. But of course, this changes only the file in the ESP, not on our disk…

I hope, this video was useful or at least interesting for you. If true then like. Bye