Chapter 4a: Cloud Protocols: Summary

This chapter will help you understand “the Cloud” and the basics of cloud protocols, which are also called application layer protocols. These protocols include HTTP, MQTT, AMQP, and CoAP. This chapter provides a summary of the protocols. Chapters 4B and 4C will cover HTTP and MQTT, respectively, in greater detail. AMQP and CoAP may be covered in future revisions of this material.

Table of contents

Document conventions

Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

4a.1 "The Cloud"

What is the Cloud? The Cloud is a simple name for a giant amalgamation of all the stuff that you need in order to provide websites and other network-based services (e.g. iTunes). Why do you need the Cloud? When you try to service large numbers of people and devices you have a very difficult and expensive problem. To have a fast and always available system, you need to have enough networks, disk drives, computers, and people to run it all. The solution to this problem is a standardized, shared, scalable system: The Cloud.

The term "The Cloud" generally includes:

- Networks (high bandwidth, worldwide, distributed)
- Storage (disks, databases)
- Servers (running Windows and Unix)
- Security
- Scalability
- Load Balancing
- Fault tolerance (redundancy)
- Management tools (reports, user identity management, etc.)
- Software (Web servers, APIs, Languages, Development tools etc.)

To make something interesting you need to be able to hook up your IoT device(s) (the "T" in IoT) to the Cloud (the "I" in IoT) which is the goal of this chapter.

4a.2 Application layer protocols

How do you get data to and from the Cloud? Simple, there are a number of standardized application layer protocols to do that task.

4a.2.1 Hyper Text Transfer Protocol (HTTP)

HTTP (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) is a text-based Application Layer Protocol that operates over TCP Sockets. It can perform the following functions:

- GET (retrieve data) from a specific place
- POST (put data) to a specific place
- HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH (less commonly used)

To initiate these commands, you open a socket typically to TCP port 80, send the text-based command (CRLF terminated) and read the replies. This request/reply protocol is used for every command. Replies are sent with a resulting Content-Type string which indicates the type of data encoding for the response. The content-type string uses a Multipurpose Internet Mail Extension (MIME) type to indicate the type of data being received (e.g. text/html or image/jpeg).

For instance, you can send an HTTP GET request followed by the Host header to open "/index.html" on www.example.com:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

www.example.com will respond with:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close
<html>
<head>
<title>An Example Page</title>
</head>
<body>
Hello World, this is a very simple HTML document.
</body>
</html>
```

It is possible (and semi-common) to build IoT devices that use HTTP to "PUT" their data to webservers in the Cloud and "GET" their instructions/data from webservers. However, HTTP is somewhat heavy (that is, bandwidth intensive) and is generally being displaced by other protocols that are more suited to IoT.

4a.2.2 Message Queueing Telemetry Transport (MQTT)

MQTT (https://en.wikipedia.org/wiki/MQTT) is a lightweight messaging protocol that allows a device to **Publish Messages** to a specific **Topic** on a **Message Broker**. The Message Broker will then relay the message to all devices that are **Subscribed** to that **Topic**.

The format of the messages being sent in MQTT is unspecified. The message broker does not know (or care) anything about the format of the data and it is up to the system designer to specify an overall format of the data. All that being said, JavaScript Object Notation (JSON) has become the lingua franca of IoT.

MQTT operates on TCP Ports 1883 for non-secure and 8883 for secure (TLS).

Cloud providers that support MQTT include Amazon AWS and IBM Bluemix.

4a.2.2.1 Topic

A Topic is simply the name of a message queue e.g. "mydevice/status" or "mydevice/pressure". The name of a topic can be almost anything you want but by convention is hierarchical and separated with slashes "/".

4a.2.2.2 Publish

Publishing is the process by which a client sends a message as a blob of data to a specific topic on the message broker.

4a.2.2.3 Subscription

A Subscription is the request by a client device to have all messages published to a specific topic sent to it.

4a.2.2.4 Message Broker

A Message Broker is just a server that handles the tasks:

- Establishing connections (MQTT Connect)
- Tearing down connections (MQTT Disconnect)
- Accepting subscriptions to a Topic from clients (MQTT Subscribe)
- Turning off subscriptions (MQTT Unsubscribe)
- Accepting messages from clients and pushing them to the subscribers (MQTT Publish)

4a.2.2.5 Quality of Service (QOS)

MQTT provides three levels of QOS:

- Level 0: At most once (each message is delivered once or never)
- Level 1: At least once (each message is certain to be delivered, possibly multiple times)
- Level 2: Exactly once (each message is certain arrive and do so only once)

4a.2.3 Advanced Message Queuing Protocol (AMQP)

AMQP (https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol) is a binary application layer protocol designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as *at-most-once*, *at-least-once* and *exactly-once*, and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardized but extensible 'messaging capabilities.'

Cloud providers that use AMQP include Microsoft (e.g. Windows Azure), VMWare, and Redhat.

4a.2.4 Constrained Application Protocol (CoAP)

CoAP (https://en.wikipedia.org/wiki/Constrained_Application_Protocol) makes use of two message types, requests and responses, using a simple, binary, base header format. The base header may be followed by options in an optimized Type-Length-Value format. CoAP is by default bound to UDP and optionally to DTLS, providing a high level of communications security.

Any bytes after the headers in the packet are considered the message body, if any. The length of the message body is implied by the datagram length. When bound to UDP the entire message MUST fit within a single datagram. When used with 6LoWPAN as defined in RFC 4944, messages SHOULD fit into a single IEEE 802.15.4 frame to minimize fragmentation.

The mapping of CoAP with HTTP is also defined, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way.

Cloud providers that use CoAP include Samsung ARTIK.

4a.3 JavaScript Object Notation (JSON)

JSON (https://en.wikipedia.org/wiki/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:

- Integers
- 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\"],\"addr
ess\":{\"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

4a.4 Creating JSON

If you need to create JSON to send out (e.g. to the cloud) you can create a string using `snprintf` with standard formatting codes 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} } }

4a.5 Reading JSON using a parser

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.

The JSON parser in ModusToolbox™ is called *JSON_parser*, and is included as a part of the connectivity-utilities library. However, in this class we will also cover the JSON parser *cJSON* which was created in 2009 by Dave Gamble which you can find online. A copy of this library that has been verified to work with PSoC™ 6 is included with this class in the *Libraries* directory. *cJSON* is a Document Object Model Parser, meaning it reads the whole JSON in one gulp, whereas *JSON_parser* is iterative, and as such, enables you to parse larger files. *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.

4a.5.1 cJSON library

The *cJSON* library reads and processes the entire document at one time, then lets you access data in the document with an API to find elements. 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 (a.k.a. postal code) would look like this:

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "cJSON.h"

int main(void)
{
     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 = zipcode->valueint; // Get the integer value associated
                                             with the key zipcode
}
```

To include the cJSON library in your project:

1. Copy the *cJSON* directory from the *Libraries* directory of this class into your application's root directory.
2. Include *cJSON.h* in the C source file:

```
#include "cJSON.h"
```

4a.5.2 JSON_parser library

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 complex 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 `cy_JSON_parser_register_callback` to register a callback function that is executed whenever a JSON item is received.
2. Use `cy_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 `cy_JSON_object_t` which is:

```
typedef struct cy_JSON_object {

    char*                 object_string;          /**< JSON object as a string */
    uint8_t               object_string_length;   /**< Length of the JSON string */
    cy_JSON_type_t        value_type;             /**< JSON data type of value parsed */
    char*                 value;                  /**< JSON value parsed */
    uint16_t              value_length;           /**< JSON length of value parsed */
    struct cy_JSON_object*  parent_object;        /**< Pointer to parent JSON object */
} cy_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
} cy_JSON_types_t;
```

Note that the value itself is returned as a string (`char*`) no matter what so 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 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 like this:

```
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "cy_retarget_io.h"
#include "cy_json_parser.h"

float zipcodeValue;
char zipcodeString[6];

cy_rslt_t jsonCallback(cy_JSON_object_t *obj_p, void *arg){
     /* 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 CY_RSLT_SUCCESS ;
}

int main(void){
    cy_JSON_parser_register_callback(jsonCallback, NULL);
    cy_JSON_parser(data, strlen(data));
}
```

To use the *JSON_parser* library in your project:

1. Verify that you have the *wifi-connection-manager* library in your project. That library includes the *freertos* library and the *wifi-mw-core* library. The *wifi-mw-core* library includes the *connectivity-utilities* library containing the JSON parser functions.
    a. Copy *FreeRTOSConfig.h* from *mtb_shared/freertos/release-vX.X.X/Source/portable/COMPONENT_CM4* to your root project directory and delete the line that starts with `#warning`.
    b. Copy the config files from *mtb_shared/wifi-mw-core/configs* to your root project directory and update the *Makefile* by adding:
    ```
    COMPONENTS=FREERTOS LWIP MBEDTLS
    DEFINES+=MBEDTLS_USER_CONFIG_FILE='"mbedtls_user_config.h"'
    ```
2. Include *cy_json_parser.h* in the C source file:
    ```
    #include "cy_json_parser.h"
    ```

4a.6 Exercises

Exercise 1: Exercise 1: 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. Create a new project called **ch01_ex01_cJSON** based on the Empty PSoC™ 6 template.
2. Add the *cJSON* library to your application.
   a. Copy the *cJSON* directory from the *Libraries* directory of this class into your application's root directory.
   b. In *main.c*:
   ```
   #include "cJSON.h"
   ```
3. 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} } }";
   ```
4. Use the cJSON parser to get the value of the temperature and print it to a terminal window using `printf`.

Note: *Refer to the section titled cJSON library.*

Note: *Don't forget to add and initialize the retarget-io library.*

Exercise 2: Process a JSON document using "JSON_parser"

Repeat the cJSON exercise using the *JSON_parser* library.

Don’t forget to:

- Copy *FreeRTOSConfig.h* from *mtb_shared/freertos/release-vX.X.X/Source/portable/COMPONENT_CM4* to your root project directory and delete the line that starts with `#warning`.
- Copy the config files from *mtb_shared/wifi-mw-core/configs* to your root project directory and update the *Makefile* by adding:
  ```
  COMPONENTS=FREERTOS LWIP MBEDTLS
  DEFINES+=MBEDTLS_USER_CONFIG_FILE='"mbedtls_user_config.h"'
  ```

4a.7 Further reading

- [2] RFC2045 – "Multipurpose Internet Mail Extensions"; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc2045
- [4] RFC2616 – "Hypertext Transfer Protocol (HTTP) " ; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc2616
- [5] RFC7159 – "The Javascript Object Notation (JSON) Data Interchange Format"; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc7159
- [6] MQTT - http://mqtt.org/
- [7] RFC7959 – "The Constrained Application Protocol (CoAP)" ; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc7252
- [8] AMQP - http://www.amqp.org/

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Published by

Infineon Technologies AG

81726 Munich, Germany

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”) .

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.