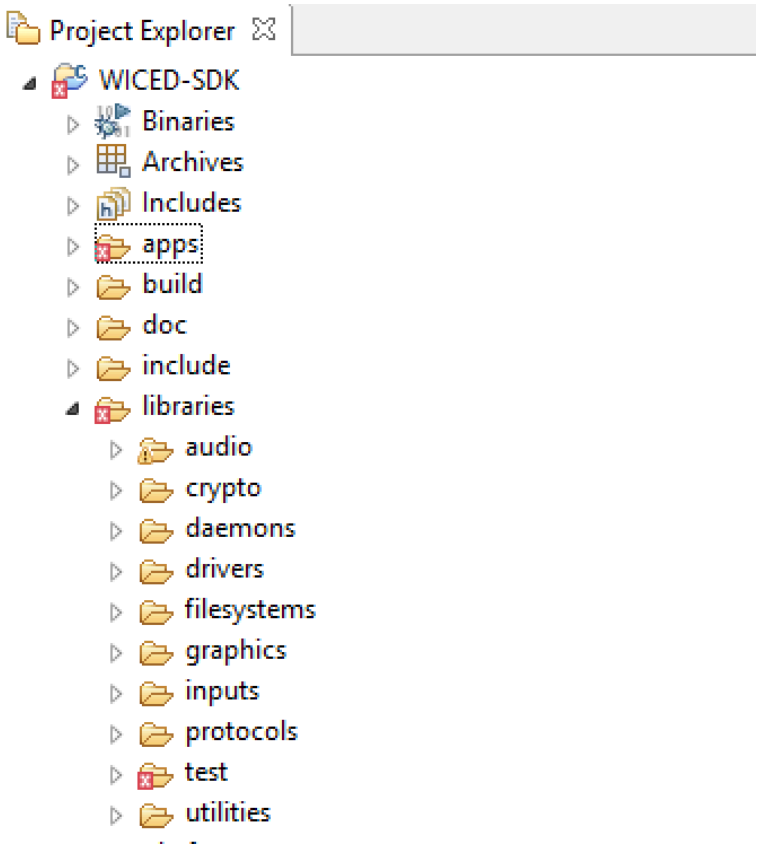
Chapter 4: Using the WICED-SDK Library

Time 1 Hour

At the end of this chapter you should understand what is contained in the WICED-SDK library. In addition, you will understand the Java Script Object Notation file format (which is widely used on the Internet)

4.1 WICED-SDK Library

At this point life is too short to develop all the “stuff” that you might want to include in your IoT project. In order to accelerate your development cycle, the WICED SDK includes a bunch of code to handle many tasks that you might want to use in your design. If you look in the “libraries” folder in the SDK Workspace you will find the following sub-folders:



- **Audio:** Contains support for Apollo (a streaming audio standard), and codecs including Free Lossless Audio Codec.
- **Crypto:** ECDH (Elliptic Curve Diffe-Hellman) and ECDSA (Elliptic Curve Digital Signature Algorithm) cryptography utilities
- **Daemons:** Contains some typical “Unix” daemons to provide networking support including an HTTP Server, Gedday, TFTP, DHCP, DNS etc.
- **Drivers:** Contains hardware support files for SPI flash, USB etc.
- **Filesystems:** FAT, FILE and other file systems that could be written to an SPI flash.
- **Graphics:** Support for the U8G OLED displays.
- **Inputs:** Drivers for buttons and GPIOs.
- **Protocols**: Support for application layer protocols including HTTP, COAP, MQTT etc.
- **Test**: Tools to test network performance, iPerf, malloc, TraceX, audio.
- **Utilities**: Support for JSON, linked lists, console, printf, buffers, etc.

4.2 U8G Graphics

In the exercises, we will be using the graphics library to display information on the OLED display present on the shield board.

To draw text to the display you must:

1. Setup a structure of type *u8g_t*. A pointer to this structure will be the first argument in almost all the u8g function calls that we use.
2. Setup and initialize an I2C structure for the OLED display. For our hardware:
    a. I2C port = WICED_I2C_2
    b. I2C address = 0x3C
    c. I2C address with = I2C_ADDRESS_WIDTH_7BIT
    d. Flags = 0
    e. Speed mode = I2C_STANDARD_SPEED_MODE
3. Initialize the I2C device using u8g_init_wiced_i2c_device. This function takes a pointer to the I2C structure from step 2.
4. Initialize the communication functions by calling *u8g_InitComFn*. It takes a pointer to the u8g structure created in step 1, a pointer to a u8g_dev_t structure which specifies the type of display, and a communication function pointer. For our hardware, if you have a display structure called “display”, the call looks like this:

    u8g_InitComFn(&display, &u8g_dev_ssd1306_128x64_i2c, u8g_com_hw_i2c_fn);

5. Select a font using *u8g_SetFont*. It takes a pointer to the u8g structure and the name of the font.
    a. The fonts are all listed in the file u8g_font_data.c in the graphics library directory. The examples use u8g_font_unifont, but feel free to experiment with others if you want.
6. Set a position using *u8g_SetFontPosTop*, *u8g_SetFontPosBottom*, or *u8g_SetFontPosCenter*.
    a. These functions determine where the characters are drawn relative to the starting coordinates specified in the DrawStr function described below. *u8g_SetFontPosTop* means that the top of the first character will be at the coordinate specified.
7. Each time you want to display a string you:
    a. Select the page to display the string using *u8g_FirstPage*.
    b. Draw the string using *u8g_DrawStr*. You must call this repeatedly until *u8g_NextPage* returns a 0. The u8g_DrawStr function takes a pointer to the u8g structure, X coordinate, Y coordinate, and the string to be printed.

As an example, assuming a display structure called “display” and an I2C structure called “display_i2c” the following will print the string “Cypress”:

```
u8g_init_wiced_i2c_device(&display_i2c);

u8g_InitComFn(&display, &u8g_dev_ssd1306_128x64_i2c, u8g_com_hw_i2c_fn);

u8g_SetFont(&display, u8g_font_unifont);

u8g_SetFontPosTop(&display);
```

```
u8g_FirstPage(&display);
do
{
    u8g_DrawStr(&display, 0, 10, "Cypress");
} while (u8g_NextPage(&display));
```

In addition, you must include “u8g_arm.h” in the .c file and you must include the u8g library in the .mk file to have access to the library functions:

In <project>.c :	#include u8g_arm.h
In <project>.mk:	$(NAME)_COMPONENTS := graphics/u8g

Note: u8g_arm.h includes wiced.h so you don’t need to include wiced.h separately but it doesn’t hurt to include both.

4.3 JavaScript Object Notation (JSON)

JSON is an open-standard format that uses human-readable text to transmit data. It is the de facto standard for communicating data to/from the cloud. JSON supports the following data types:

- Double precision floating point
- Strings
- Boolean (true or false)
- Arrays (use "[]" to specify the array with values separated by ",")
- Key/Value (keymap) pairs as "key":value (use "{}" to specify the keymap) with "," separating the pairs

Key/Value values can be arrays as well as key/value maps

Arrays can hold Key/Value Maps

For example, a legal JSON file looks like this:

```
{
        "name" : "alan",
        "age" : 49,
        "badass" : true,
        "children":  ["Anna","Nicholas"],
        "address" : {
                "number":201,
                "street": "East Main Street",
                "city": "Lexington",
                "state":"Kentucky",
                "zipcode":40507
        }
}
```

Note that carriage returns and spaces (except within the strings themselves) don't matter. For example, the above JSON code could be written as:

```
{"name":"alan","age":48,"badass":true,"children":["Anna","Nicholas"],"address":{"number":201,"street":"East Main Street","city":"Lexington","state":"Kentucky","zipcode":40507}}
```

While this is more difficult for a person to read, it is easier to create such a string in the firmware when you need to send JSON documents.

There is a website available which can be used to do JSON error checking. It can be found at:

https://jsonformatter.curiousconcept.com

4.4 WICED JSON Parsers in WICED

There are two JSON parsers built into the WICED library: *cJSON* and *JSON_parser*. cJSON is a Document Object Model Parser, meaning it reads the whole JSON in one gulp. The JSON_parser is iterative, and as such, enables you to parse larger files. You can find them in the SDK under *libraries/utilities/*.

4.4.1 WICED cJSON

cJSON reads and processes the entire document at one time, then lets you access data in the document with an API to find elements in the JSON. You can look at the README file which is found in libraries/utilities/cJSON/README. Your code will look something like:

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

void application_start()
{
   const char data[] = "{ \"initials\" : \"arh\", \"age\" : 49 }";
   cJSON *root;
   cJSON *myObj;
   root = cJSON_Parse(data);
   myObj = cJSON_GetObjectItem(root,"initials");
   WPRINT_APP_INFO(("Initials = %s\n",myObj->valuestring));
   myObj = cJSON_GetObjectItem(root,"age");
   WPRINT_APP_INFO(("Age = %f\n",myObj->valuedouble));
 }
```

To include cJSON in your project, add it to the Makefile:

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

4.4.2 WICED JSON_parser

The JSON_Parser library is an iterative parser, meaning that it reads one chunk at a time. This kind of parser is good for situations where you have very large structures where it is impractical to read the entire thing into memory at once but it is generally more difficult to use than the cJSON parser so we will not cover the details here.

To add it to your project Makfile

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

4.5 Exercise(s)

Exercise - 4.1 Browse the library directory to see what functions are available

Exercise - 4.2 Review the graphics library documentation and run the examples

1. Go to the documentation directory in the SDK (43xxx_Wi-Fi/doc) and open the WICED-LED_Display.pdf file. Review the documentation.
2. Copy the project from snip/graphics/hello to ww101/04/02_hello. Rename files and update the make file as necessary.
3. Verify that the I2C port is set to WICED_I2C_2.
4. Change the I2C speed to I2C_STANDARD_SPEED_MODE.
5. Review the rest of the project to understand what it is doing.
6. Create a make target for your project and run it.
7. Repeat the above steps for the graphicstest project.
    a. Hint: you will have to remove the VALID_PLATFORMS line from the make file (or add CYW943907*) in order to build the project.

Exercise - 4.3 (Advanced) Display sensor information on the OLED display

1. Copy 02_hello to 03_sensorData. Update the names and make target as necessary.
2. Update the code so that the temperature, humidity, ambient light, and potentiometer values are read from the analog co-processor and displayed to the screen every ½ second.
    a. Hint: see the I2C read exercise in chapter 2 for information on reading the sensor values using I2C.
    b. Hint: you will need to create two different I2C structures and initialize two I2C devices – one for the analog co-processor and one for the OLED display. They will both use the same physical interface (WICED_I2C_2) but not at the same time.
    c. Hint: use *snprintf* to format the strings. This safer than sprint because you tell it the max number of characters to output – there is no chance of over-running the buffer which can cause all sorts of odd behavior. The prototype is:

        int snprintf(char *buffer, size_t n, const char *format-string, argument-list);

        Note that the string produced includes a terminating null character so the size parameter must be large enough to hold the string plus the terminating null.

    d. Hint: If you are using threads, this would be a great place to use a mutex.

Exercise - 4.4 Parse a JSON document using the library “cJSON”.

Write a program that will read JSON from a given character array, then set the 4 I2C controlled LEDs (next to the CapSense buttons) and the 2 GPIO LEDs to the correct state based on the content of the document. The document will be a key/value map with the two keys “i2cleds” and “gpioleds”. The value of those keys will be another key/value map with one key per LED, e.g. “1” to represent the first LED. The value of those keys will be “on” or “off”.

Make a JSON string that represents the desired states of the 4 I2C LEDs and the 2 GPIO LEDs that looks like this:

```
{ "i2cleds" : { "1":"on", "2":"off", "3":"on", "4":"on"}, "gpioleds" : { "1":"off", "2":"on"}}
```

Remember that the quotes inside the string that you define need to be escaped with a backslash (i.e. \” instead of just “) or you will confuse the compiler.

Hint: refer to the I2C write example from the peripherals chapter for details on how to control the I2C LEDs.

Exercise - 4.5 (Advanced) Process a JSON document using “JSON_Parser”

Write a program that will parse the same JSON document as exercise (04), but using the JSON_Parser library.