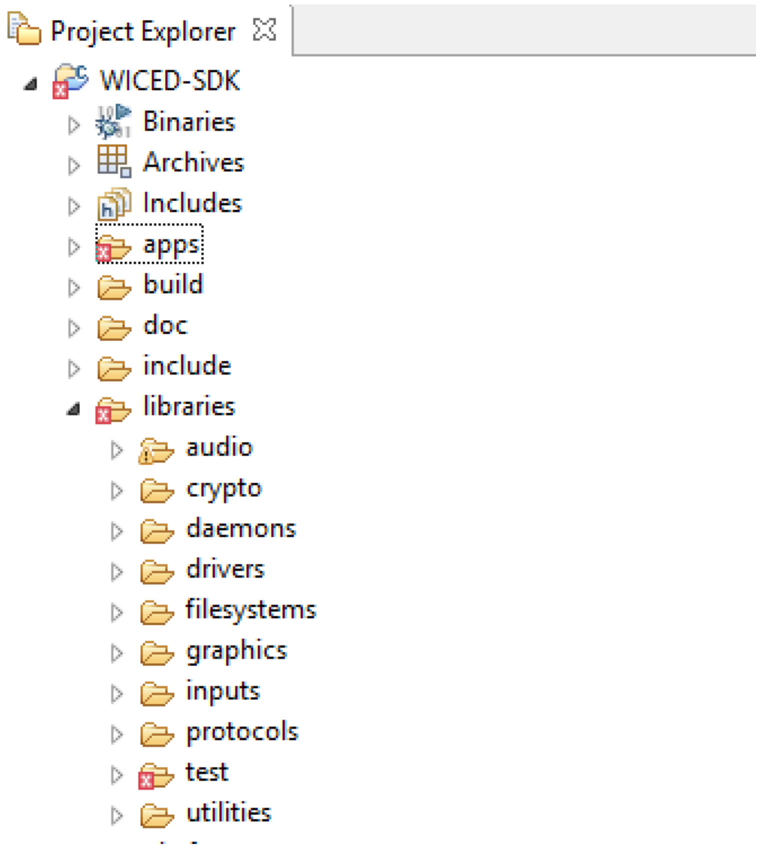
Chapter 4: Using the WICED-SDK Library

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:

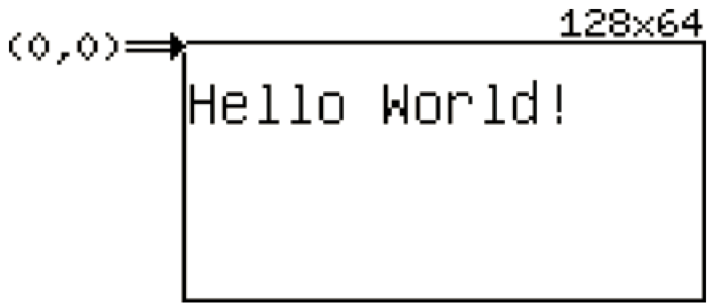


- **Audio:** Contains support for Apollo (a streaming audio standard), and codecs including Free Lossless Audio Codec.
- **Crypto:** ECDH (Elliptic Curve Diffe-Hellman) and ECDSA (Elliptic Curve Digital Signature Algorithm) cryptography utilities
- **Daemons:** Contains some typical "Unix" daemons to 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. The display is a 128x64 pixel OLED screen and an I2C based Freetronics SSD1306 driver chip. The I2C interface of the Freetronics SSD1306 is connected to the same I2C bus as the PSoC 4, but at a different I2C address. The Freetronics SSD1306 on our display has a fixed I2C address of 0x3C.

The OLED allows you to draw shapes and write text. When you use a function such as u8g_DrawLine() the x and y coordinates start in the top left corner (0, 0), like this.



Updates always occur in what is known as a "picture loop" where the drawing functions are repeatedly executed until the OLED is properly updated. Two library functions - u8g_FirstPage() and u8g_NextPage() - make this very simple by using a do-while loop as you will see below. This is done so that the data needed to fill the entire screen (128x64 = 8192 pixels) can be broken into multiple "pages" which reduces the amount of memory required on the host processor to create and send the image.

The steps to draw text to the display are:

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.
      i. To draw to the screen efficiently, the *u8g* graphics package divides the screen into a few horizontal sections, or pages. Using *u8g_FirstPage* and *u8g_NextPage* iterates over these pages to draw the full image.

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 pair values can be arrays or can be other key/value pairs

Arrays can hold Key/Value pairs

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":49,"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.

Unfortunately, quotes mean something to the C compiler so if you are including a JSON string inside a C program you need to escape the quotes that are inside the JSON with a backslash (\). The above JSON would be represented like this inside a C program:

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

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 Creating JSON

If you need to create JSON to send out (e.g. to the cloud) you can create a string using snprintf. You can use standard printf formatting to substitute in values for variables. For example, the following could be used to send the temperature as a floating point value from an IoT device to a cloud service provider:

```
char json[100];

snprintf(json, sizeof(json), "{\"state\" : {\"reported\" : {\"temperature\":%.1f} } }",
psoc_data.temperature);
```

The %.1f is replaced in the string with psoc_data.temperature as a floating point value with one place after the decimal. If the actual temperature is 25.4, the resulting string created in the array *json* would be:

```
{"state" : {"reported" : {"temperature":25.4} } }
```

4.5 Reading JSON (WICED JSON Parsers)

If you need to receive JSON (e.g. from the cloud) and then pull out a specific value, you can use a JSON parser. A parser will read the JSON and will then find and return values for keys that you specify.

There are two JSON parsers in 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/*. cJSON is easier to use but it may be necessary to use JSON_parser for very large JSON files. For IoT devices, cJSON will almost always be sufficient.

4.5.1 WICED cJSON Library

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. You will traverse the JSON hierarchy one level at a time until you reach the key:value pair that you are interested in. The functions generally return a pointer to a structure of type cJSON which has elements for each type of return data (i.e. valuestring, valueint, valuedouble, etc.)

For example, if you have a char array called ***data*** with the JSON related to Alan:

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

The code to get Alan's zip code would look something like:

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

void application_start()
```

```
{
    int zipcodeValue;
    cJSON *root = cJSON_Parse(data); //Read the JSON
    cJSON *address = cJSON_GetObjectItem(root,"address"); // Search for the key "address"
    cJSON *zipcode = cJSON_GetObjectItem(address,"zipcode"); // Search for the key "zipcode" under address
    zipcodeValue = (float) zipcode->valueint; // Get the integer value associated with the key zipcode
}
```

To include the cJSON library in your project:

1. Include cJSON.h in the C source file:

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

2. Add it to the Makefile:

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

4.5.2 WICED JSON_parser Library (Advanced)

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. You won't normally need it for IoT devices since they typically transmit data in small batches. To use it you:

1. Use *wiced_JSON_parser_register_callback* to register a callback function that is executed whenever a JSON item is received.
2. Use wiced_JSON_parser to pass in the JSON data itself.
3. Wait for the callback function to be called and process the data as necessary.

The callback function receives a structure of the type wiced_json_object_t which is:

```
typedef struct json_object{
    char*               object_string;
    uint8_t             object_string_length;
    wiced_JSON_types_t  value_type;
    char*               value;
    uint16_t            value_length;
    struct json_object* parent_object;
} wiced_json_object_t;
```

You can use conditional statements to check the name of the object that was received, check the type of value received, or even check values of parent objects.

The value types are:

```
typedef enum
{
    JSON_STRING_TYPE,
    JSON_NUMBER_TYPE,
    JSON_VALUE_TYPE,
    JSON_ARRAY_TYPE,
    JSON_OBJECT_TYPE,
    JSON_BOOLEAN_TYPE,
    JSON_NULL_TYPE,
```

```
    UNKNOWN_JSON_TYPE
} wiced_JSON_types_t;
```

Note that the value itself is returned as a string (char*) no matter what so if you will need to use atof to convert the string to a floating-point value or atoi to convert to an integer if that is what you need.

You must make sure a parent_object is not NULL before trying to access it or else your device will reboot.

Using the previous example, if you have an array called ***data*** with the JSON related to Alan:

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

The code to get Alan's zip code would look like:

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

float zipcodeValue;
char zipcodeString[6];

wiced_result_t jsonCallback(wiced_json_object_t *obj_p)
{
    /* Verify that the JSON path is address: zipcode and that zipcode is a number */
    if( (obj_p->parent_object != NULL) &&
        (strncmp(obj_p->parent_object->object_string, "address", strlen("address")) == 0) &&
        (strncmp(obj_p->object_string, "zipcode", strlen("zipcode")) == 0 ) &&
        (obj_p->value_type == JSON_NUMBER_TYPE) )
    {
        /'* Get zipcode value and convert to an integer */
        snprintf(zipcodeString, (obj_p->value_length)+1, "%s", obj_p->value);
        zipcodeValue = atoi(zipcodeString);
    }
    return WICED_SUCCESS;
}

void application_start()
{
    wiced_JSON_parser_register_callback(jsonCallback);
    wiced_JSON_parser(data, strlen(data));
}
```

To include the JSON_parser library in your project:

1. Include JSON.h in the C source file:

   ```
   #include <JSON.h>
   ```

2. Add it to the Makefile:

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

4.6 Exercise(s)

Exercise - 4.1 Browse the libraries folder 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. Remove the VALID_PLATFORMS line from the make file (or add WW101_*).
4. Review the rest of the project to understand what it is doing.
5. Create a make target for your project and run it.
6. Repeat the above steps for the graphicstest project.

Exercise - 4.3 Parse a JSON document using the library "cJSON".

Write a program that will read JSON from a hard-coded character array, parse out specific values, and print them to a UART terminal window.

1. Make a JSON string that contains the reported temperature. In a real IoT device, this would likely have been received from the cloud, but for now we will just hard-code it:

   ```
   const char *jsonString = "{\"state\" : {\"reported\" : {\"temperature\":25.4} } }";
   ```

2. Use the cJSON parser to get the value of the temperature and print it to a terminal window using WPRINT_APP_INFO.

Exercise - 4.4 (Advanced) Display sensor information on the OLED display

Read the sensor data from the PSoC on the shield using I2C and then use the u8g library to display the information to the 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: This would be a great place to use threads – one for reading sensor data from the PSoC, and one for updating the OLED display.

i. Make sure you use a mutex to prevent both threads from accessing the I2C at the same time.
ii. Use a queue to pass the sensor data from the thread that reads it to the thread that displays it.

d. Hint: use *snprintf* to format the strings. This is 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.

Exercise - 4.5 (Advanced) Process a JSON document using "JSON_parser"

Repeat the cJSON exercise using the JSON_parser library.